

**Birgit Vera SCHMIDT**

**Optimal Strategies in  
Deduction Games  
with Situation-Dependent  
Cost Functions**

**DIPLOMA THESIS**

written to obtain the academic degree of a  
Master of Science (MSc)

Diploma Study in Technical Mathematics



Graz University of Technology

**Technische Universität Graz**

**Supervisor:**

**Assoc.Prof. Dipl.-Ing. Dr.techn. Oswin AICHHOLZER**

**Institute for Software Technology**

**Graz, January 2012**

## EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am .....  
(Datum) .....  
(Unterschrift)

## STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used any other sources/resources than those declared as such, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, .....  
(date) .....  
(signature)

*The child is laughing: The game is my wisdom and my love.  
The young is singing: The love is my wisdom and my game.  
The old is silent: The wisdom is my love and my game.*

– Lucian Blaga

# Preface and Acknowledgments

Let me start with a story. It was in 2004 that I signed up for an online gaming site and discovered their implementation of BlackBox. After around 1300 games I had painstakingly lowered my initial average score of 22 to 18.9 (lower scores are better), which I was immensely proud of. It was only then that I looked at the global highscore table and noticed that the first place in the ranking was held, with a fair margin to the second place, by a player with an average of 11.52. And not merely due to having played two or three games and having been extremely lucky in them – no, an average of 11.52 over 2789 games! I was incredulous. Surely, that cheater must have hacked the system and simply read the correct answer from the internal state of the program. So I set out to prove that an average of 11.52 was not achievable by legal means.

Now, seven years later, I still do not know what the best legally achievable average BlackBox score is. I have however proven in the process that the problem is NP-hard, and have successfully bludgeoned it from a calculation effort of several billions of years down to mere decades. Hopefully I therefore will, within my lifetime, find out whether an average score of 11.52 is achievable.

As a nice side effect, I have gathered enough optimization tricks to fill this entire thesis with it. Since the problem is related to several problems in the field of artificial intelligence, I hope that some of these tricks might be useful to someone else as well, possibly applied to problems I have not even heard of yet.

Since this is the end of a long time at university, there are many people whom I wish to thank for their support along the way:

I would like to thank a number of professors, including Peter Auer, Bettina Klinz, Klaus Schmaranz and Volker Ziegler, for their captivating lectures at university. In addition to that, I want to thank Peter Grabner and Clemens Heuberger not only for their teaching, but also for all their support in organizational matters.

My thanks furthermore go to a handful of professors, especially Oswin Aichholzer and Karl-Christian Posch, who allowed me to be teaching assistant in their courses and who have been among the most inspiring supervisors I had the pleasure to work with.

I wish to thank all teachers and professors involved in the Austrian Mathematical Olympiad; I would not have studied mathematics if it had not been for them and all their hard work in preparing and organizing courses and competitions.

With all my heart I thank my friends who give me the courage to be who I am, and my family for their continued support in everything I do.

As far as the creation of this thesis is concerned, I thank the kind people who mercilessly emptied their red pens (or virtual equivalents) on proofreading it: Thomas Hackl, Lukas Daniel Klausner, Johann Schmidt, Juho Snellman, and Martin Windischer. Their input has been invaluable for improving it. Any imperfections, mistakes or omissions that remain are due to me.

Finally, I would like to wholeheartedly thank my advisor, Oswin Aichholzer, for allowing me to write this thesis on a topic of my choice, and for his support, patience and guidance in creating it.

Graz, January 2012

BIRGIT VERA SCHMIDT

# Abstract

A DEDUCTION GAME is a game in which one player – the CODE-MAKER or simply COMPUTER – secretly chooses a situation, and the other player – the CODE-BREAKER or simply PLAYER – has to deduct it by asking questions. After some questions, he has to take a guess, and gets a penalty (if he is wrong) in addition to the question costs. Popular games of this type are MasterMind, BlackBox and BattleShips.

We will show several ways to reduce the calculation effort of finding optimal strategies for these games. First of all we will prove that the best question to ask next only depends on the *set of situations that are still considered possible* after the questions that were already asked, but not on those questions themselves. Using this, we will introduce the DECISION SET GRAPH which has these sets of remaining situations as nodes and reflects the structure of the original game tree otherwise. We will define an algorithm to find best strategies on this decision set graph, and show how these can be translated back to the original game tree.

We furthermore show several upper bounds for the number of nodes in the decision set graph, and prove that calculating the best strategy is NP-hard.

Finally, we will look at some implementation tricks: simplifying weights so that we can use integers instead of real numbers for all calculations, introducing upper bounds so that we can abort calculation of branches that are certain not to be optimal, describing situations and sets thereof by using fingerprints that can be stored as simple bitmasks, as well as some techniques for caching and distributed computation.

All of this cannot hide the fact that the problem is still NP-hard. We therefore might be able to give significant performance improvements – for a game with  $s$  situations,  $q$  questions, and  $a$  possible answers per question we find, among others, upper bounds of  $O(2^qs)$  and  $O((1+a)^q)$  for the number of calculated nodes, compared to  $O(sq!)$  and  $O(q!a^q)$  in the two trivial implementations –, but we are nonetheless still facing very high runtimes. Consequently, using heuristic approaches will be preferable in practical applications. This thesis does provide a well-optimized way of calculating exact solutions that can among other things be used for evaluating such heuristics.

An implementation of the described algorithms is included in the thesis.

# Overview

In Chapter 1 we describe and formally define the class of deduction games, and in Chapter 2 we show how several well-known games, as well as problems from classical computer science, artificial intelligence, and industry, can be represented in this setting.

Only after that we have a look at existing work in Chapter 3 (since we will need many of the game definitions in the discussion of existing literature).

The core chapter of this thesis is Chapter 4, in which we show that the best next move only depends on the *set of situations that are still considered possible* after the questions that we have asked already. In particular, it is not important *which* questions have been asked before, let alone in what order they were asked. Since there are many nodes in the game tree in which the same situations are still considered possible, we define and create the DECISION SET GRAPH, in which all such nodes are “merged” into one common node. We show how to find best strategies on this graph, and how to translate them back to strategies on the original game tree. We also show that we can calculate the expected cost of a strategy directly in this decision set graph without transforming it back at all.

In Chapter 5 we show several upper bounds for the number of nodes in the decision set graph. Since these can depend both on the number of situations  $s$  and the number of questions  $q$  that a game has, as well as the structure of those questions and situations, there exist many upper bounds of which none can be said to be best in all cases. For example, we find upper bounds  $2^s$  and  $2^q s$ ; for a small  $q$ , the latter bound will be lower, whereas for a large  $q$ , the former will be lower.

Chapter 6 shows several tricks how to make these calculations fast in practice. In particular, we

- show how we can calculate everything we need without having to keep the entire decision set graph in memory;
- simplify the weights so that we can use integers for all calculations instead of real numbers or fractions;
- introduce upper bounds so that we can abort the calculation of a branch as soon as we find out that it is certain not to be optimal;
- introduce fingerprints and fingerprint masks that allow us to describe situations and sets thereof using simple bitmasks, and that additionally help to simplify calculating the children of a node;
- shortly discuss how known partial results can be cached efficiently if we do not have enough memory to store them all at the same time; and
- look at ways how the calculation can be distributed.

As a part of that chapter we find out that calculating the best estimate (when taking a guess) for a set of remaining situations efficiently is important. In Chapter 7, we therefore find such methods for each game discussed in Chapter 2.

We do a rough performance comparison to heuristic strategies and random strategies, which are described in Chapter 8.

Last but not least, Chapter 9 describes details about the algorithm implementation that accompanies this thesis, Chapter 10 presents some results of calculations with this implementation, and Chapter 11 presents conclusions and outlook.

# Contents

<b>Preface and Acknowledgments</b>	<b>4</b>
<b>Abstract</b>	<b>5</b>
<b>Overview</b>	<b>6</b>
<b>1 Introduction and Problem Statement</b>	<b>10</b>
1.1 What is a deduction game? . . . . .	10
1.2 Best strategies . . . . .	10
1.3 Question- and result-dependent costs . . . . .	11
1.4 Estimates . . . . .	11
1.5 Formal problem definition . . . . .	12
<b>2 Embedding popular games into this model (including game descriptions)</b>	<b>14</b>
2.1 Number guessing . . . . .	14
2.2 Number guessing with bit questions . . . . .	15
2.3 Number guessing with fixed penalty . . . . .	16
2.4 Number guessing with estimates . . . . .	16
2.5 Number guessing with different question costs . . . . .	16
2.6 MasterMind / Bulls and Cows . . . . .	17
2.7 BattleShips . . . . .	19
2.8 Minesweeper . . . . .	21
2.9 Minesweeper with free empty fields . . . . .	23
2.10 BlackBox . . . . .	24
2.11 Classification . . . . .	27
2.12 Classification with different question costs . . . . .	27
2.13 Life . . . . .	28
2.14 Production parameters . . . . .	30
2.15 Scales . . . . .	31
2.16 Milk . . . . .	31
<b>3 Background</b>	<b>34</b>
3.1 Research about deduction games in general . . . . .	34
3.2 Known results about particular games . . . . .	34
3.2.1 MasterMind . . . . .	34
3.2.2 Minesweeper . . . . .	35
3.2.3 BattleShips . . . . .	35
3.2.4 BlackBox . . . . .	36
3.3 Decision tree learning . . . . .	36
<b>4 Reducing the number of distinct states by transformation into two equivalent games</b>	<b>38</b>
4.1 Basic definitions from game theory . . . . .	38
4.1.1 Properties of deduction games . . . . .	38
4.1.2 Game trees . . . . .	39
4.1.3 Solving game trees . . . . .	40

4.1.4	Dealing with randomness	40
4.1.5	Dealing with imperfect information	41
4.1.6	Types of solutions	41
4.2	Game tree of a deduction game	42
4.2.1	Information sets	44
4.2.2	Expected costs	46
4.2.3	Payoffs	46
4.2.4	Finding optimal strategies and redefining payoffs	48
4.2.5	Situation subtrees	51
4.3	Decision sets	51
4.3.1	Ignoring the order of previously asked questions	52
4.3.2	Considering only the set of still possible situations	52
4.4	An equivalent game	54
4.4.1	Equivalence to original game	57
4.4.2	Calculating optimal strategies	60
4.4.3	Decision sets	63
4.5	Decision set graph	64
4.5.1	Solving the game on the decision set graph	70
<b>5</b>	<b>Upper bounds for the number of nodes in the decision set graph</b>	<b>73</b>
5.1	Relationship between number of player, computer and estimate nodes	73
5.2	Relationship between number of nodes and calculation effort	74
5.3	Upper bounds for the number of player nodes	74
5.3.1	Upper bound: $n \leq 2^s - 1$	75
5.3.2	Upper bound: $n \leq s \cdot q! \cdot e$	76
5.3.3	Upper bound: $q! \cdot a^q \cdot e^{\frac{1}{a}}$	76
5.3.4	Upper bound: $n \leq 2^q s - s + 1$	77
5.3.5	Upper bound: $n \leq 2^{aq}$	81
5.3.6	Upper bound: $n \leq (1 + a)^q$	82
5.3.7	Upper bounds in the hypercuboid	84
5.3.8	Summary of upper bounds	104
5.4	NP-Hardness	104
<b>6</b>	<b>Implementation tricks and details</b>	<b>109</b>
6.1	Input	109
6.2	The trivial implementation	112
6.2.1	Constructing the decision set graph	112
6.2.2	Calculate the expected outcome	113
6.2.3	A small optimization	114
6.3	Implicit decision set graph	115
6.4	Simplified weights	118
6.5	Branch and bound	127
6.5.1	Limits in player nodes	127
6.5.2	Limits in computer nodes for average-optimal solutions	132
6.5.3	Limits in computer nodes for worst-case-optimal solutions	134
6.5.4	Randomization	135
6.6	Fingerprinting	136
6.6.1	Creating situation fingerprints	136
6.6.2	Expressing remaining situation lists in fingerprints	145
6.6.3	Greatest common fingerprint mask	148
6.6.4	Calculating children from fingerprint masks	152
6.6.5	Storing and retrieving known results	157
6.6.6	A critical word on the performance of greatest common fingerprint masks	157
6.6.7	Performance comparison	158
6.7	Caching known results	159
6.7.1	Caching by node size	159



6.8	Distributed calculation . . . . .	160
6.8.1	Splitting after the first few questions . . . . .	161
6.8.2	Variations of this method . . . . .	162
6.9	Summary . . . . .	162
<b>7</b>	<b>Efficiently finding best estimates for particular games</b>	<b>163</b>
7.1	Situation has to be guessed exactly . . . . .	164
7.2	Part of situation has to be guessed exactly . . . . .	164
7.3	Number guessing with finite penalty . . . . .	165
7.4	Number guessing with estimates . . . . .	165
7.5	Number guessing with estimates and bit questions . . . . .	166
7.6	BlackBox . . . . .	167
7.7	Life (both variants, first way to define costs) . . . . .	168
7.8	Milk . . . . .	170
<b>8</b>	<b>Not optimal strategies</b>	<b>171</b>
8.1	Heuristic methods . . . . .	171
8.1.1	Greedy . . . . .	172
8.1.2	Information Gain . . . . .	172
8.1.3	Gini coefficient . . . . .	173
8.2	Random . . . . .	174
<b>9</b>	<b>Implementation details</b>	<b>175</b>
9.1	BigIntegerInfinity and BigIntegerInfinityFraction . . . . .	175
9.2	Input . . . . .	175
9.3	Analysis . . . . .	176
<b>10</b>	<b>Results</b>	<b>178</b>
10.1	Upper bounds . . . . .	178
10.2	Runtime comparison . . . . .	180
10.2.1	Behavioural game tree vs. Decision set graph . . . . .	180
10.2.2	Original weights vs. Simplified weights . . . . .	181
10.2.3	Calculation of all children vs. Upper bounds . . . . .	181
10.2.4	Given question order vs. Randomized question order . . . . .	182
10.2.5	Normal question evaluation vs. Question evaluation with fingerprints . . . . .	183
10.2.6	Lists of remaining situations vs. Greatest common fingerprint masks . . . . .	183
10.2.7	Summary . . . . .	184
10.3	Heuristics and random strategies . . . . .	184
10.4	MasterMind ( $4 \times 4$ ) . . . . .	185
10.5	BlackBox ( $8 \times 5$ ) . . . . .	185
10.5.1	Calculating fingerprints . . . . .	185
10.5.2	Estimating the number of nodes after three questions . . . . .	186
10.5.3	Estimating the calculation time for a subgame after three questions . . . . .	186
10.5.4	Estimating the total calculation time . . . . .	186
<b>11</b>	<b>Conclusions and outlook</b>	<b>187</b>
	<b>Index</b>	<b>189</b>
	<b>Bibliography</b>	<b>189</b>
<b>A</b>	<b>DVD</b>	<b>192</b>

# Chapter 1

## Introduction and Problem Statement

### 1.1 What is a deduction game?

**Definition 1.1** (deduction game). A DEDUCTION GAME is a game in which one player chooses a secret setup, and the other player has to deduct this setup by asking questions [20].

(Note that in some literature the term DEDUCTIVE GAME is used instead.)

The most popular game of this type is MasterMind, in which one player chooses a combination of four colour pegs, and the other player has to deduct this combination by trying combinations of four colours and being told the number of correct colours in each guess. A much simpler “game” of this type could be one person choosing a number between 1 and 100, and the other person trying to deduct that number by asking questions of the form “Is the number larger than  $x$ ?”. Another game that is very close to being a deduction game is BattleShips, in which one player “hides” his ships on a grid, and the other player tries to shoot them by bombarding grid fields and being told whether he hit part of a ship or just water. (The only two differences that prevent this from being a pure deduction game are that both players hide ships and shoot alternately, and that a player has to hit all ship parts rather than only determine where the ships are. We will later see that these two differences are (almost) negligible and turn BattleShips into a pure deduction game.) In the royal league of deduction games we finally have BlackBox, the game that motivated the author of this thesis to analyze this class of games. (The rules for all games analyzed in this thesis can be found in Chapter 2.)

For obvious reasons, deduction games are often played as solitary games against a computer, since the player choosing the secret combination is in fact not really *playing*, but rather just evaluating questions. In this thesis we will therefore simply use COMPUTER and PLAYER to refer to the player choosing the combination (in literature also often called CODE-MAKER) and the player deducting the combination (accordingly sometimes called CODE-BREAKER in literature), respectively.

### 1.2 Best strategies

Obviously, there are better and worse strategies in such a game. Let us stick with the number guessing game as an example. The computer randomly chooses a number between 1 and 100, and the player can then ask questions of the form “Is the number larger than  $x$ ?” (for every integer  $x$  from 1 to 99). A trivial (but not very efficient) strategy for the player would be to first ask whether the number is larger than 1. If not, the number is 1 and he is done, otherwise he next asks whether the number is larger than 2, then whether it is larger than 3, larger than 4, larger than 5, and so on. Obviously, that can take a rather long time. Assuming that the computer chooses each number between 1 and 100 with the same probability, it takes on average 50.49 questions<sup>1</sup>.

Or, the player could do a binary search, first asking whether the number is larger than 50. If yes, he next asks whether it is larger than 75, otherwise whether it is larger than 25, and so on, roughly halving the

---

<sup>1</sup>If the chosen number is 1, he needs 1 question, if it is 2 he needs 2 questions, and so on, until 98 for which he needs 98 questions, 99 for which he needs 99 questions, and 100 for which he needs 99 questions as well. He therefore needs on average  $\frac{1+2+3+\dots+98+99+99}{100} = 50.49$  questions.

interval length with each question. On average he now needs only 6.72 questions<sup>2</sup>.

Or, he could first ask whether the number is larger than 64. If not, he does a binary search on the interval from 1 to 64 with the advantage that in each step, he can split the interval exactly into two intervals that are again powers of two, needing exactly 6 more questions. If the number is between 65 and 100, he just does a classical binary search on the remaining interval. In total, he on average needs 6.72 questions again<sup>3</sup>, but with this strategy, it is easier to predict after the first question how much longer it will take.

The objective of this thesis is to efficiently find optimal strategies, or at least calculate the average outcome expected with such strategies (since actually storing such a strategy can well take several hundred gigabytes of storage).

### 1.3 Question- and result-dependent costs

While in some games the player just has to minimize the number of questions, in other games some questions are more expensive than others. For example, let us say that in a variant of the number guessing game, in addition to the normal “Is it larger than  $x$ ?” questions for a cost of 1, a player could be allowed to also ask whether the last digit of the number is larger than the first, for a cost of 2. While this question is more expensive than normal questions, it also rules out roughly half of the possible numbers, so it could still pay off.

Besides different but fixed costs for questions, it is also possible and even common that the cost of a question depends on the answer. In most variants of BattleShips (described in detail in Section 2.7), for example, a player who hit the opponent’s ship is allowed to take another turn. Such a question is therefore practically free if the answer is “hit”, but costs one turn if the answer is “miss”. Similarly, in BlackBox (described in Section 2.10) the cost of a question depends on whether the light ray was absorbed or reflected, in which case the question costs 1, or just routed through the grid, in which case it costs 2.

In those games, instead of just minimizing the number of questions, their accumulated costs have to be minimized. In BattleShips, for example, it is not important to minimize the number of attacks, but to minimize the number of misses. An attack with a higher chance of hitting is therefore on average cheaper than an attack that will most probably miss.

### 1.4 Estimates

In some games, the goal is not to completely determine the hidden information, but only to find a reasonably good estimation of it and then take a guess. Depending on the accuracy of the estimate, a certain penalty is added to the final result. The most prominent example here is BlackBox, where the player in the end guesses the position of the hidden dots and gets a penalty of 5 points for each wrong guess, which is added to the total result.

A variant of the number guessing game might allow players to stop asking questions and take a guess, at the cost of the absolute difference between the guess and the correct answer. For example, a player could decide to first ask whether the number is larger than 50, and after that just guesses either 25 or 75, depending on the answer. If, for example, the correct number was 38, he would pay 1 for the question plus  $|38 - 25| = 13$  for the guess, so a total of 14. Obviously this strategy is not very good yet, but, for example, if the number is already known to be between 13 and 19, then it is indeed cheaper on average to just guess 16 (for an average penalty of  $\frac{3+2+1+0+1+2+3}{7} = \frac{12}{7} \approx 1.71$ ) instead of splitting the interval one more time and guessing afterwards (for an average penalty of  $\frac{7}{3}$  for the estimate in 3 out of 7 cases and an expected penalty of 1 in the other 4 cases, plus a fixed cost of 1 for the question, thus an expected total cost of 1.86).

<sup>2</sup>Let  $x_i$  denote the number of questions needed (using binary search) to determine the number within an interval that contains  $i$  numbers. Then we get  $x_1 = 0$  and

$$x_k = 1 + \frac{\lfloor \frac{k}{2} \rfloor \cdot x_{\lfloor \frac{k}{2} \rfloor} + \lceil \frac{k}{2} \rceil \cdot x_{\lceil \frac{k}{2} \rceil}}{k}.$$

(The weighted average of finding the number in an interval of length  $\lfloor \frac{k}{2} \rfloor$  in  $\lfloor \frac{k}{2} \rfloor$  cases and in an interval of length  $\lceil \frac{k}{2} \rceil$  in  $\lceil \frac{k}{2} \rceil$  cases, plus 1 for asking the question for the number in the middle of the interval.)

Solving this recursively leads to  $x_{100} = \frac{168}{25} = 6.72$ .

<sup>3</sup>Using the definition of  $x_i$  from the previous footnote, we get an average of  $1 + \frac{64 \cdot x_{64} + 36 \cdot x_{36}}{100} = 1 + \frac{64 \cdot 6 + 36 \cdot \frac{47}{9}}{100} = 6.72$  questions.

## 1.5 Formal problem definition

After this informal introduction, let us now define the setting in a more formal way.

**Definition 1.2.** Let  $\mathbb{N}$  denote the set of the non-negative integers including 0, that is,  $\mathbb{N} = \{z \in \mathbb{Z} \mid z \geq 0\}$ .

Let  $\mathbb{N}_\infty$  denote the set of the non-negative integers including 0 and infinity, that is,  $\mathbb{N}_\infty = \{z \in \mathbb{Z} \mid z \geq 0\} \cup \{\infty\} = \mathbb{N} \cup \{\infty\}$ .

**Definition 1.3** (deduction game). A DEDUCTION GAME consists of:

- a finite set of possible situations  $\mathcal{S}$ ;
- a finite set of available questions  $\mathcal{Q}$  with a corresponding finite set of possible answers  $\mathcal{A}$ ;
- a finite set of possible estimates  $\mathcal{E}$ ;
- a weight function  $\mathbf{w}: \mathcal{S} \rightarrow \mathbb{N}$  describing the weights that determine the probability with which each situation is chosen by the computer;
- an evaluation function  $\mathbf{r}: \mathcal{S} \times \mathcal{Q} \rightarrow \mathcal{A}$  that calculates the answer to a question  $q \in \mathcal{Q}$  in a situation  $s \in \mathcal{S}$ , and an evaluation function  $\mathbf{c}: \mathcal{Q} \times \mathcal{A} \rightarrow \mathbb{N}_\infty$  that gives the corresponding cost; and
- an evaluation function  $\mathbf{p}: \mathcal{S} \times \mathcal{E} \rightarrow \mathbb{N}_\infty$  that calculates the penalty for an estimate  $e \in \mathcal{E}$  when the actual situation is  $s \in \mathcal{S}$ .

For convenience, we furthermore define the abbreviation  $\mathbf{c}: \mathcal{S} \times \mathcal{Q} \rightarrow \mathbb{N}_\infty: \mathbf{c}(s, q) = \mathbf{c}(q, \mathbf{r}(s, q))$ . Which function  $\mathbf{c}$  we refer to will be obvious from the arguments.

Note that we are using integers for all weights, costs and penalties in order to avoid rounding errors. Theoretically, deduction games that require real numbers could exist, but for all games analyzed in this thesis, integers are sufficient.

We will use infinity to denote invalid choices in some games. We could, without changing the final result, also just use very very large integers for those moves: Let  $N$  be the largest total cost a player ever has to pay in a game in which he takes only valid (though possibly very dumb) moves, then you can set each invalid move to have cost, say,  $2N$ . Hence, any optimal strategy will not include those invalid moves (since by our assumption any strategy using only valid moves has a cost of *at most*  $N$ , and any strategy using at least one invalid move has a cost of *at least*  $2N$ ). However, explicitly setting them to infinity allows some optimizations later on, because we can safely ignore such branches (whereas  $N$  is usually not known, so also branches containing “invalid” moves would have to be considered). At the same time, as long as infinity follows “normal” calculation rules (i.e., infinity is larger than any other number, and infinity plus anything non-negative is still infinity), algorithms that do not give infinity special treatment will still work the same way as if we had just used a large number instead.

**Definition 1.4** (answer set).

- For each question  $q \in \mathcal{Q}$ , we are given  $\mathcal{A}_q \subseteq \mathcal{A}$ , a set of AVAILABLE ANSWERS that has to fulfil that  $\forall s \in \mathcal{S}: \mathbf{r}(s, q) \in \mathcal{A}_q$ , i.e., all answers that can occur for a question  $q$  are contained in  $\mathcal{A}_q$  (though  $\mathcal{A}_q$  may contain additional, superfluous elements as well).
- For each question  $q \in \mathcal{Q}$ , let  $\overline{\mathcal{A}}_q := \{a \in \mathcal{A} \mid \exists s \in \mathcal{S}: \mathbf{r}(s, q) = a\}$  be the set of POSSIBLE ANSWERS for a question.

Note that  $\overline{\mathcal{A}}_q \subseteq \mathcal{A}_q$ , and that  $\overline{\mathcal{A}}_q$  is unambiguous.

The point of having these two very similar definitions is that often it is easy to give a set  $\mathcal{A}_q$  of answers that can theoretically occur, but hard to calculate which ones actually occur. For example, in MasterMind (described in Section 2.6) with 4 positions it is easy to say that an answer contains between 0 and 4 black dots and between 0 and 4 white dots, which trivially gives us 25 theoretically available answers. Only 14 of those are actually used though, since the sum of black and white dots cannot be larger than 4, also the answer “3 black 1 white” cannot occur (and we can give an example for each of the remaining 14 answers in which it will be returned<sup>4</sup>). While in this example it is still relatively easy to calculate the set of possible answers manually, in games like BlackBox it is already somewhat tricky to prove that certain answers cannot occur.

<sup>4</sup>If ABCD is the correct code, then ABCD, ABCE, ABEE, ABDE, ABDC, AEEE, ACEE, ACDE, ACDB, EEEE, BEEE, BCEE, BCDE and BCDA produce those 14 answers.

When it comes to analyzing games, we can therefore either use  $\mathcal{A}_q$  directly, even though it contains a few superfluous answers, or we can calculate  $\overline{\mathcal{A}_q}$ , which takes some time at the beginning but possibly speeds up things later.

Finally, we need to define the cost for the player, which consists of the costs for the questions asked and the penalty for the final estimate:

**Definition 1.5** (cost). *Let  $s \in \mathcal{S}$  be the situation chosen by the computer,  $\hat{\mathcal{Q}} \subseteq \mathcal{Q}$  the set of questions that were asked by the player, and  $e \in \mathcal{E}$  the given estimate. The total cost  $C$  is then defined to be the sum of the costs of the individual questions and the penalty for the final estimate:*

$$C := \sum_{q \in \hat{\mathcal{Q}}} \mathbf{c}(s, q) + \mathbf{p}(s, e)$$

## Chapter 2

# Embedding popular games into this model (including game descriptions)

Let us now start to transform our games into this model. For each game, we will first describe the rules of the game, and then give the necessary definitions:

1. The set  $\mathcal{S}$  of situations and the weight function  $\mathfrak{w}$  that determines the probability with which they are chosen.
2. The set  $\mathcal{Q}$  of questions, the set  $\mathcal{A}$  of answers, as well as the answer function  $\mathfrak{r}$  and answer cost function  $\mathfrak{c}$ .

In some cases we additionally give  $\mathcal{A}_q$ , in others we will just assume  $\mathcal{A}_q = \mathcal{A}$  for all  $q$ .

3. The set  $\mathcal{E}$  of estimates and the function  $\mathfrak{p}$  for calculating penalties.

We will look at several problems that can be represented this way: some classical informatics questions (number guessing and variants thereof), some classical games (MasterMind, BattleShips, Minesweeper, BlackBox), some problems related to artificial intelligence (classification problems) and to industry (production parameters), a simple puzzle about scales and fake coins, and two tongue-in-cheek examples about life and milk that simply demonstrate the variety of problems that can be described this way.

## 2.1 Number guessing

Number guessing is a simple yet classic problem in computer science. Here we play it with the following rules: Let  $n$  be a fixed number. (Technically, it is a different game for each  $n$ .) The computer chooses a random integer from 1 to  $n$ , using a uniform random distribution (i.e., each number is equally likely). The player can ask questions of the form “Is the number larger than  $x$ ?”, where  $x$  can be any integer from 1 to  $n - 1$ . The player has to find out the exact number, using as few questions as possible.

This leads to the following:

1. The set  $\mathcal{S}$  contains the  $n$  situations corresponding to the numbers from 1 to  $n$ , each with a weight  $\mathfrak{w}$  of 1.
2. The set  $\mathcal{Q}$  contains the  $n - 1$  questions corresponding to the numbers from 1 to  $n - 1$ . The answer set  $\mathcal{A}$  contains exactly two answers, “true” and “false”. For the answer function, we get the following:

$$\mathfrak{r}(s, q) = \begin{cases} \text{“true”} & [\text{number corresponding to } s] > [\text{number corresponding to } q] \\ \text{“false”} & \text{otherwise} \end{cases}$$

Since we want to minimize the number of questions, all questions cost the same. The cost function is therefore constant, i.e.,  $\mathfrak{c}(q, a) = 1$  for all values of  $q$  and  $a$ .

3. The set  $\mathcal{E}$  contains the  $n$  estimates corresponding to the numbers from 1 to  $n$ . The player has to find out the correct number, so we set the cost of the correct estimate (i.e., the one that has the same number as was chosen by the computer) to be 0, and the cost of a wrong estimate to  $\infty$ . The penalty function therefore looks as follows:

$$p(s, e) = \begin{cases} 0 & [\text{number corresponding to } s] = [\text{number corresponding to } e] \\ \infty & \text{otherwise} \end{cases}$$

## 2.2 Number guessing with bit questions

As a slight variation of the previous game (and to have one simple example of a game with different types of questions), let us assume that we can in addition ask questions of the form “What is the value of the  $k$ -th bit of the number in binary representation?”. Each such question costs 3, i.e., as much as three normal questions. We obviously have  $\lfloor \log_2(n) \rfloor + 1$  such questions, where  $\log_2$  stands for the logarithm of base 2. All other parts of the game stay the same.

For simplicity, we call these two kinds of questions “normal questions” and “bit questions”. This leads to the following:

1. The set  $\mathcal{S}$  contains the  $n$  situations corresponding to the numbers from 1 to  $n$ , each with a weight  $w$  of 1.
2. The set  $\mathcal{Q}$  contains the  $n - 1$  normal questions corresponding to the numbers from 1 to  $n - 1$ , plus the  $\lfloor \log_2(n) \rfloor + 1$  bit questions. The answer set  $\mathcal{A}$  contains four answers, “true”, “false”, 1 and 0. For each normal question  $q$  we get  $\mathcal{A}_q = \{\text{true}, \text{false}\}$ , and for each bit question  $q'$  we get  $\mathcal{A}_{q'} = \{1, 0\}$ . (Note that of course we *could* have manually translated 1 to “true” and 0 to “false”. However, we want to give an example of a game in which we have different answer sets for different questions.)

For the answer function, if  $q$  is a normal question then we have the same answer function as before. If  $q$  is a bit question, then  $r(s, q)$  is the value of the bit in  $s$  that corresponds to  $q$ .

More mathematically (and less readable):

$$\begin{aligned} r'(s, q) &= \begin{cases} \text{“true”} & [\text{number corresponding to } s] > [\text{number corresponding to } q] \\ \text{“false”} & \text{otherwise} \end{cases} \\ r''(s, q) &= \text{Value of the bit at [bit position corresponding to } q] \text{ in [number corresponding to } s] \\ r(s, q) &= \begin{cases} r'(s, q) & q \text{ is a normal question} \\ r''(s, q) & q \text{ is a bit question} \end{cases} \end{aligned}$$

The cost function now depends on the type of question:

$$c(q, a) = \begin{cases} 1 & q \text{ is a normal question} \\ 3 & q \text{ is a bit question} \end{cases}$$

3. The set  $\mathcal{E}$  again contains the  $n$  estimates corresponding to the numbers from 1 to  $n$ . The penalty function is the same as before:

$$p(s, e) = \begin{cases} 0 & [\text{number corresponding to } s] = [\text{number corresponding to } e] \\ \infty & \text{otherwise} \end{cases}$$

## 2.3 Number guessing with fixed penalty

In another variant of the number guessing game, let us assume that we first can ask any number of questions. After that, we have to take a guess. If it is wrong, we pay a fixed penalty  $C$ .

The complete description of the game:

1. The set  $\mathcal{S}$  contains the  $n$  situations corresponding to the numbers from 1 to  $n$ , each with a weight  $\mathfrak{w}$  of 1.
2. The set  $\mathcal{Q}$  contains the same  $n - 1$  questions as in Section 2.1. Also  $\mathcal{A}$ ,  $\mathfrak{r}$  and  $\mathfrak{c}$  are identical to the original number guessing game from Section 2.1.
3. The set  $\mathcal{E}$  contains the  $n$  estimates corresponding to the numbers from 1 to  $n$ . The penalty function looks as follows for an integer constant  $C$ :

$$p(s, e) = \begin{cases} 0 & \text{[number corresponding to } s] = \text{[number corresponding to } e] \\ C & \text{otherwise} \end{cases}$$

## 2.4 Number guessing with estimates

In the next variant of the number guessing game, let us assume that we are again allowed to take a guess instead of having to find out the exact value. If the correct number is  $a$  and we guess  $b$ , then the penalty is  $C \cdot |a - b|^D$ , for some predefined positive integer constants  $C$  and  $D$ . For example, for  $C = D = 1$  we simply get  $|a - b|$  as penalty, the distance between the estimate and the correct number.

We can either use the standard questions as defined in Section 2.1, or optionally also allow the bit questions from Section 2.2.

The complete description of the game:

1. The set  $\mathcal{S}$  contains the  $n$  situations corresponding to the numbers from 1 to  $n$ , each with a weight  $\mathfrak{w}$  of 1.
2. The set  $\mathcal{Q}$  contains the same  $n - 1$  questions as before. Also  $\mathcal{A}$  and  $\mathfrak{r}$  and  $\mathfrak{c}$  are identical to the original number guessing game described in Section 2.1. Alternatively, we can also define  $\mathcal{Q}$ ,  $\mathcal{A}$ ,  $\mathfrak{r}$  and  $\mathfrak{c}$  like in the number guessing game with bit questions from Section 2.2.
3. The set  $\mathcal{E}$  contains the  $n$  estimates corresponding to the numbers from 1 to  $n$ . The penalty function looks as follows:

$$p(s, e) = C \cdot |([\text{number corresponding to } s] - [\text{number corresponding to } e])|^D$$

## 2.5 Number guessing with different question costs

In yet another variant of the number guessing game, let us assume that not all questions cost the same. Instead, asking “Is the number larger than  $x$ ?” costs  $g(x)$  for some function  $g$ . Simple examples for such functions are  $g(x) = x$ ,  $g(x) = x^k$  for some positive integer constant  $k$ , or  $g(x) = \lfloor \text{ld}(x) \rfloor$ . A more complex one could be  $g(x) = \#[\text{bits in } x \text{ that are 1 (in binary representation)}]$ .

The complete description of the game:

1. The set  $\mathcal{S}$  contains the  $n$  situations corresponding to the numbers from 1 to  $n$ , each with a weight  $\mathfrak{w}$  of 1.
2. The set  $\mathcal{Q}$  contains again the  $n - 1$  questions corresponding to the numbers from 1 to  $n - 1$ . The answer set  $\mathcal{A}$  and the answer function  $\mathfrak{r}$  are defined as before (in Section 2.1). For the cost function, we get

$$c(q, a) = g([\text{number corresponding to } q]) .$$

3. The set  $\mathcal{E}$  contains the  $n$  estimates corresponding to the numbers from 1 to  $n$ . The penalty function can be defined like either of the previously used penalty functions from Sections 2.1, 2.3, or 2.4.



## 2.6 MasterMind / Bulls and Cows

MasterMind is a popular code-breaking game invented by Mordecai Meierowitz in 1970 [4]. A code is defined as a sequence of  $n$  colour pegs (out of a selection of  $k$  colours) that has to adhere to certain rules. There are two commonly used variants of the rules for creating legal codes:

1. The more common variant does not impose any additional restrictions; we can for each position choose any of the  $k$  colours. Therefore, we get altogether  $n^k$  possible codes. [41]
2. The rarer variant is that each colour may appear only once in the code. We therefore have to choose  $n$  out of the  $k$  colours, and put them into some order. This leads to  $\binom{k}{n} \cdot n! = \frac{k!}{(k-n)!}$  possible codes. (Obviously, we have to have  $k \geq n$  for this to work.) [23]

The computer secretly chooses one such code. The description of legal codes above therefore defines our set of situations  $\mathcal{S}$ . Each code is chosen with the same probability, therefore  $\mathbf{w}$  is 1 for each situation. [4, 41, 23]

The player then “tries” one code in each move, and gets an answer that gives some information about the correctness of his code. There are two different variants for calculating this answer:

- A. For each position in which the hidden code and the player’s code match, the player gets one black dot. In addition, the player gets one white dot for each peg in his code that has the correct colour, but is in the wrong position.

More mathematically: First, the player gets one black dot for each peg that has the same colour as the computer’s peg in that position. After that, we ignore these positions, since they have already been scored. Of the remaining positions, we count for each colour how often it appears in the player’s code and how often in the computer’s code. Let us assume that of some colour there are  $p$  pegs in the player’s code and  $c$  in the computer’s code (not counting the already scored ones). If  $p \leq c$ , then we could move all  $p$  player pegs of that colour to positions in which they would yield black dots. Therefore, we get  $p$  white dots. If  $p > c$ , then only  $c$  of the pegs of that colour can be moved to spots where they would give black dots, so we get  $c$  white dots. Hence, we always get  $\min(p, c)$  white dots (separately for each remaining colour). [4, 41, 23]

- B. Again, the player gets one black dot for each position in which the hidden code and the player’s code match. In addition, the player gets one white dot for each remaining peg whose colour also appears in the code of the computer – regardless of how often it appears there.

An example:

**Example 2.1.** *Let us assume we have four colours A, B, C, and D. Let us assume the computer chose ABCD, and the player tried ACCB.*

*The first and the third position are the same in player and computer code, so in both variants, the player gets two black dots.*

*In variant A, we see that we could switch the second and fourth position to generate one additional black dot. Therefore, the player gets one white dot in addition.*

*In variant B, we see that the colours in the second and fourth position, C and B, both appear in the computer’s code. Therefore, the player gets two white dots.*

As we can see, the possible questions  $\mathcal{Q}$  are the same set as  $\mathcal{S}$ . The answer function  $\mathbf{r}$  is defined as described above (and depends on the variant of the game we look at).

The set of answers  $\mathcal{A}$  is by definition allowed to contain answers that do not even occur. We can therefore either define  $\mathcal{A}$  the lazy way as the set of all answers of the form “ $x$  black  $y$  white” with  $0 \leq x \leq n$  and  $0 \leq y \leq n$ , giving  $n^2$  possible answers. Or we can restrict  $\mathcal{A}$  to contain only answers of the form “ $x$  black  $y$  white” with  $0 \leq x + y \leq n$ , since each of the  $n$  pegs contributes at most one (black *or* white) dot, for a total of  $\frac{(n+1)(n+2)}{2}$  different answers. Or we can really calculate which of these answers actually occur, which is not entirely trivial. For example, with 4 pegs the answer “3 black 1 white” cannot occur, and with 4 pegs and 2 colours, also the answer “1 black 3 white” is not possible.

The player wins when he “tries” the correct code. Also for scoring we can look at two variants:

- a. The player’s score is the number of moves he needed before trying the correct solution. [4, 41, 23] We can therefore calculate the total cost as the number of questions *after which he needed an additional move*. The only answer after which he does not need an additional move is “ $n$  black 0 white”. (Otherwise, even if he knows for sure what the correct answer is, he still needs to take another move to end the game.) Thus, we define the cost function as

$$c(q, a) = \begin{cases} 1 & a \neq \text{“}n \text{ black 0 white”} \\ 0 & a = \text{“}n \text{ black 0 white”} \end{cases} .$$

(Note that in literature, usually the number of questions *including* the last question is optimized, so the results differ by an offset of exactly 1.)

- b. The cost of each question depends on the number of correct pegs; the more are correct, the lower the penalty for the player. The cost of a question is defined as

$$c(q, \text{“}x \text{ black } y \text{ white”}) = 2n - 2x - y .$$

Therefore, the cost is at most  $2n$  if all pegs are completely wrong, and it is 0 if and only if all pegs are correct.

We do not really need estimates here since the player is done after asking the question with the correct code. Since our definition requires estimates, though, we reuse  $\mathcal{S}$  as set of estimates, and define an estimate to be free if it is correct, and infinitely expensive otherwise:

$$p(s, e) = \begin{cases} 0 & s = e \\ \infty & s \neq e \end{cases}$$

That is, after having guessed the correct code and gotten “ $n$  black 0 white” as answer, the player simply officially announces “I guess that the code is [code just tried].”. If he would try that without being sure what the answer is, then the expected outcome for this would be infinitely expensive. Even if he was very lucky and guessed correctly, he could instead simply have asked that question at no cost (since the question for the correct code is free anyway).

Therefore, a (smart) player will only take an estimate if he knows for sure what the answer is, so introducing these estimates does not change the game.

As we can see, we altogether have 8 variants of the game. We will refer to them as 1Aa, 1Ab, 1Ba, 1Bb, 2Aa, 2Ab, 2Ba, and 2Bb.

The most commonly played variants are:

- Variant 1Aa with  $k = 4$  colours and  $n = 4$  pegs.
- Variant 1Aa with  $k = 6$  colours and  $n = 4$  pegs.
- Variant 2Aa with  $k = 10$  colours and  $n = 3$  pegs. In this variant, the game is also called “Bulls and Cows”.
- Variant 2Aa with  $k = 10$  colours and  $n = 4$  pegs. In this variant, the game is also called “Bulls and Cows”.

The B-variants are mostly a result of people having misunderstood the rules at some point and then insisting on playing it this way ever after. The b-variants are very rare, but are interesting to us because they have a large range of different costs for different answers.

## 2.7 BattleShips

BattleShips is a game first published by Milton Bradley Company in 1931, though the game idea is said to predate World War I [2]. Each player hides a predefined fleet of ships on a rectangular  $m \times n$  grid, adhering to certain rules. Usually these ships are rectangles of size  $1 \times 1$  to  $1 \times 5$ , and a fleet can contain ships of different sizes [53]. An example of a game situation can be seen in Figure 2.1.

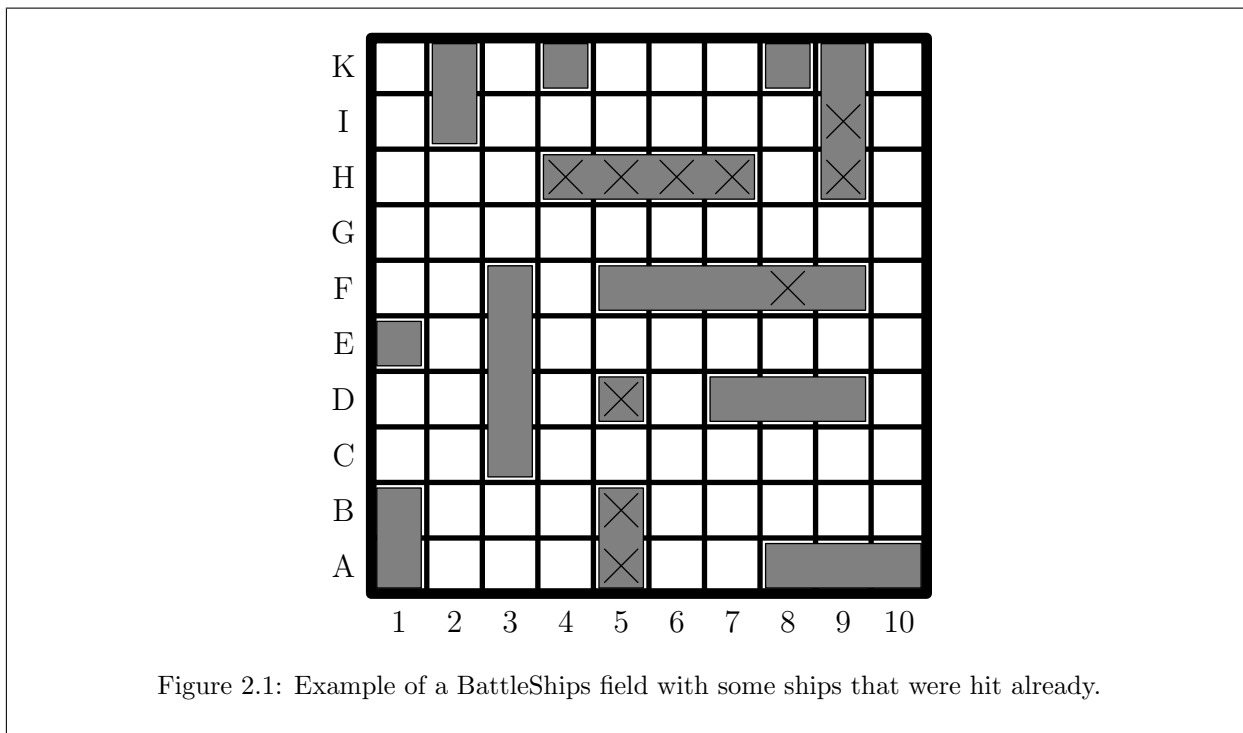


Figure 2.1: Example of a BattleShips field with some ships that were hit already.

Then the players alternately shoot at the ships of their opponent. For shooting, a player names one field, denoted by row and column. He is then told whether he hit part of a ship or just water. If he hit a ship, he is additionally told whether the ship is completely destroyed, though some variants do not give this additional information. (A ship is destroyed if each of its parts has been hit.)

If the player has hit a part of a ship, then he may shoot again. If he hits empty water, then it becomes the turn of the other player to shoot. The winner is the player who first destroys all ships of his opponent.

The three main variants for placing ships are:

1. Ships may not touch each other at all, that is, each field that is vertically, horizontally or diagonally adjacent to a ship must be water [53].
2. Ships may only touch each other at the corners, that is, each field that is vertically or horizontally adjacent to a ship must be water, but fields diagonally adjacent may contain other ships [9].
3. Ships may touch each other (but not overlap) [9, 2].

We can get an almost equivalent game as follows:

First, player A hides some ships according to these rules. Then player B starts shooting using the normal rules above. If he hits water, that is, if he would have to give the right to shoot to his opponent in the normal game, he has to write down a black dot. Then he continues shooting until he hits water again and has to write another dot, and so on. The game ends when B has destroyed all hidden ships, and his score is the number of black dots. (That is, the number of times he would have had to pass the right to shoot to the other player.)

Then, the two players reverse roles. Whoever collects less black dots, wins.

This is not entirely equivalent to the original game for the following reason: Let us assume that player A has the choice between three strategies. In the first strategy, which yields the best average result, he gets 20 dots on average, where in the worst case he gets 26 and in the best case he gets 18. In the second strategy, which has the lowest possible worst case, he gets 22 dots on average, 23 dots in the worst case and 20 dots

in the best case. In the third strategy, which is really bad both in terms of average and worst case, he gets 36 dots on average, 48 in the worst case and 4 in the best case. Now let us assume that player B has played the first part already and destroyed all ships while gathering only 15 dots. Then the third strategy, which is far from being the best strategy by itself, still is the only strategy with which player A can still win.

In short, if a player is far behind, he will be more willing to take risks. (This is a topic that could fill at least another thesis.) Therefore, playing the game in two parts, one after the other, is not equivalent to playing alternately.

However, it is still very similar to the original game, and thus interesting to find best strategies for.

The game we play therefore is defined as follows: The computer hides his fleet of ships on the grid according to the rules above. Then the player starts shooting at the ships, and has to take one penalty point every time he hits water. His score is the number of penalty points he takes until all ships are destroyed.

For the set of situations  $\mathcal{S}$  we use the possible constellations of ships, as defined in the rules above. The weight  $w$  is 1 for each constellation.

As far as the questions are concerned, we obviously have “shooting” as one type of question. That is, for each field  $(x, y)$  in the grid, there exists one question  $q(x, y)$ . For now, let us assume this question only answers whether there is water or a ship, not whether the ship was completely destroyed. Then the question is free if a ship was hit (since the player may move again), and costs 1 if there was water. In formulas:

$$\begin{aligned} \tau(s, q(x, y)) &= \begin{cases} \text{“water”} & s \text{ has water at position } (x, y) \\ \text{“ship”} & s \text{ has a part of a ship at position } (x, y) \end{cases} \\ c(q(x, y), a) &= \begin{cases} 1 & a = \text{“water”} \\ 0 & a = \text{“ship”} \end{cases} \end{aligned}$$

We call such a question a “shooting question”.

What we are still missing is the information whether the ship was completely destroyed. However, by the definition of a deduction game, the answer to a question depends only on the situation, not on the questions that have been asked before. That is, the computer does not keep any history, so it is absolutely impossible to answer whether this was the last part of the ship – simply because we have no knowledge at all whether the other parts of the ship have been hit already.

We therefore need to use dirty tricks.

We introduce new questions of the type “Is  $(x, y)$  to  $(x+i, y)$  a complete ship?” and “Is  $(x, y)$  to  $(x, y+i)$  a complete ship?”, for  $1 \leq i < [\text{maximum size of a ship}]$ . The player therefore can ask after each hit on a ship whether he already hit all parts of that ship.

As an example: Let us assume a player has hit a ship on  $(1, 4)$ ,  $(1, 5)$  and  $(1, 6)$ . Then he can ask “Is  $(1, 4)$  to  $(1, 6)$  a complete ship?”. The computer answers “no”. The player shoots at  $(1, 7)$  next (i.e., he asks the shooting question  $q(1, 7)$ ), and the computer answers “ship”. The player now asks “Is  $(1, 4)$  to  $(1, 7)$  a complete ship?”, and the computer answers “yes”.

This question can be answered by the computer, since it contains all the information about the relevant history as part of the question. Now we only need to discourage the player from asking this question without knowing for sure that the range really contains only ship parts. That is, if he has only shot at  $(1, 4)$  and  $(1, 6)$ , but does not know what  $(1, 5)$  contains, he is not allowed to ask this question.

We simply enforce this by making the answer infinitely expensive if this precondition is not met. Of course, this does not yet force the player to shoot at  $(1, 5)$  if he knows for sure that there is a ship part there. Let us assume that the player has already found and destroyed all ships except for one  $1 \times 3$  ship, and knows that  $(1, 4)$  and  $(1, 6)$  are the ends of this ship. Then he can ask “Is  $(1, 4)$  to  $(1, 6)$  a complete ship?” without having shot at  $(1, 5)$  before. However, shooting at  $(1, 5)$  is sure to be free in this case anyway, so not having done it does not change the score.

We therefore call such questions “sinking questions”, and denote “Is  $(x, y)$  to  $(x+i, y)$  a complete ship?” with  $\bar{q}(x, y, x+i, y)$  and “Is  $(x, y)$  to  $(x, y+i)$  a complete ship?” with  $\bar{q}(x, y, x, y+i)$ .

For writing the answer function in an easier way, let  $R(x_1, y_1, x_2, y_2)$  denote the rectangle containing the fields with  $x$ -coordinates from  $x_1$  to  $x_2$  and  $y$ -coordinates from  $y_1$  to  $y_2$ . The answer function for questions

of this type is then defined as follows:

$$\tau(s, q(x_1, y_1, x_2, y_2)) = \begin{cases} \text{“sinking”} & R(x_1, y_1, x_2, y_2) \text{ is a complete ship} \\ \text{“not yet sinking”} & R(x_1, y_1, x_2, y_2) \text{ is part of a ship} \\ \text{“player tries to cheat”} & R(x_1, y_1, x_2, y_2) \text{ contains water} \end{cases}$$

$$\mathfrak{c}(q(x_1, y_1, x_2, y_2), a) = \begin{cases} 0 & a = \text{“sinking”} \\ 0 & a = \text{“not yet sinking”} \\ \infty & a = \text{“player tries to cheat”} \end{cases}$$

$\mathcal{Q}$  therefore contains these two kinds of questions. Let the rectangle on which the ships are hidden have dimension  $m \times n$  and let  $k$  be the maximum length of a ship, then there are  $mn$  shooting questions and

$$\begin{aligned} X &= m \cdot (n + (n-1) + (n-2) + \dots + (n-k+1)) + n \cdot (m + (m-1) + (m-2) + \dots + (m-k+1)) \\ &= m \cdot (k \cdot n - 1 - 2 - \dots - (k-1)) + n \cdot (k \cdot m - 1 - 2 - \dots - (k-1)) \\ &= m \cdot \left( k \cdot n - \frac{(k-1)k}{2} \right) + n \cdot \left( k \cdot m - \frac{(k-1)k}{2} \right) \\ &= m \cdot n \cdot k - (m+n) \cdot \frac{(k-1)k}{2} \\ &\approx mnk \end{aligned}$$

sinking questions. Thus, there are almost  $k$  times as many sinking questions as shooting questions, and we will see that the number of questions significantly contributes to the calculation effort. On the other hand, in most situations the cost will be infinite for such a question. For efficiently calculating this game, it will therefore be crucial to find ways to avoid such questions.

Finally, as in the MasterMind game we need to define some estimates. We again use  $\mathcal{E} = \mathcal{S}$  and set

$$\mathfrak{p}(s, e) = \begin{cases} 0 & s = e \\ \infty & s \neq e \end{cases}.$$

## 2.8 Minesweeper

In Minesweeper,  $k$  mines are hidden in an  $m \times n$  grid (where often  $m = n$ ). The task of the player is to find the location of the mines without getting killed. In each step, he may look at one field. If the field contains a mine, the player loses. Otherwise he is told the number of mines on adjacent (horizontally, vertically or diagonally) fields [39, 51]. Some game situations can be seen in Figures 2.2, 2.3 and 2.4.

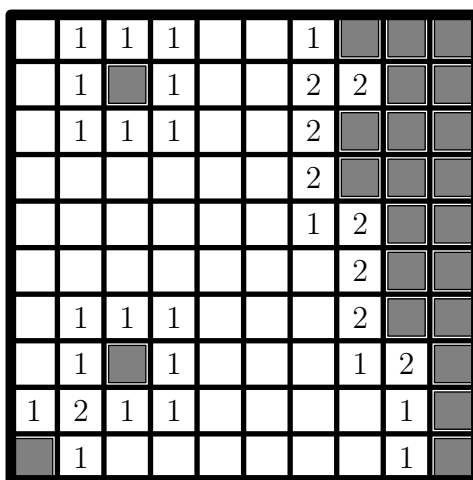


Figure 2.2: Example of a game situation in Minesweeper.

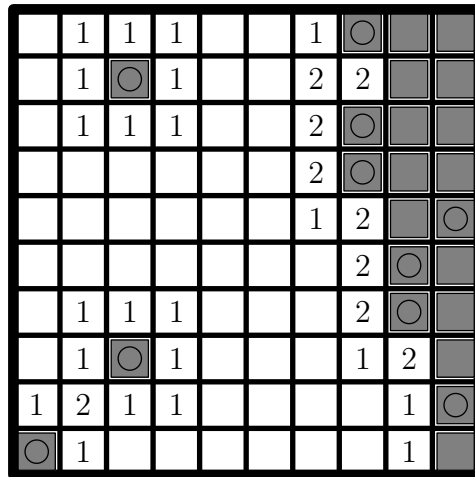


Figure 2.3: Example of a game situation in Minesweeper with location of the hidden mines.

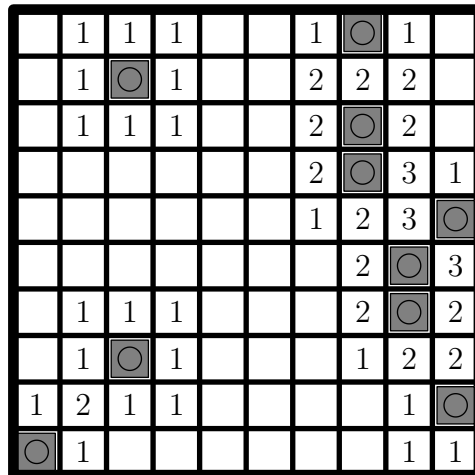


Figure 2.4: Completely solved Minesweeper game.

The task is to find the positions of the mines as quickly as possible. Most implementations of the game measure the time that a player needs, but since this is not possible in our framework, we will instead try to find a solution that accesses as few fields as possible (assuming that the number of clicks needed is closely linked to the time needed).

In some variants, looking at a field that has no mines on neighbouring fields automatically opens these fields.

The basic variant is fairly simple to describe:

1. The set  $\mathcal{S}$  contains all  $\binom{mn}{k}$  ways to distribute  $k$  mines onto the  $m \times n$  field. For all situations the weight  $\mathfrak{w}$  is 1.
2. The set  $\mathcal{Q}$  contains one question  $q(x, y)$  for each field  $(x, y)$ , so a total of  $mn$  questions. For the answer set we get  $\mathcal{A} = \{0, 1, 2, \dots, 8, \text{“mine!”}\}$ . The answer function is defined as follows:

$$\mathfrak{r}(s, q(x, y)) = \begin{cases} \text{“mine!”} & \text{field } (x, y) \text{ contains a mine} \\ \#[\text{mines adjacent to } (x, y)] & \text{otherwise} \end{cases}$$

In theory, the cost for hitting a mine is infinite. However, it is well known that there exist situations in Minesweeper in which risking to step onto a mine cannot be avoided [39, 51]. Therefore, also the average score would certainly be infinite, and optimizing it would not be very interesting.

Therefore, we define an optimal strategy as a strategy in which we have to take as little risk as possible; only among those we want to find the one in which we detect the mines fastest. That is, we would rather open every single safe space on the board than risking to step onto a mine. We can achieve this by setting the cost of a question to be 1 if we find a safe field and get the number of adjacent mines, and to be very high, for example  $\binom{mn}{k} \cdot mn$ , if we hit a mine:

$$c(q(x, y), a) = \begin{cases} \binom{mn}{k} \cdot mn & a = \text{“mine!”} \\ 1 & \text{otherwise} \end{cases}$$

If there is any risk that a field contains a mine, this means that in at least one of the  $\binom{mn}{k}$  situations there is a mine on this field. Therefore, if there is any risk, this risk is at least  $\frac{1}{\binom{mn}{k}}$ . The expected cost for opening such a field is therefore at least  $\frac{1}{\binom{mn}{k}} \cdot \binom{mn}{k} \cdot mn = mn$ . This is therefore already more expensive than opening all safe fields, of which there are at most  $mn - 1$ .

3. As in the previous games, the set  $\mathcal{E}$  equals  $\mathcal{S}$ , with the following penalty function:

$$p(s, e) = \begin{cases} 0 & s = e \\ \infty & s \neq e \end{cases}$$

## 2.9 Minesweeper with free empty fields

If we want empty fields (empty in the meaning of not being adjacent to any mines and thus returning 0 as answer) to open neighbouring fields for free, we can use a trick similar to the one used in BattleShips: We introduce a new type of question “What is on field  $(x, y)$ , which is a neighbour of the empty field  $(x', y')$ ?”. (We define one such question for each pair of neighbours  $(x, y)$  and  $(x', y')$ .) Let us denote such questions with  $q'(x, y, x', y')$ .

The answer function and cost function can be defined as follows:

$$\begin{aligned} r(s, q'(x, y, x', y')) &= \begin{cases} \#[\text{mines adjacent to } (x, y)] & r(s, q(x', y')) = 0 \\ \text{“player tries to cheat”} & \text{otherwise} \end{cases} \\ c(q'(x, y, x', y'), a) &= \begin{cases} 0 & a \in \{0, 1, 2, \dots, 5\} \\ \infty & a = \text{“player tries to cheat”} \end{cases} \end{aligned}$$

Note that  $\mathcal{A}_{q'} = \{0, 1, 2, 3, 4, 5\}$ , since 6, 7, 8 and “mine!” are not possible if there is a neighbour that returns 0.

This backdoor now allows the player to look at the neighbours of empty fields for free. The first empty field still has to be opened with a normal question and thus costs an action (just like in most implementations, the first empty field also still has to be clicked before the rest is opened automatically). Again, trying to cheat by using this function in other situations (without knowing a field to be empty) has expected costs of  $\infty$  and is therefore not worth even trying.

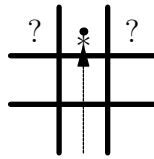
There is one small corner case: When a player knows a field  $(x, y)$  to be empty (returning 0), he now does not need to actually open it and can start looking at its neighbours immediately. Thus, he (wrongly) is not forced to pay the cost for the one click that starts the chain reaction for opening neighbours of empty fields. There are however rather few cases in Minesweeper in which we know a field to be free without having opened it. Therefore, and because it would be fairly hard to implement with our means anyway, we allow this little inaccuracy to exist.

## 2.10 BlackBox

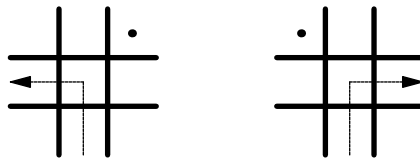
BlackBox is a game first published in 1977. The computer hides  $k$  dots on an  $n \times n$  grid, where in most variants  $k = 5$  and  $n = 8$ . Then the player can shoot light rays into the board. These are deflected by the dots according to certain rules until they leave the board again or are absorbed. The player is told where the light ray left the board (or whether it was absorbed). After shooting some light rays, the player has to take a guess about the location of the dots, and gets a certain penalty depending on the number of wrong positions in his guess [1, 8, 3].

Let us first describe the rules for deflecting light rays [1, 8, 3]. In the following graphs, “?” denotes that the content of a field is irrelevant, a dot denotes a dot, and an empty field denotes an empty field.

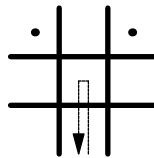
- If a light ray hits a dot, it is absorbed:



- If there is a dot diagonally ahead of the ray (and no dot straight ahead), it is deflected by  $90^\circ$  away from the dot:

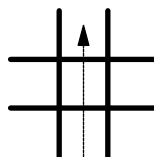


- If there are two dots diagonally ahead of the ray, one on each side (and no dot straight ahead), the ray is reflected:

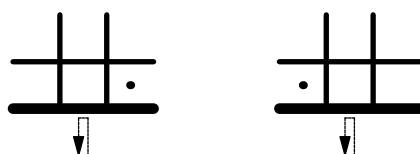


We can basically imagine that it is deflected with the normal rules twice shortly after another, first by the dot on the right, turning it counterclockwise to move towards the left, and then by the other dot (which is now diagonally right ahead of it) turning it back into the direction it came from.

- If there are no dots ahead of the ray, then it simply moves forward:



- Finally, we need one special rule when there already is a dot diagonally ahead of the ray before it even enters the field. The normal rules would deflect the ray, so it would then move outside the field. Therefore, we instead define the ray to be reflected in this case:





- Other than that, the rules when entering the board work as expected: If there is a dot ahead, the ray is absorbed, if there are dots diagonally ahead on both sides it is reflected by the normal rules, and if there are no dots ahead, it moves forward:

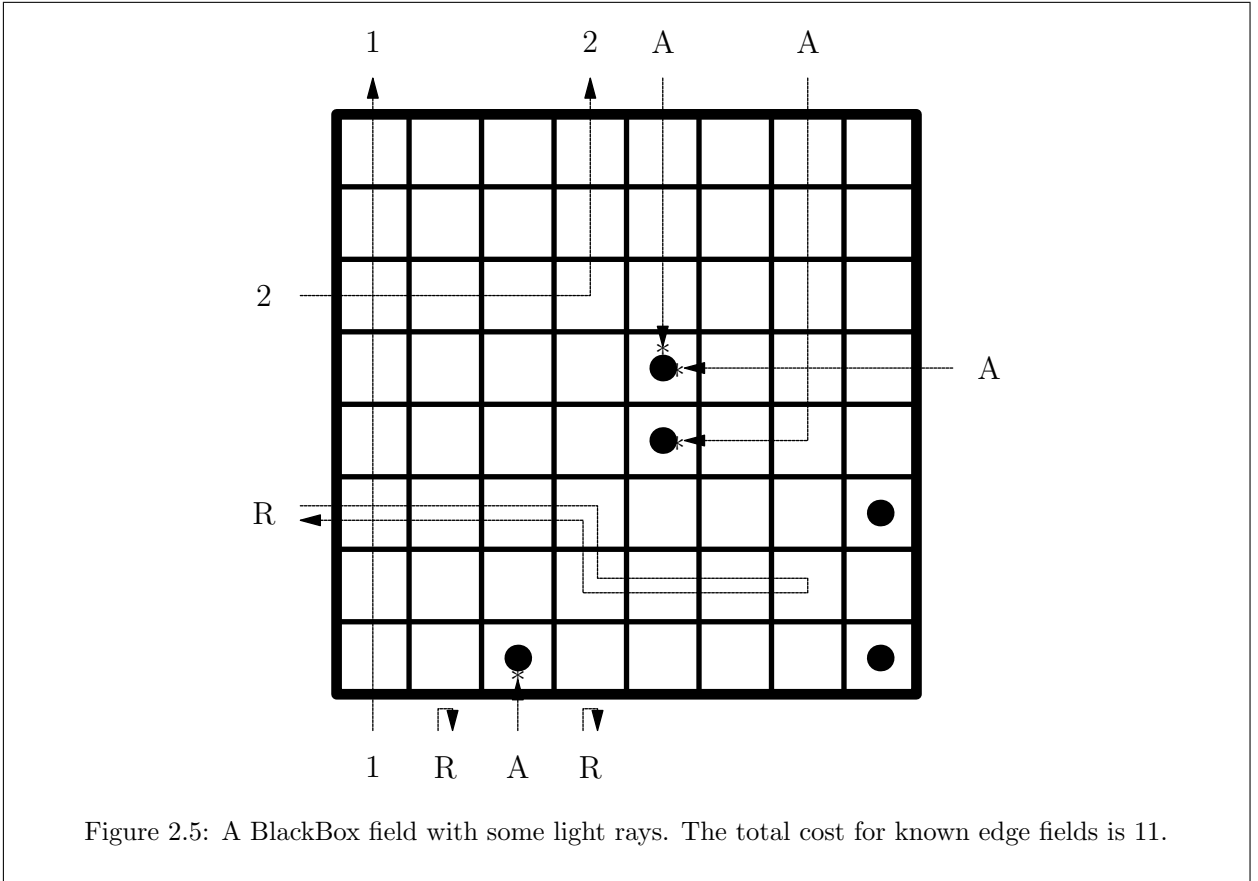
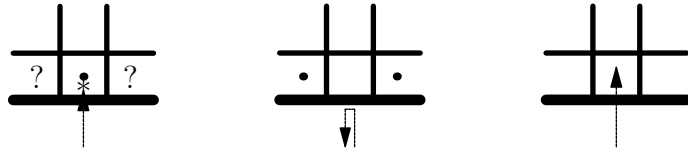


Figure 2.5: A BlackBox field with some light rays. The total cost for known edge fields is 11.

As we can see, light rays move back exactly the same way they came along. That is, if we shoot in a ray at position  $x$  and it leaves the grid at position  $y$ , then a light ray shot in at position  $y$  would leave the grid at position  $x$ . Therefore, light rays (except for those that get absorbed or reflected) connect two edge fields. We label corresponding edge fields with the same number. Edge fields from which the ray is absorbed we label with “A”, and edge fields where it is reflected to we label with “R”. (In most implementations, colours are used instead; for example, absorbed rays can be marked with grey, reflected rays with white, and for rays that leave the field in some other location, the two corresponding edge fields are marked in the same colour.)

Figures 2.5 and 2.6 show two examples of boards with some light rays and labeling of the known edge fields.

After some shots, the player takes a guess where the  $k$  dots are.

The total cost for the player is the number of known edge fields, plus a penalty of 5 for each dot guessed wrongly.

In our framework, we get the following:

1. The set  $\mathcal{S}$  contains the  $\binom{n^2}{k}$  ways to place  $k$  dots in the grid, each with a weight  $w$  of 1.



## 2.11 Classification

A classical problem from artificial intelligence: The computer chooses one element from a set, and we can ask yes-no-questions to find out which element was chosen. For example, the computer chooses an animal, and we can ask “Is it a mammal?”, “Is it larger than a horse?”, and so on. Or, the computer chooses a shape from a list, and we can ask “Is it convex?”, “Is it red?”, et cetera. The task is to find a good order for asking questions so that we can deduce the element chosen by the computer as quickly as possible [37].

These are simple examples, but artificial intelligence research provides ample supply of problems of this kind (except that there it usually is the *researcher* choosing a situation and the *computer* trying to deduce it with as few questions as possible).

Transformation into our framework is very straightforward:

1. The set  $\mathcal{S}$  contains the elements from which the computer can choose, either all with a weight  $\mathfrak{w}$  of 1, or using the weights from the original problem definition.
2. The set  $\mathcal{Q}$  contains the questions we can ask, and the answer set  $\mathcal{A}$  contains exactly two answers, “true” and “false”. The answer function  $\mathfrak{r}(s, q)$  is defined by the original problem.

The cost function is constant, i.e.,  $\mathfrak{c}(q, a) = 1$  for all values of  $q$  and  $a$ .

3. The set  $\mathcal{E}$  equals  $\mathcal{S}$ , and the penalty function looks as follows:

$$\mathfrak{p}(s, e) = \begin{cases} 0 & s = e \\ \infty & s \neq e \end{cases}$$

Note that this can easily be generalized to have any finite number of answers for each question.

The similarities are not surprising, given that deduction games are closely related to decision trees, which we will discuss in more detail in Sections 3.3 and 5.4.

## 2.12 Classification with different question costs

A generalization of the previous problem allows different cost functions for different questions. Let us look at just one practical example: A scientist is given a cube consisting of some material, and is asked to find out what the material is. There are various tests that she can perform to find the answer. She can look at it and tell that it is not cardboard. She can take it into her hand and guess from the weight that it is most likely a metal. She can apply electric current to it and measure the resistance. She can look at it under an electron microscope. She can try to put it on fire and test at what temperature it burns. (For simplicity, let us assume we have enough equivalent cubes of the material that we can destroy some of them in the process.)

Obviously, looking at it is a lot faster and cheaper than setting it on fire.

The formal game description:

1. The set  $\mathcal{S}$  contains the elements from which the computer can choose, either all with a weight  $\mathfrak{w}$  of 1, or using the weights from the original problem definition.
2. The set  $\mathcal{Q}$  contains the questions we can ask (or experiments we can perform), which have to have a finite number of possible outcomes. (In many classification problems, questions are defined in such a way that the only possible outcomes are “true” and “false”.) For each question  $q \in \mathcal{Q}$ , the set  $\mathcal{A}_q$  contains these (finitely many) possible outcomes for the question, and we define the answer set  $\mathcal{A}$  as the union of these sets  $\mathcal{A}_q$ . The answer function  $\mathfrak{r}(s, q)$  is defined by the original problem.

The cost function is constant for each question, that is, for each question  $q_i$  we have a constant  $c_i$ , and  $\mathfrak{c}(q_i, a) = c_i$  for all values of  $a$ .

3. The set  $\mathcal{E}$  equals  $\mathcal{S}$ , and the penalty function looks as follows:

$$\mathfrak{p}(s, e) = \begin{cases} 0 & s = e \\ \infty & s \neq e \end{cases}$$

## 2.13 Life

A slightly humorous example: We all go through life striving for happiness. However, there are many choices to be made that can influence our happiness: Many superficial acquaintances or few but close friends? Spend your evenings working or playing computer games? Use a car or public transport? Eat meat or be vegetarian? Believe in God or be atheist? Be honest or be polite? Shower in the morning or in the evening? Decisions, decisions.

Of course we can find the best configuration (which is a little different for everyone) by first trying one approach, and then trying what happens when leaving everything the same except for the answer to one of these questions. However, finding out how well an approach performs does take a while. Not eating meat for one day does not tell us yet whether being a vegetarian would make us happier, and going to church once does not yet make us religious.

Therefore, finding the best strategy can take a significant part of our life, so we also care about our happiness *during* the experimentation process.

We once again look at two different variants:

1. In variant one, we know how important each factor is. That is, we might know that the choice between being religious or being atheist can have a huge impact on our happiness, whereas the choice between showering in the morning or in the evening is rather unimportant. In other words, for each choice  $i$  we know the importance  $b_i$  of choosing correctly. If we choose correctly, nothing happens, but if we choose wrongly, we lose  $b_i$  happiness compared to how happy we could be. (In short, we assume we can be perfectly happy, and deduct points for not reaching this goal in some area.)
2. In the other variant, we know that each choice can either be very important, somewhat important or almost irrelevant. We do however not know which choice is how important. Let us assume that important choices contribute  $X$  (in some unit) to our happiness, somewhat important choices contribute  $Y$ , and almost irrelevant choices contribute  $Z$ , for any predefined positive integer constants  $X$ ,  $Y$  and  $Z$  with  $X > Y > Z$ . That is, we know that  $b_i \in \{X, Y, Z\}$  for all choices  $i$ , but do not know the exact values  $b_i$ .

For simplicity, we assume that the choices do not have any correlation.

Now let us define the game formally for  $n$  choices:

1. We assume that for each person there exists a better and a worse decision for each choice, and that each combination of such decisions is possible. In the first variant, the set  $\mathcal{S}$  therefore contains all possible configurations of correct decisions, for a total of  $2^n$  situations. In the second variant, we additionally need to take different importances into account. For each choice, we therefore have 6 possibilities: it might be better for the player to take the one or the other option, and the choice might be very important, somewhat important or almost irrelevant. Therefore, we get  $6^n$  situations.

As far as the weights are concerned, we might sometimes have *a priori* probabilities. For example, there are so many aphorisms that emphasize the importance of real friends, that we might already expect that for most people, few but close friends are a better choice than many superficial acquaintances. We can represent this by giving a higher weight to configurations in which “few close friends” is the better decision.

In the second variant, we might additionally have certain ideas about the importance of certain choices. For example, we might expect the choice between being religious or not to have a high influence on happiness for most people (even though there might be some who are almost equally happy being religious or being atheist), and we might expect the choice between showering in the morning or in the evening to be largely irrelevant to most people (even though to some people it might make a huge difference). Again, we can represent this by giving a higher weight to configurations in which the former is important and the latter irrelevant, meaning that such configurations will happen more often.

There might even be some form of correlation between those; for example, it could be that doing a lot of sports and showering in the evening is a frequent combination, and doing little sport and showering in the morning is quite likely as well, but the other two combinations are unlikely.

In short, the weights  $\mathfrak{w}$  can be pretty much anything.

2. The “questions”  $\mathcal{Q}$  are the experiments with some combination of decisions. There are  $2^n$  such combinations that the player can try. (This is the same for both variants of the game.)

Let  $b_i(s)$  be the importance of choice  $i$  in situation  $s$ , let  $c_i(s)$  be the correct decision for choice  $i$  in situation  $s$ , and let  $c_i(q)$  be the option chosen by the player for choice  $i$  in question  $q$ . (In the first variant, we can simply set  $b_i(s) = b_i$  for all  $i$  and  $s$  to use the same terminology.)

The answer function describing how happy or unhappy a configuration  $q \in \mathcal{Q}$  makes us can be expressed as follows:

$$\mathfrak{r}(s, q) = \sum_{i=1}^n \begin{cases} b_i(s) & c_i(s) \neq c_i(q) \\ 0 & c_i(s) = c_i(q) \end{cases}$$

As described above, experiments that make us happy are better (and thus cheaper) than those that make us unhappy during the duration of the experiment. Therefore,

$$\mathfrak{c}(s, q) = \mathfrak{r}(s, q) .$$

The interesting thing here is that the cost equals the answer.

3. Once we have done all experiments we want to make, we can decide on some combination of choices and live that way for the rest of our lives. Let us assume that the rest of our lives is approximately 50 years, so this choice adds 50 times as much to the total costs as another year of experimenting. Therefore,  $\mathcal{E} = \mathcal{Q}$ , and

$$\mathfrak{p}(s, e) = 50\mathfrak{c}(s, e) .$$

Alternatively, we can assume that modern medicine will give us a very very long life, so we are not content with anything but the best solution:

$$\mathfrak{p}(s, e) = \begin{cases} 0 & c_i(s) = c_i(q) \quad \forall i \in \{1, 2, \dots, n\} \\ \infty & \text{otherwise} \end{cases}$$

One proofreader pointed out that this is a very simplified view of life. If we wanted to increase the realism, we could replace the answer function  $\mathfrak{r}$  with basically any other function in the  $3n$  variables  $b_i(s)$ ,  $c_i(s)$ , and  $c_i(q)$ .

Ideally, choosing correctly in many areas should lead to lower costs than choosing wrongly, though it is absolutely possible to define the function in such a way that also a combination of completely wrong choices could give good scores. For example, the player might be very happy working hard and owning a car, even though spending time with family and friends and using public transport would be their ideal combination.

Also, this answer function could contain cross-effects between the choices. For example, the combination of doing lots of sports and showering only once per week might give very bad scores, even though the other combinations (no sports and showering once per week, no sports and showering daily, and lots of sports and showering daily) might all be rather neutral.

In short, the answer function  $\mathfrak{r}(s, q)$  can be defined completely arbitrarily. The player does however have to know its definition.

What is not possible in our framework is to have a correlation between consecutive experiments. For example, in real life, skipping all business meetings in one year might not go well with getting a promotion in the next. Since our framework does not allow keeping any history (as we have seen in BattleShips and Minesweeper already), we can however not simulate that.

## 2.14 Production parameters

A closely related but much more practical application from industry: Let us assume that we have a machine (factory, process, ...) for producing something, and several parameters we can tweak in the production process. For example, we can heat the material to 80°C or to 100°C, we can let the intermediate product rest for 2 hours or for 10 hours, we can use steel or aluminium, and so on.

Each of these decisions has some influence on the quality of the end product, and we want to optimize this quality. However, each decision also has some costs associated with it. In the final product, we do not care about these costs and simply want the product to be as good as possible. (Or we simply assume that the production costs are already accounted for in the quality measurement.) We do however want to minimize the costs of the experiments needed to determine the best settings for the parameters.

Let us assume that for each parameter, we have costs  $d_i$  and  $d'_i$  for the two options. (For example, if parameter 1 is the heating temperature, then heating to 80°C has cost  $d_1$ , and heating to 100°C has cost  $d'_1$ .) Like in the previous game, we can either know the values  $b_i$  that tell us how much influence each parameter has, or we know that  $b_i \in \{X, Y, Z\}$  for some constants  $X, Y$  and  $Z$ .

This results in the following formal game description:

1. We again assume that for each parameter there exists a better and a worse option, and that each combination is possible. Like before, the set  $\mathcal{S}$  contains all  $2^n$  possible configurations of best parameters in the first variant, and all  $6^n$  possible combinations of best parameters and importances in the second variant. Again, weights can be arbitrarily defined.
2. The questions  $\mathcal{Q}$  are again the combinations of parameter we can use for experiments. There are  $2^n$  such combinations that we can try (in both variants of the game).

Let  $b_i(s)$  be the importance of parameter  $i$  in situation  $s$ , let  $c_i(s)$  be the correct decision for parameter  $i$  in situation  $s$ , and let  $c_i(q)$  be the option chosen for parameter  $i$  in experiment  $q$ . (In the first variant, we again set  $b_i(s) = b_i$  for all  $i$  and  $s$ .)

The answer function describing the quality for an experiment  $q \in \mathcal{Q}$  can be expressed as follows:

$$r(s, q) = \sum_{i=1}^n \begin{cases} b_i(s) & c_i(s) \neq c_i(q) \\ 0 & c_i(s) = c_i(q) \end{cases}$$

The costs of a question  $q$  (i.e., of an experiment) only depend on the experiment parameters  $c_i(q)$  this time:

$$c(s, q) = \sum_{i=1}^n \begin{cases} d_i & c_i(q) = [\text{option 1 for parameter } i] \\ d'_i & c_i(q) = [\text{option 2 for parameter } i] \end{cases}$$

3. We are only happy with the perfect configuration. Therefore,  $\mathcal{E} = \mathcal{Q}$ , and

$$p(s, e) = \begin{cases} 0 & c_i(s) = c_i(q) \quad \forall i \in \{1, 2, \dots, n\} \\ \infty & \text{otherwise} \end{cases} .$$

Note that this can easily be generalized to have any finite number of options for each choice.

## 2.15 Scales

Just for the heck of it, we show that some classical riddles can be transformed into this format. The riddle we look at is the following one:

“We are given 13 coins, and know that exactly one of them is fake. The fake coin can be heavier or lighter than the other coins. Using a normal scale that compares the weights on its sides, how many measurements do we need to find the fake coin?”

While this does not look like a deduction game at first glance, we can transform it into our framework quite easily:

1. We have 13 choices of a fake coin, and we can make it lighter or heavier. This leads to 26 situations in  $\mathcal{S}$ .
2. The set  $\mathcal{Q}$  contains all possible ways to put some coins onto the left and some onto the right side of the scale. Since each coin can be on the left, on the right, or not on the scale at all, we get  $3^{13}$  possible questions. The answer set  $\mathcal{A}$  contains exactly three answers, “left is heavier”, “right is heavier”, and “the sides are equal”. The answer function  $\mathfrak{r}(s, q)$  is defined by comparing the weight on the left and right side. The cost  $\mathfrak{c}(q, a)$  is 1 for all values of  $q$  and  $a$ .
3. The set  $\mathcal{E}$  contains 13 estimates, one for each coin that we can suspect to be fake. The penalty function looks as follows:

$$\mathfrak{p}(s, e) = \begin{cases} 0 & [\text{fake coin in } e] = [\text{fake coin in } s] \\ \infty & \text{otherwise} \end{cases}$$

## 2.16 Milk

Last but not least, let us look at a tiny (and again somewhat humorous) example to demonstrate how the costs of a question can depend on the answer: Let us assume we have a milk carton of questionable age and state of decay in our fridge, and we want to find out whether it is still drinkable.

While this is not anything that we will ever implement on a computer, it does demonstrate a lot of the special properties that we allow deduction games to have (even though most of the games described so far feature at most one or two of these at the same time):

1.  $\mathcal{E}$ ,  $\mathcal{Q}$  and  $\mathcal{S}$  can be different sets.
2. Situations can occur with different probabilities (represented by weights  $\mathfrak{w}$ ).
3. It can contain questions with entirely different structures.
4. The cost of an answer depends not only on the question, but also on the given answer.
5. Not all answers in  $\mathcal{A}$  are possible for all questions in  $\mathcal{Q}$ .
6. The answer sets  $\mathcal{A}_q$  may be disjoint, but do not have to be.

Milk can exist in four states:

- It can be okay;
- It can be a little sour but still look and smell okay;
- It can be very sour and smell sour, but still look okay; or
- It can be completely mouldy (and smell sour).

We also have three ways of checking its state: we can look at it (which is a little hard because it is inside the carton, so we need to pry through the small opening), we can check if it smells sour, or we can taste it.

An overview of the possible outcomes:

	taste	smell	look
okay	✓	✓	✓
a little sour	×	✓	✓
very sour	×	×	✓
mouldy	×	×	×

What we want to find out is whether we should throw it away or still use it. (We want to still use it if and only if it is still okay, in case that was not obvious after this discussion.)

As we can see, the only question that can reliably answer this question is tasting the milk. Thus, if we only want to optimize the number of tests we have to make before being able to decide what to do, going directly for tasting is the best strategy.

Unfortunately, this optimization does not take all relevant factors into account yet, such as, for example, the not very desirable outcomes of accidentally tasting mouldy milk.

An updated overview of possible outcomes:

	taste	smell	look
okay	✓	✓	✓
a little sour	×	✓	✓
very sour	××	×	✓
mouldy	×××	×	×

A corresponding overview of the costs (measured in effort and outcome):

	taste	smell	look
okay	1	1	3
a little sour	3	1	3
very sour	6	2	3
mouldy	73827518	2	3

In addition to that, we might be able to remember how long the milk has been in the fridge and thereby guess how likely it is to be in each state.

Defined as a game, we get the following:

1. We have the possible situations  $\mathcal{S} = \{\text{"okay"}, \text{"slightly sour"}, \text{"very sour"}, \text{"mouldy"}\}$ . The weights depend on how long it has been in the fridge. Two examples:

- If it is only two days old, we might get the following weights:

$$\begin{aligned} \mathfrak{w}(\text{"okay"}) &= 20 \\ \mathfrak{w}(\text{"slightly sour"}) &= 15 \\ \mathfrak{w}(\text{"very sour"}) &= 5 \\ \mathfrak{w}(\text{"mouldy"}) &= 1 \end{aligned}$$

- If it was in the fridge for 2 months, the weights are more likely to look like this:

$$\begin{aligned} \mathfrak{w}(\text{"okay"}) &= 1 \\ \mathfrak{w}(\text{"slightly sour"}) &= 15 \\ \mathfrak{w}(\text{"very sour"}) &= 40 \\ \mathfrak{w}(\text{"mouldy"}) &= 60 \end{aligned}$$



2. As questions we get  $\mathcal{Q} = \{\text{“taste”}, \text{“smell”}, \text{“look”}\}$ . The answer function is defined as follows:

$$\begin{aligned} \tau(s, \text{“look”}) &= \begin{cases} \text{“okay”} & s \in \{\text{“okay”}, \text{“slightly sour”}, \text{“very sour”}\} \\ \text{“Yuck!”} & s = \text{“mouldy”} \end{cases} \\ \tau(s, \text{“smell”}) &= \begin{cases} \text{“smells okay”} & s \in \{\text{“okay”}, \text{“slightly sour”}\} \\ \text{“smells sour”} & s \in \{\text{“very sour”}, \text{“mouldy”}\} \end{cases} \\ \tau(s, \text{“taste”}) &= \begin{cases} \text{“okay”} & s = \text{“okay”} \\ \text{“slightly sour”} & s = \text{“slightly sour”} \\ \text{“very sour”} & s = \text{“very sour”} \\ \text{“Uaaghh!”} & s = \text{“mouldy”} \end{cases} \end{aligned}$$

Consequently, we get the set of answers  $\mathcal{A} = \{\text{“okay”}, \text{“Yuck!”}, \text{“smells okay”}, \text{“smells sour”}, \text{“slightly sour”}, \text{“very sour”}, \text{“Uaaghh!”}\}$ . For the individual questions, we get

$$\begin{aligned} \mathcal{A}_{\text{“look”}} &= \{\text{“okay”}, \text{“Yuck!”}\} \\ \mathcal{A}_{\text{“smell”}} &= \{\text{“smells okay”}, \text{“smells sour”}\} \\ \mathcal{A}_{\text{“taste”}} &= \{\text{“okay”}, \text{“slightly sour”}, \text{“very sour”}, \text{“Uaaghh!”}\} \end{aligned}$$

For the cost functions, we get:

$$\begin{aligned} \mathfrak{c}(\text{“look”}, a) &= 3 \\ \mathfrak{c}(\text{“smell”}, a) &= \begin{cases} 1 & a = \text{“smells okay”} \\ 2 & a = \text{“smells sour”} \end{cases} \\ \mathfrak{c}(\text{“taste”}, a) &= \begin{cases} 1 & a = \text{“okay”} \\ 3 & a = \text{“slightly sour”} \\ 6 & a = \text{“very sour”} \\ 73827518 & a = \text{“Uaaghh!”} \end{cases} \end{aligned}$$

3. The set  $\mathcal{E}$  contains “use” and “throw away”. The penalties  $\mathfrak{p}$  can, for example, be defined as follows:

	use	throw away
okay	0	20
slightly sour	100	0
very sour	500	0
mouldy	29357872749832	0

# Chapter 3

## Background

There exists very little prior work that deals directly with deduction games. However, individual games are well-studied, and the concept of deduction games is closely related to decision tree problems from the field of artificial intelligence research.

### 3.1 Research about deduction games in general

There are very few papers treating deduction games in general, and even those few that do so apply their findings only to MasterMind (described in Section 2.6) and the closely related Bulls and Cows (described in the same section).

In [20], a heuristic approach is shown that in each step estimates the usefulness of the next possible questions and only searches for optimal strategies among the questions that are ranked to be most useful. It also shows a probabilistic algorithm for proving lower bounds on the number of questions needed. It has been re-published by almost the same authors but in a slightly different form in [23] three years later. However, in spite of having “deductive games” in the title, even these papers only evaluate their algorithms on MasterMind and Bulls and Cows.

Other than that, the author of this thesis is not aware of any papers that study deduction games in general at all.

### 3.2 Known results about particular games

#### 3.2.1 MasterMind

MasterMind (described in Section 2.6) is the best-studied deduction game. A particularly famous result is a strategy for 4 positions and 6 colours by Donald E. Knuth in [41] that takes at most 5 moves (and shows that this is an optimum, i.e., there does not exist any strategy that takes at most 4 moves). This strategy needs 4.478 guesses on average.

In contrast to that, Kenji Koyama and Tony Lai show in [44] how to obtain the strategy that needs the least questions *on average*. This strategy takes 4.34 guesses on average, but in the worst case it needs 6 guesses.

A whole table of best possible averages and best possible worst cases for various different (small) numbers of positions and colours can be found in [30]. They furthermore discuss optimal strategies for MasterMind games with 2 positions and  $n$  colours.

Also in [21], a general approach to find optimal strategies for MasterMind games with 2 positions and  $n$  colours is presented. The authors claim that this approach can be applied to other deduction games as well, yet at the same time even admit that their “techniques are not easily extensible to  $m \times n$  MasterMind games and have not yet produced results for them” (where  $m \times n$  denotes a game with  $m$  positions and  $n$  colours).

Since all known ways to calculate optimal strategies require a lot of calculation effort to find them, they are not easily generalized to MasterMind games with more colours or more positions. Therefore, there exist a number of heuristic and greedy strategies; most commonly quoted are Irving [38] and Neuwirth [48], who

achieve averages of 4.369 and 4.364, respectively, for MasterMind with 4 positions and 6 colours. Others include [54] (studying a strategy in which simply any code that is still possible is chosen at random), [33] (studying heuristic strategies for 5 positions and 8 colours), [18] (using genetic algorithms), [22] (using a heuristic two-phase optimization algorithm that indeed finds the optimal strategy for MasterMind with 6 colours and 4 positions), and [29] (describing a strategy that is simple enough to be playable by a human with pen and paper). A very thorough discussion of various MasterMind strategies for 4 positions and 6 colours can be found in [43].

Besides strategies, there are also some more theoretical results. For example, in [24] an upper limit on the number of distinct questions necessary to differentiate between all questions is shown.

Finally, in [55] it is shown that given a set of questions and answers (for MasterMind with  $l$  positions), it is NP-complete (with regard to  $l$ ) to calculate whether there exists a situation that satisfies all these questions and answers.

### Variants and related games

For the closely related game Bulls and Cows (also described in Section 2.6), the already mentioned paper [23] shows that 7 guesses are necessary and sufficient in the worst case in a game with 4 positions and 10 colours.

Similarly, [36] shows an approach for optimal strategies in Bulls and Cows with 3 positions and  $n$  colours.

In [35], a variant of MasterMind is studied in which the computer is allowed to lie at most once during the game.

All that is quoted here is only a fraction of the research that has been done on MasterMind.

### 3.2.2 Minesweeper

Minesweeper (described in Section 2.8) is fairly well-studied as well.

Most research focuses on questions in which we are given one game situation, and have to determine whether a safe move exists, or whether a valid configuration of mines for this game situation exists.

The most famous result, published by Richard Kaye in [39] and later again in [40], is that it is NP-complete to determine whether a valid configuration of mines for any given game situation exists. He does so by showing that Minesweeper is in fact even Turing-complete.

Also [27], [47], and [51] study best strategies for given situations. The latter is a very thorough report that in addition discusses several heuristic strategies which try to minimize the total risk a player has to take. Note that [47] also provides ample pointers to other literature.

Just to name two example of artificial intelligence methods being used on Minesweeper, [50] uses multi-relational learning for training an artificial intelligence to play Minesweeper, and [58] applies Bayesian networks to it.

A generalized version of Minesweeper played on graphs rather than grids (considering a standard grid to be a graph in which grid fields are vertices and “neighbour” relationships are denoted by edges) is studied in [31], and shown to be solvable in linear time on trees.

Of course, calling Minesweeper a deduction game is a bit of a stretch, so there is hardly any traditional research on minimizing the number of clicks needed. However, there exist Minesweeper championships for human players (which, as a side note, are very fascinating to watch), and many of those players have written strategy guides. Most of these are published on homepages rather than as papers, such as for example [13], [12], [5] and [11]. Some go so far as to provide tips on mouse handling and reducing the distance between consecutive clicks. Sadly, none go far enough to actually analyze the expected performance of these strategies.

### 3.2.3 BattleShips

A BattleShips puzzle is defined as a game situation in BattleShips (described in Section 2.7) in which we know about some fields whether they contain water or part of a ship, and have to determine where the ships are. It is shown in [53] that also this problem is NP-complete.

Other than that, there seems to exist precious little material on BattleShips. There does exist the occasional informal strategy guide, either hidden between woefully much advertisement ([14]), or in form of answers in a maths forum ([7]). Neither of these strategy guides goes far beyond “shoot at fields on the diagonals first”, though in [7] you can find a suggestion for a modified binary search as well.

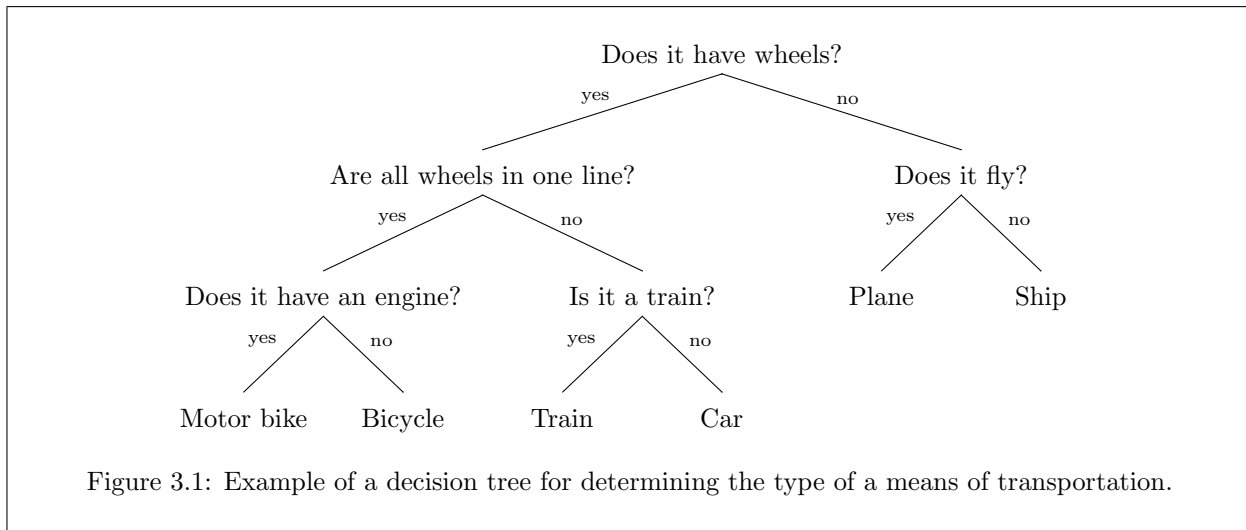
Also, there exist several computer implementations of BattleShips, most of which provide a computer enemy. A list of those can be found in [10]. Sadly, there seems to be no information whatsoever how these computer enemies are programmed (and chances are that many of them cheat by looking at the state of the player's field).

### 3.2.4 BlackBox

Finally, "BlackBox" is a somewhat unfortunate name for finding information about the game. However, to the thesis author's best knowledge, no research about BlackBox has been published to this date.

## 3.3 Decision tree learning

The field in artificial intelligence research that is very closely linked to deduction games is decision tree learning. Decision trees, also known as classification trees or regression trees, are trees that define an order for asking questions to determine the value of a variable [19]. A good introduction to decision tree learning can be found in [56]. A simple example of a decision tree is shown in Figure 3.1.



We are given a set of values that the variable we try to deduce can have (which is equivalent to our list of situations from which the computer can choose), a list of questions, and for each situation we know the answer to all questions. The task is to find a decision tree that uses these questions and has either minimum height or minimum external path length<sup>1</sup>.

The similarity to deduction games, in which we also want to find a good order of asking questions, is obvious. The differences are:

1. In deduction games, questions can have more than 2 different answers.
2. In deduction games, questions can have different costs.
3. In deduction games, questions can even have different costs for different answers.
4. In deduction games, it is allowed to take an estimate rather than having to determine the exact value of the variable.

While the first two differences are generalization that are mentioned in some papers on decision trees, the other two are specific to deduction games.

On the other hand, since this is a classical artificial intelligence problem, one central question is how well a decision tree created this way classifies values that were not in the original (training) set.

Nonetheless, the problems have a lot in common, so let us have a look at existing research.

Most interesting to us is the finding that constructing optimal binary decision trees is NP-complete, as shown in [37]. We will use this result in Section 5.4 to prove that also solving deduction games is NP-complete.

<sup>1</sup>The external path length of a full binary tree is defined as the sum of the path lengths from the root to each leaf [37].

---

Once it was shown that constructing optimal decision trees is NP-complete, there seems to have been little further interest in optimizing algorithms for it.

Instead, there is once again plenty of research on heuristic algorithms. The two most notable concepts here are Information Gain and Gini coefficients.

Information Gain, also known as Kullback–Leibler divergence, describes the change in information entropy before and after having asked a question. For example, if we have to guess an animal, then “Is it larger than a dog?” will give us a lot of extra knowledge, and therefore has a high information gain. Asking “Is it larger than a dinosaur?”, on the other hand, will barely tell us anything and therefore has low information gain. One way to construct trees therefore is to always choose the remaining question with the highest information gain. Closely related is the Information Gain Ratio, which favours questions with few outcomes over questions with many outcomes; a property that improves classification results for real-life classification problems. The exact mathematical definitions can be found in [45] (or, in a less scientific but much more readable form, in [46]), and in [32] it is shown how decision trees can be created using those. The most frequently used implementation is “C4.5” by Quinlan [52].

The Gini coefficient on the other hand measures the “unevenness” of a distribution. As such, it is frequently used to measure the unevenness of the wealth distribution in a society [26]. However, we can also use it to measure the unevenness of the distributions of answers to a question. For example, “Is the person male?” is almost equally likely to result in “yes” or “no”. On the other hand, “Does the person have red hair?” has a probability of 2% for “yes” and 98% for “no” (using recent population statistics). Therefore, the first question has a high Gini coefficient and the second a low one. We can then construct a tree by always choosing the question with the highest Gini coefficient. The mathematics can be found in [28] (or several papers citing this, since the original text from 1912 is not easy to get), and details about an implementation using Gini coefficients in [57]. This algorithm is occasionally referred to as “CART”, short for Classification And Regression Tree, although CART can also be used to describe the entire class of algorithms.

Both Information Gain and Gini coefficients, as well as their similarities, are explained very well in [56].

In contrast to those, there are also some heuristic approaches that do not locally optimize the next question for one particular node by using one of the heuristics described above, but rather try to globally optimize the resulting decision tree. One example of such an approach can be found in [17]. Another one is [25], which starts with a greedy solution and then gradually optimizes it, potentially in the background while the first draft of the decision tree can already be used.

Other approaches include hyperplanes [49], Naive-Bayes classifiers [42], and, once again, genetic algorithms [59].

## Chapter 4

# Reducing the number of distinct states by transformation into two equivalent games

### 4.1 Basic definitions from game theory

First of all, let us recapitulate some basic concepts from game theory. This entire section (until the beginning of Section 4.2, in which we will start to apply this to deduction games) mostly follows standard introductions to game theory and game trees as you can find them in hundreds of books and hear them in most game theory lectures at university. Just to name two such books, [16] and [34] both present the topic in this way.

Even though most of this chapter describes concepts that can also be found in these books, there are already some comments and definitions specific to deduction games interleaved in it. Therefore, all presented definitions that are standard usage in the field of game theory are marked as quoted from these books, whereas definitions that do not mention them are specific to this thesis.

#### 4.1.1 Properties of deduction games

We start with two definitions about the properties of games:

**Definition 4.1** (perfect information). *A game is said to have PERFECT INFORMATION if (and only if) all information is known to all players at all times. [16, 34]*

Chess is a classical example of a game with perfect information, as are many other abstract games.

In particular, a game does not have perfect information if it features hidden hand cards or hidden moves.

Deduction games per definition do *not* have perfect information, since the chosen situation that the player has to deduct is only known to the computer.

**Definition 4.2** (complete information). *A game is said to have COMPLETE INFORMATION if (and only if) the rules and variables of the game (and consequently the possible strategies and the resulting payoffs of all players) are known to all players at all times. (Note that the moves that the other players made do not have to be known to each player.) [16, 34]*

In particular, games with random elements (such as dice rolls or shuffled draw piles) do not have complete information. As a rule of thumb, games have complete information if we can exactly determine the outcome only from knowing the actions of the players. In chess, for example, the exact sequence of moves of the players is sufficient to completely reconstruct the game situation after those moves, whereas in Settlers of Catan, watching only the moves of the players (counting building on a given location, buying development cards, and trading as moves), we could neither tell for sure what the board looks like, nor what resources players got; both the random playing board and the outcomes of the dice rolls are outside the control of the players.

Therefore, deduction games do *not* have complete information because of the randomly chosen situation that the player has to deduct. They fulfill all other criteria for a game with complete information though, since the set of possible situations and the set of possible answers for each question is known.

Note that *choosing* one situation has to be done according to the given probability distribution  $\mathfrak{w}$ , so it is a random element rather than a *move* by the computer. That it is done randomly prevents deduction games from having complete information, whereas that the outcome of this random event is only known to the computer prevents it from having perfect information.

### 4.1.2 Game trees

In game theory, game trees are used to represent a game. We will first define a game tree for a game with complete and perfect information and will then add methods to deal with the additional aspects of deduction games.

**Definition 4.3** (game tree (for games with complete and perfect information)). *In a game with perfect and complete information, the game tree is a rooted tree with the following properties:*

- *Each node represents one game situation, i.e., the positions of all pawns, tokens, cards, pieces and so on, as well as whose turn it is to move.*
- *The root represents the starting position of the game.*
- *A (downward) edge between two nodes represents that the game situation in the upper node can be transformed into that in the lower node by a valid game move of the player on turn. From each node there is one downward edge for each valid move.*
- *Each leaf represents a possible ending of the game.*
- *For each leaf we are given the payoffs for all involved players.*

[16, 34]

(Note that in some literature this is also known as an EXTENSIVE-FORM GAME, whereas a GAME TREE is then defined to be only a subtree of it. [16])

Every played game can be represented as one path from the root to one leaf.

Note that a game tree (in this definition) does not necessarily have to be finite. If a game continues infinitely (for example, because it starts to “loop”), the game tree becomes infinitely deep. If a player has an infinite choice of options in some situation (for example, by selecting a real number from an interval), the tree becomes infinitely wide. Fortunately, neither of this can happen in deduction games (as long as you play them smart enough, i.e., forbid the player to ask the same question twice).

**Example 4.4** (chess). *In chess, each node represents one possible setup of the chess pieces plus the information whose turn it is. This is exactly the information we are given in an average chess puzzle of the “How does white win here in 5 moves?” kind. In addition, the position of the node within the tree, i.e., the unique path from the root to the node, contains the information about the moves that led to this situation. From this we can draw information like whether castling is still possible, and how often each position has been repeated already.*

*The root is the starting position. In each node, there is one outgoing edge for every possible game move.*

*Leafs are those positions where either one player is checkmate, or there is a draw (which can be caused by stalemate, fifty-move rule or threefold repetition). The possible payoffs in chess are 1/0, meaning the white player wins the game and gets 1 point (while black loses and gets no point), 0/1 in a situation where black wins, and 0.5/0.5 in case of a draw.*

**Definition 4.5** (subgame). *A SUBGAME is the game that results when taking one node and all its successors from the original game tree and defining the root of that subtree as the new root of the game. [16, 34]*

A subgame thus is the game that results when starting from one situation in the middle of the original game.

We will introduce one more definition for simplicity:

**Definition 4.6.** *For any node  $N$  in a rooted tree, let  $\text{parent}(N)$  denote the parent node of  $N$ , and let  $\text{children}(N)$  denote the set of the children of  $N$ .*

### 4.1.3 Solving game trees

**Definition 4.7** (strategy). *A STRATEGY is the complete plan of the decisions (moves) that a player will make for all situations that might arise. [16, 34]*

If the complete game tree is known, it is simple (but not necessarily easy, due to the large size of such game trees) to calculate the best move for every player in every situation under the assumption that all other players will play perfectly as well (and resolve ties between equally good moves in a predefined way).

This is because once the payoffs of all children of a node are known, the player's best move is to choose the move (and thus child) that will give him the highest payoff. By doing this recursively starting from the leaves and going all up to the top, the payoffs and best moves in all nodes can be calculated. This is also known as a SUBGAME PERFECT EQUILIBRIUM [16, 34].

Assuming that all players play perfectly, the game will always follow the same path from the root to some node. (In case that two moves are equally good in any situation, we can decide that the player will always take the leftmost of those moves.) Often we are only interested in this one path, since it already tells us everything about the outcome of the game and the payoffs for the players if all players play perfectly, as well as how the players should move in all situations that will (in perfect play) occur. This leads to the following definition:

**Definition 4.8** (perfect play, perfect path). *A PERFECT PLAY is the course (and outcome) of a game in which all players always take the best possible move. The corresponding path from the root to a leaf is called the PERFECT PATH. [16, 34]*

As mentioned before, game trees are often rather immense, which usually makes calculating all possible paths that a game can take virtually impossible. When it comes to actually calculating the outcome in a perfect play, it is therefore crucial to stick to the perfect path as closely as possible and avoid calculating unnecessary branches.

However, if some players do not play perfectly, this might change the best strategies for the other players. (For example, if one player always makes the same stupid mistake in some situation, it is an advantage for his opponent to create this situation, even though against a perfectly playing opponent it would be a disadvantage.) We therefore cannot define an optimal strategy for a single player in a multi-player game. For games with only one player (such as deduction games) an optimal strategy can be defined though:

**Definition 4.9** (optimal strategy). *An OPTIMAL STRATEGY in a one-player game with perfect and complete information is a strategy with a higher (or at least equal) payoff than any other strategy.*

Since deduction games have neither perfect nor complete information, we need two additional concepts:

### 4.1.4 Dealing with randomness

In order to represent random events, we introduce an additional player (often called "Chance", "Randomness" or "Nature"). For nodes in which Chance moves, we are additionally given probabilities for all outgoing edges. This "player" chooses one of the possible moves, adhering to the given probability distribution.

While without randomness the best move is simply the one with the highest payoff, we now need to think about how we define an optimal strategy – do we, for example, want to maximize the average outcome, or do we want to minimize the risk of a bad outcome? Do we prefer a move with a high average, but a certain chance of a really bad outcome, or do we prefer one with a slightly lower average, but no risk of a bad outcome? In short: Do we prefer playing Russian Roulette, with a high probability of no negative consequences, but a small chance of immediate death, or do we prefer jumping from the first floor, with almost certainty of a few broken bones, but also almost certainty of nothing worse than that?

In this thesis, we will look at two kinds of optimal strategies:

**Definition 4.10** (average-optimal strategy). *We call a strategy in a single-player game AVERAGE-OPTIMAL if its outcome is on average higher than (or at least equal to) any other strategy.*

**Definition 4.11** (worst-case-optimal strategy). *We call a strategy in a single-player game WORST-CASE-OPTIMAL if the worst possible outcome is better than (or at least equal to) the worst possible outcome of any other strategy.*



In case of MasterMind for example, Knuth found a worst-case-optimal strategy that needs at most 5 guesses to deduct any situation, and needs 4.478 guesses on average [41]. Kenji and Lai in contrast calculated an average-optimal strategy that on average needs only 4.34 guesses, but in the worst case needs 6 guesses [44].

For solving game trees, this changes various things. Whenever we get to a node where Chance plays, we now need to calculate the payoffs of *all* branches that can possibly be chosen by Chance, and then have to either average their payoffs (for finding average-optimal strategies) or find their maximum (for finding worst-case-optimal strategies). Depending on how often Chance moves, this can drastically increase calculating effort: while before the bare minimum of necessary calculation was one path from the root to a leaf (the perfect path), it is now an entire subtree.

#### 4.1.5 Dealing with imperfect information

The whole concept of deduction games revolves around imperfect information. If there were perfect information, i.e., all players knew at all times all the information, there would not be any point in *deducting* that information. Deduction games are about hidden information, so we need a way to represent this in a game tree.

**Definition 4.12** (information set). *An information set is a subset of nodes of a game tree such that*

- *it is the same player's turn in all nodes and*
- *to the player who has to move, these nodes are not distinguishable.*

*In short, all information that differentiates the nodes of an information set from each other is hidden from the player whose turn it is.*

*This also means that the same moves have to be available in all nodes of an information set, since a move's availability or lack thereof makes situations distinguishable. [16, 34]*

Usually all nodes in an information set have to have the same distance from the root, since in most cases the number of moves since the beginning of the game is known to all players.

Since a player can not distinguish between nodes in an information set, he has to take the same decision in all nodes of an information set. When calculating optimal strategies, instead of giving the best move in every node we therefore now need to give the best move in every information set.

Similarly, the expected cost (both average and worst-case) is the same for all nodes in an information set (logically because the player does not know which node of the information set he is in; mathematically because the costs for questions that were already asked are the same, and the expected future costs are the average or maximum of all nodes in the information set).

**Definition 4.13.** *For any node  $N$ , we use  $\langle N \rangle$  to denote the set of all nodes that are in the same information set as  $N$ .*

#### 4.1.6 Types of solutions

What we are looking for now are two kinds of solutions:

**Definition 4.14** (solutions).

- *A STRATEGY SOLUTION describes a sequence of moves that minimizes the (average or worst-case) cost. Alternatively we can say that a strategy solution is a strategy that chooses the best move in all situations that will occur in perfect play.*
- *A COST SOLUTION only gives the (average or worst-case) cost that will occur when playing such a strategy, without describing the strategy itself.*

Altogether we therefore try to calculate four different things, for which different optimizations might be useful:

- average-optimal strategies;
- expected cost when playing an average-optimal strategy;
- worst-case-optimal strategies; and
- worst possible cost when playing a worst-case-optimal strategy.

## 4.2 Game tree of a deduction game

Now that we have all necessary definitions for game trees, it is easy to show what the game tree of a deduction game looks like (see Figure 4.1). There is only one player, and in addition we have the computer which both represents Chance and answers questions.

Each node contains the information about the situation chosen by the computer, the questions that have been asked so far and the answers that were given.

In the root node, nothing is determined yet, and Chance has to move. Chance chooses one of the possible situations  $\mathcal{S}$  (as defined in Section 1.5) adhering to the given weight function  $\mathbf{w}$ .

After that, Chance is out of the game, and in all further nodes either the player moves or the computer answers. In each player node there is one outgoing edge for each question  $q \in \mathcal{Q}$  that has not been asked before (on the path from the root to the node). In addition, we have one outgoing edge for “taking a guess” that will lead to a leaf. In every computer node there is only one outgoing edge for giving the correct answer. In every following layer the player can ask one question less than before (since he already has asked it), therefore in the last layer (after having asked all possible questions), taking a guess is the only remaining option.

We can therefore easily calculate how many nodes such a tree has:

- On the highest level, we only have the root in which Chance moves.
- The root has one outgoing edge for every situation, so there are  $|\mathcal{S}|$  player nodes on the second level.
- Each of these nodes has one outgoing edge for every question, plus the edge for taking a guess, so  $|\mathcal{Q}| + 1$  edges.
- On the next level, we have two different kinds of nodes. Having taken a guess leads to a leaf, so there are no more outgoing edges there. In the other nodes the computer answers, so there is exactly one outgoing edge.
- In the next round the player can still ask all questions except for the one that has been asked already (which is different for every node), or take a guess, which gives us a total of  $|\mathcal{Q}|$  outgoing edges.
- Similarly, on the next level we have some estimate leafs and some nodes in which the computer answers. On the level after that, we have player nodes with  $|\mathcal{Q}| - 1$  outgoing edges, then follows a level with leafs and computer nodes, then player nodes with  $|\mathcal{Q}| - 2$  edges, and so on.
- ...
- On level  $2|\mathcal{Q}|$ , we have player nodes with the choice between one last question that can be asked or taking a guess.
- On level  $2|\mathcal{Q}| + 1$ , we have leafs and computer nodes.
- On level  $2|\mathcal{Q}| + 2$ , we have player nodes in which all questions have been asked already, so the only remaining choice is taking a guess.
- On level  $2|\mathcal{Q}| + 3$  finally, we have only leafs from the guesses in the previous layer.

We see that there are four kinds of nodes:

- player nodes;
- computer nodes;
- estimate leafs; and
- one Chance node.

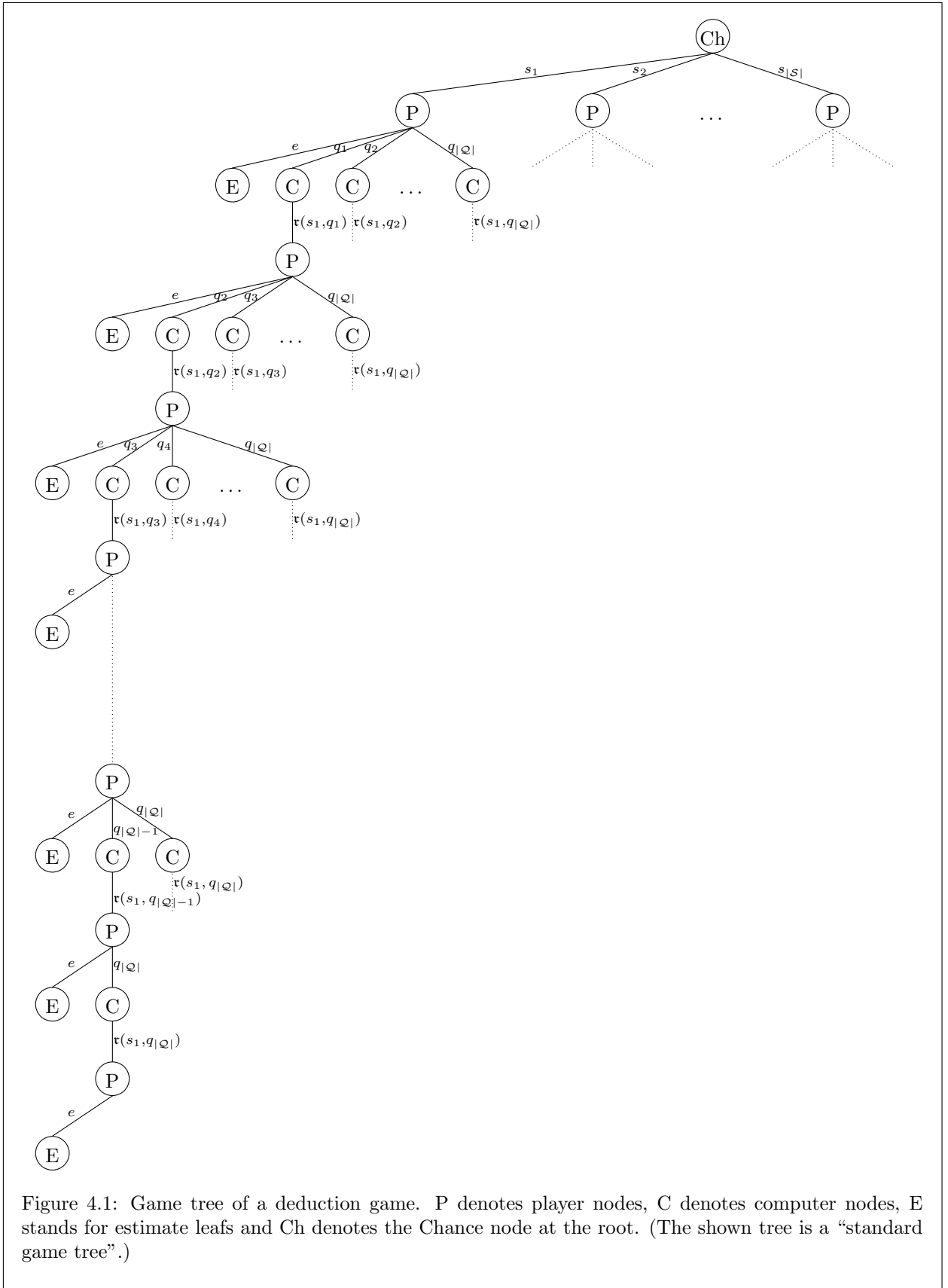


Figure 4.1: Game tree of a deduction game. P denotes player nodes, C denotes computer nodes, E stands for estimate leafs and Ch denotes the Chance node at the root. (The shown tree is a “standard game tree”.)

Let us count those nodes independently, starting with the player nodes. Let  $s = |\mathcal{S}|$  and  $q = |\mathcal{Q}|$ . There are  $s$  player nodes on the second level,  $s \cdot q$  on the fourth,  $s \cdot q(q - 1)$  on the sixth,  $s \cdot q(q - 1)(q - 2)$  on the eighth and so on, until we have  $s \cdot q(q - 1)(q - 2) \cdots 3 \cdot 2$  nodes on level  $2q$  and  $s \cdot q(q - 1)(q - 2) \cdots 3 \cdot 2 \cdot 1$  nodes on level  $2q + 2$ . Altogether, there are therefore

$$\begin{aligned} p &= s \cdot (1 + q + q(q - 1) + q(q - 1)(q - 2) + q(q - 1)(q - 2) \cdots 3 \cdot 2 + q(q - 1)(q - 2) \cdots 3 \cdot 2 \cdot 1) \\ &= s \cdot \left( \frac{q!}{q!} + \frac{q!}{(q - 1)!} + \frac{q!}{(q - 2)!} + \frac{q!}{(q - 3)!} + \cdots + \frac{q!}{1!} + \frac{q!}{0!} \right) \\ &= s \cdot \sum_{n=0}^q \frac{q!}{n!} \end{aligned}$$

player nodes.

Each player node leads to exactly one estimate leaf, and there are no other estimate leaves. Therefore there are exactly the same number of player nodes and estimate leaves, i.e., there are  $p$  estimate nodes.

Every player node (except for those on the second level) has a computer node as a parent, and every computer node is parent to exactly one player node. The number of computer nodes therefore equals the number of player nodes minus the number of player nodes on the second level, so there are  $p - s$  computer nodes.

In addition we have one Chance node.

In total, there are

- $p$  player nodes,
- $p - s$  computer nodes,
- $p$  estimate leaves and
- 1 chance node,

so a total of

$$3p - s + 1 = 3s \cdot \sum_{n=0}^q \frac{q!}{n!} - s + 1$$

nodes.

For a simple game of MasterMind with 4 positions and 4 possible colours (and thus 256 situations and the same number of questions) that are no less than 1790815081607894645094669385755502533175174838908724371417851621712261483310232872207753968182164851082919824660593450210385465175744545467894107908747398491433399863372542204211049142555921992999223968647008415078672299180749199671457663464386108945149487177350014018409982809208544574686220593881598935897180667951752699668868964232771897203688332551371273512959977610060876479227348791506410235280916770846524061053188632760679650450540680932228653617926058027664564399138916398391730840736530963339819614721 nodes ( $\approx 1.8 \cdot 10^{510}$ ).

We can approximate  $\sum_{n=0}^k \frac{1}{n!}$  with  $\Theta(1)$  (because  $1 \leq \sum_{n=0}^k \frac{1}{n!} \leq \sum_{n=0}^{\infty} \frac{1}{n!} = e$ ), so the number of nodes is  $\Theta(s \cdot q!)$ . Consequently, the number of situations is harmless compared to the number of questions. This explains why the game tree of BlackBox is comparatively small in spite of having a large number of possible situations (7624512 to be exact) – having only 32 possible questions, the game tree has “only” 16360611650782809934502340636359609793564289  $\approx 1.6 \cdot 10^{43}$  nodes (which still takes a few billion years to calculate even on a modern super computer [15]).

### 4.2.1 Information sets

For the player, all nodes in the second layer are in one information set, since he does not know which situation the computer has chosen and does not have any other information. On the following layers, the information sets consist of player nodes for which

- the same questions were asked in the same order and
- the same answers were given

(see Figure 4.2).

In addition, we have information sets among the estimate nodes that follow the same rules.

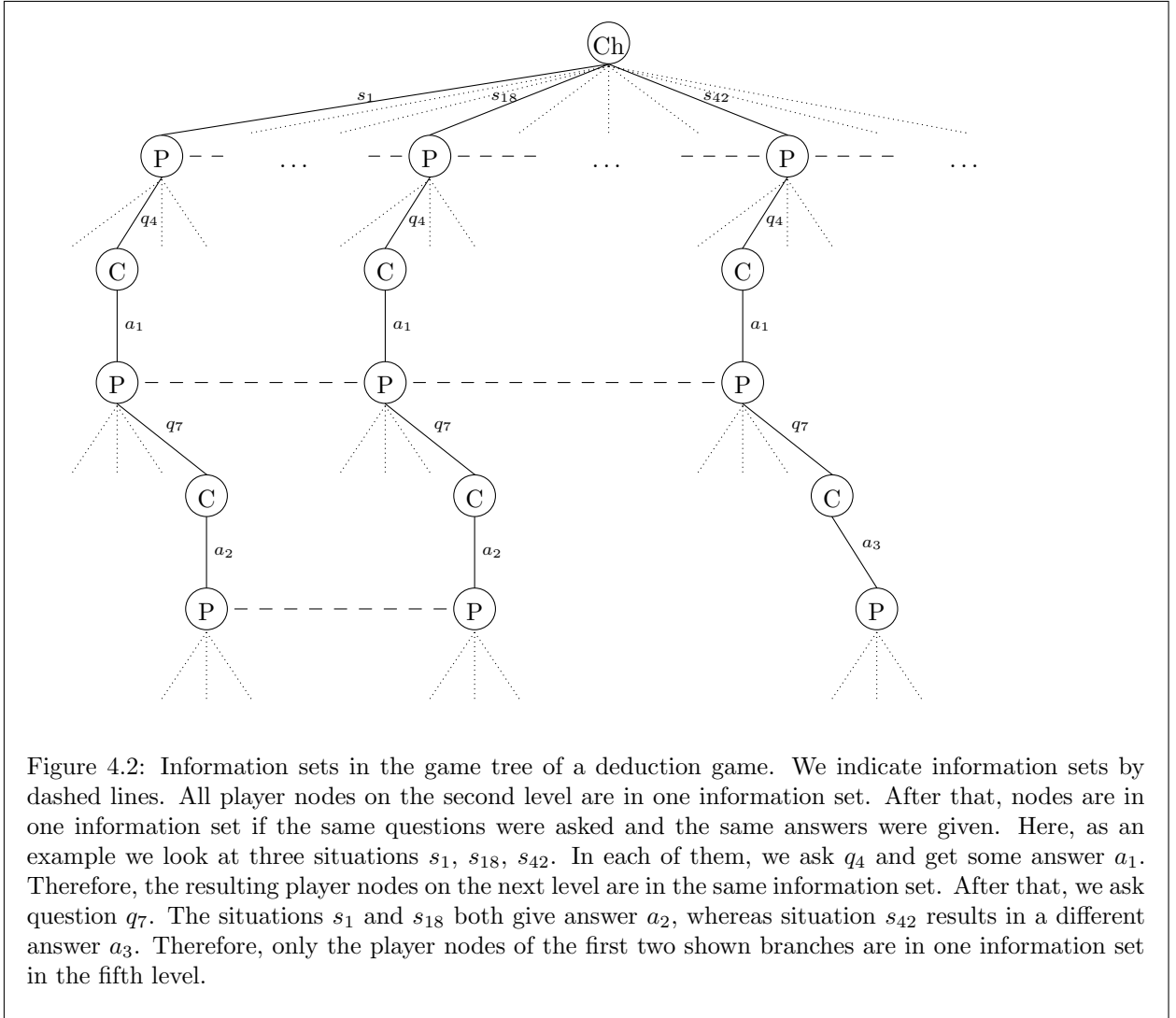


Figure 4.2: Information sets in the game tree of a deduction game. We indicate information sets by dashed lines. All player nodes on the second level are in one information set. After that, nodes are in one information set if the same questions were asked and the same answers were given. Here, as an example we look at three situations  $s_1$ ,  $s_{18}$ ,  $s_{42}$ . In each of them, we ask  $q_4$  and get some answer  $a_1$ . Therefore, the resulting player nodes on the next level are in the same information set. After that, we ask question  $q_7$ . The situations  $s_1$  and  $s_{18}$  both give answer  $a_2$ , whereas situation  $s_{42}$  results in a different answer  $a_3$ . Therefore, only the player nodes of the first two shown branches are in one information set in the fifth level.

Consequently, every information set contains at most one node from every subtree after Chance's move, and all of the information set's nodes are on the same level. It also follows that whenever two nodes are in the same information set, their parent nodes are in the same information set as well.

Looking at it from another perspective, we can see that before the first question, there is just one large information set that contains one node for every situation. From there on, every question  $q$  "splits" this information set into up to  $|\overline{\mathcal{A}}_q|$  information sets on the next layer. Since for the first question asked in the game all answers in  $\overline{\mathcal{A}}_q$  are still possible, in the second layer we have exactly  $|\overline{\mathcal{A}}_{q_1}|$  information sets after asking the first question  $q_1$ ,  $|\overline{\mathcal{A}}_{q_2}|$  information sets after asking the question  $q_2$ , and so on, so altogether  $|\overline{\mathcal{A}}_{q_1}| + |\overline{\mathcal{A}}_{q_2}| + \dots + |\overline{\mathcal{A}}_{q_{|Q|}}|$  information sets. On the next layer, each of these sets can again be split into up to  $|\overline{\mathcal{A}}_{q_i}|$  information sets by every question  $q_i$  (although now not all options necessarily still occur).

Just to give an example:

**Example 4.15.** *Let us assume we are playing the number guessing game with 100 numbers. In the root, Chance selects one number. Since the player does not know about the outcome, all nodes on the second level are in one information set.*

*Each of these nodes has one outgoing edge for "Is the number larger than 38?". For 38 numbers, the answer is "no", and for the other 62 numbers, the answer is "yes". The player cannot differentiate between those 38 nodes (yet), nor can he differentiate between the 62 other nodes. He can however keep the 38 that answered "no" apart from the 62 that answered "yes". Therefore, this gives two information sets on the*

fourth level, one containing the 38 nodes after having asked “Is the number larger than 38?” and received “no” as answer, and one containing the 62 after having asked the same question and received “yes” as answer.

We could however also have asked any other question on the second level. For example, asking “Is the number larger than 90?” will, analogously, lead to one information set on the fourth level with 90 and one with 10 nodes.

On level 4, we therefore have 2 information sets for each question, for a total of 198 information sets. (The available questions only go up to 99, not up to 100.)

Let us look at the information set with the 38 elements from before again, and let us for convenience call this set  $B$  from now on (so that we do not need to write “the information set after having asked whether the number is larger than 38 and received ‘no’ as answer” every time). Every node in  $B$  has an outgoing edge corresponding to “Is the number larger than 13?”. Of the 38 nodes in  $B$ , 13 will answer “no” and 25 will answer “yes”. This leads to 2 information sets on level 6. Note how for each node in  $B$ , the outgoing edge for this question leads to exactly one of the two new situation sets. Thus, the information set  $B$  appears to be “split” into two new sets.

If we ask a different question in  $B$ , say, “Is the number larger than 1?” we would reach different information sets on level 6. Thus, every question “splits”  $B$  into new information sets.

Well, every question? Let us look at “Is the number larger than 70?”, which is a perfectly valid (though pointless) question. Every node in  $B$  answers “no” to it, so it leads only to one information set on level 6. However, we already said before that each question  $q$  “splits” the information set into up to  $\overline{A}_q$  information sets on the next player level. Only on the very first level, all possible answers of a question still have to appear, since this is how we defined  $\overline{A}_q$ .

For every information set, we can easily find the situations the player still believes possible: the player believes a situation  $s \in \mathcal{S}$  to be still possible if and only if the information set he is in contains a node from the subtree after Chance’s move that corresponds to  $s$ . The objective of the player is to find out which of the still possible situation he actually is in, or at least restrict it to a sufficiently small subset of situations to be able to choose a good estimate.

From the observations above it follows that after asking a question, the new information set the player gets into will contain a subset of the situations that were contained in the previous information set. Or, putting it more simply: The answer might rule out a few situations that were up to this point still considered possible, but cannot re-allow a situation that was already ruled out by one of the previous questions.

## 4.2.2 Expected costs

A short word on our usage of “expected costs”: When looking for average-optimal solutions, we will of course frequently talk about “expected costs” and mean the average cost that we have to expect.

If looking for worst-case-optimal solutions, on the other hand, we are more interested in the worst possible cost that can occur. However, “worst possible cost” is a bit unwieldy to write, and in many places, the argumentation will be equivalent for those two. Writing “expected costs (if looking for average-optimal solutions) or worst possible costs (if looking for worst-case-optimal solutions)” every time would just be very annoying.

We therefore assume to be pessimists: when looking for a worst-case-optimal solution, we *expect* the worst possible case to happen. Fortunately, when looking at worst-case-optimal solutions we also do not care about any averages either, so there is no risk of confusion. Thus, when writing about worst-case-optimal solutions in the future, *expected cost* simply means *worst possible cost*.

## 4.2.3 Payoffs

Let us take a closer look at the so-called estimate leafs. Technically, after deciding that we will take a guess, we also have to decide *what* guess we take. Therefore, each estimate “leaf” should actually be followed by  $|\mathcal{E}|$  outgoing edges, one for each estimate  $e \in \mathcal{E}$  we can take (see Figure 4.3). Each of those would then lead to an actual leaf in which the actual situation (chosen by Chance at the beginning) and the chosen estimate (chosen by the player in the previous step) are known and the payoff can be calculated.

Practically, there is little point in choosing anything but the best estimate, so instead of actually adding these nodes to our game tree, we just calculate the results of this last step directly.

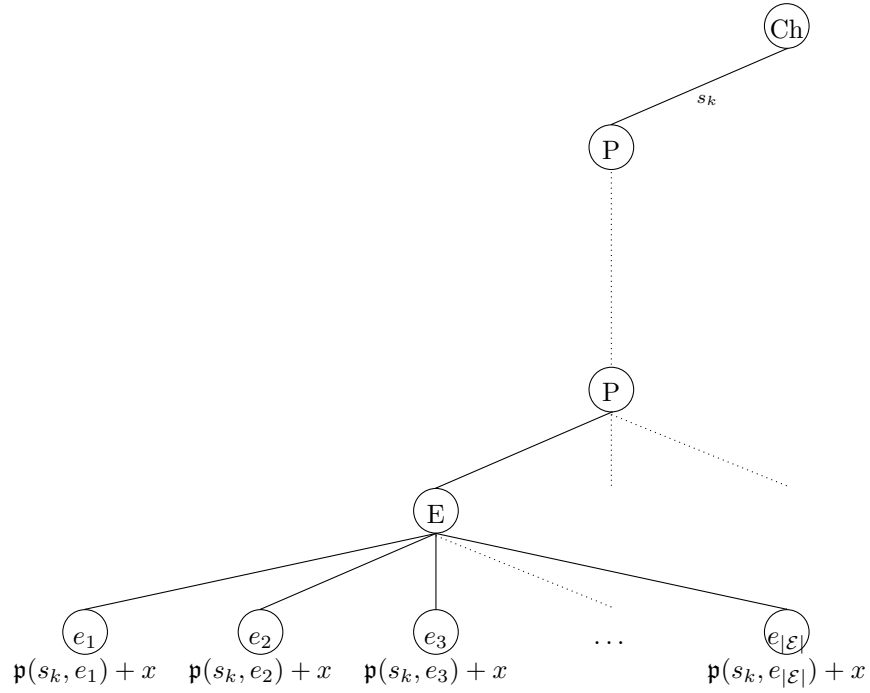


Figure 4.3: Estimate leafs and their “imaginary” children. Below each node, the payoff in that node is given, where  $x$  denotes the cost of the questions and answers on the path from the root to the estimate node.

Let us therefore calculate the payoffs as follows:

**Algorithm 4.16.** Let  $E$  be an estimate leaf and let  $\langle E \rangle$  be the set of estimate leafs in the same information set as  $E$ . Each node in  $\langle E \rangle$  corresponds to exactly one situation in  $\mathcal{S}$ . Let  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  be the subset of situations that have nodes in  $\langle E \rangle$  corresponding to them.

If we want to find an average-optimal solution for the game, choose the estimate  $\bar{e} \in \mathcal{E}$  with the best expected average over all situations  $s \in \hat{\mathcal{S}}$  using the weight function  $\mathbf{w}$ . That is, choose  $\bar{e}$  such that

$$\frac{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot p(s, \bar{e})}{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)}$$

is minimal.

If we want a worst-case-optimal solution, choose  $\bar{e}$  as the estimate with the lowest worst-case outcome over all situations in  $\hat{\mathcal{S}}$ . That is, choose  $\bar{e}$  such that

$$\max_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot p(s, \bar{e})$$

is minimal.

Either way, set the payoff in  $E$  to  $p(s(E), \bar{e}) + x$ , where  $s(E)$  is the situation corresponding to  $E$ , and  $x$  is the sum of the answer costs that occurred on the path to  $E$ .

For optimization reasons, we can at this point also immediately calculate the costs for all other estimate leafs in  $\bar{E} \in \langle E \rangle$  as  $p(s(\bar{E}), \bar{e}) + x$ .

Simply put:

- If looking for an average-optimal solution, choose the estimate with the best expected average outcome. That outcome is the expected outcome for the estimate leaf.
- If looking for a worst-case-optimal solution, choose the estimate with the lowest worst-case outcome. That outcome is the expected outcome for the estimate leaf.

**Theorem 4.17.** *Algorithm 4.16 returns the same cost that would occur if we added the nodes for choosing an estimate as described above and let the player play an optimal strategy on them.*

*Proof.* This follows quite straightforwardly from the definition of Algorithm 4.16. Since a player could not know which node of an information set he is in, he would have to pick the best estimate using the same criteria.  $\square$

Note however that our algorithm said “choose the best estimate”, not “calculate the outcomes of all estimates and choose the best”. For many games, there are better ways to calculate the best estimate than having to calculate all of them. We will look at the individual games again in Chapter 7 and try to find for each of them an efficient algorithm to calculate the best estimate.

#### 4.2.4 Finding optimal strategies and redefining payoffs

First we will repeat how to find best strategies using the rules explained in the introduction in Section 4.1.3. Then we will show an equivalent but slightly more efficient method.

**Algorithm 4.18.** *We calculate the expected overall outcome by recursively calculating the expected outcome in each information set.*

*For each estimate leaf  $E$ , calculate the payoffs as described earlier in Section 4.2.3. The expected outcome for an estimate leaf equals its payoff.*

*Let  $N$  be a player node for which we want to calculate the expected outcome, and let  $\langle N \rangle$  be the set of player nodes in the same information set as  $N$ . First calculate the expected outcomes of all children of all nodes in  $\langle N \rangle$ .*

*By definition, all nodes in  $\langle N \rangle$  have the same outgoing edges; they all have one outgoing edge for each question  $q \in \hat{Q} \subseteq Q$  that can still be asked, plus one edge leading to an estimate leaf. Let us call those  $|\hat{Q}| + 1$  edges the “types” of outgoing edges that nodes in  $\langle N \rangle$  have. Each type represents one possible move that the player can take, and we have to find out which one is best.*

*If looking for an average-optimal solution, we therefore for each type of outgoing edge look at the children of the nodes in  $\langle N \rangle$  that correspond to it, calculate the expected costs of those children, and average those costs (using the weight function  $\mathbf{w}$ ). If we chose that move, this would be our expected cost. We then choose the move for which this expected cost is lowest. (This can be either a computer node if asking a question, or an estimate leaf if taking a guess.) This is at the same time the expected cost for  $N$ .*

*If looking for a worst-case-optimal solution, then we again calculate for each type the costs of all children we could reach if we took that move. Then we choose the move for which the worst outcome we could get is most harmless, that is, for which the most expensive child is cheapest. Again, this is the expected cost for  $N$ .*

*For computer nodes  $C$ , since there is only one outgoing edge, the expected cost trivially equals the cost of the only child.*

*For the Chance node finally, all its children are in one information set and therefore have the same expected cost anyway. We simply set the expected cost in the root to that cost.*

However, this generic approach does not take into account that our payoffs are strongly linked to the questions that were asked, and thus to the tree structure itself. In particular, the payoffs contain the costs of the questions on the path from the root to the estimate. Instead of calculating all the costs in the leaves, we therefore now put the costs to exactly the place where they occur: the answer edges from computer nodes.

In Algorithm 4.18, we calculated in each node the expected *total* cost for playing game. In the next algorithm, we will split these costs into two parts: the costs that already occurred on the path from the root to that node, plus the costs that we expect to still have to pay in future. (We will, however, only calculate the expected future costs explicitly, and will use the past costs only for proving equivalence.) Intuition tells us that this should be equivalent to the other algorithm, and mathematics will prove us right.

In fact, one might argue that the following even is the more “natural” approach to the problem. However, Algorithm 4.18, in which the payoffs are defined in the leaves, is how calculating optimal strategies is usually done in game theory. Consequently it is the one for which we know from literature that it indeed finds optimal strategies. In order to prove that the following Algorithm 4.19 correctly calculates expected costs as well, we will show that it is equivalent to Algorithm 4.18.



**Algorithm 4.19.** First, remove the payoffs that were calculated in Section 4.2.3 again. We will instead label the edges with costs in the following way:

- For each edge leading from a player node to an estimate leaf, calculate the expected payoff for the estimate the same way as in Algorithm 4.16, but without adding the cost  $x$  of the questions asked on the path from the root to that estimate leaf. Using the symbols from that algorithm, label the edge leading to the estimate leaf  $E$  with  $\mathbf{p}(s(E), \bar{e})$ , the expected penalty for the estimate.
- For each edge leading from a computer node to a player node, label the edge with the cost corresponding to the answer given by the computer.
- Label all other edges with 0.

(See Figure 4.4.)

We immediately see that the payoffs we originally used in the estimate leafs in Algorithm 4.18 now equal the sum of the newly defined edge costs between the root and the estimate leaf. Instead of calculating the expected cost in each node, we will now calculate the expected future costs after that node, i.e., the costs we expect to have to pay for all future moves, but not counting the costs for moves already made.

For each estimate leaf  $E$ , set the expected future cost to 0.

Let  $N$  be a player node for which we want to calculate the expected future cost, and let  $\langle N \rangle$  be the set of player nodes in the same information set as  $N$ . Let “types” of edges be defined as in Algorithm 4.18.

For each type of outgoing edge, look at all the children that are reached from nodes in  $\langle N \rangle$  by following this type of edge. For each child, calculate the expected future cost of that child, and add the cost of the edge that leads to it. If we chose that move, we would set the expected future costs of  $N$  to the weighted average of these costs. If looking for an average-optimal solution, we therefore again choose the move (or type of question) for which these expected future costs are lowest.

If looking for a worst-case-optimal solution, then we again choose the type of edge for which the most expensive child is cheapest, and set the expected future costs to that value.

For computer nodes  $C$ , the expected future cost equals the cost of the only child plus the cost of the edge leading to it.

For the Chance node, again all children are in one information set and therefore have the same expected future costs, and the edges leading to them are all free. We therefore again simply set the expected future costs of the root to the expected future costs of its children.

**Theorem 4.20.** Algorithms 4.18 and 4.19 will result in the same strategy and the same expected outcome in the root node.

*Proof.* We will show that in each node, the sum of expected future costs and the costs on the path from the root to that node from Algorithm 4.19 equals the expected costs from Algorithm 4.18. Let  $f(A)$  denote the expected future costs in a node  $A$  as calculated in Algorithm 4.19, let  $p(A)$  denote the sum of the (past) costs of the edges on the path between the root and the node  $A$ , and let  $c(A)$  denote the expected costs of node  $A$  as calculated in Algorithm 4.18. We thus want to prove that for each node  $A$  it holds that  $f(A) + p(A) = c(A)$ .

To do so, we will use complete induction going from the leafs upwards. For the estimate leafs, we have basically pointed this property out already, since  $c(A) = p(A)$  by definition, and  $f(A) = 0$ .

Let now  $N$  be a player node so that for all children of all nodes in the information set  $\langle N \rangle$  the equality holds. First we show that the same child is chosen with both algorithms. By our induction hypothesis, for each child  $C$  of a node in  $\langle N \rangle$  it holds that  $f(C) + p(C) = c(C)$ . Algorithm 4.18 simply chooses the outgoing edge for which the weighted average (or worst-case) of the corresponding children is minimal. That is, for each child node, we consider its cost  $c(C)$  and compare those costs. Algorithm 4.19 on the other hand compares the sum of the future cost of the child and the edge between the parent node and the child. Let  $x(N, C)$  denote the cost of the edge between  $N$  and  $C$ , then this algorithm compares  $f(C) + x(\text{parent}(C), C)$  of each child  $C$ . However, since  $p(X)$  is defined as the sum of the costs on the path from the root to  $X$ , we immediately see  $x(\text{parent}(C), C) = p(C) - p(\text{parent}(C))$ , and thus

$$\begin{aligned} f(C) + x(\text{parent}(C), C) &= f(C) + p(C) - p(\text{parent}(C)) \\ &= c(C) - p(\text{parent}(C)) . \end{aligned}$$

However, since  $p(\tilde{N})$  is the same for all nodes  $\tilde{N} \in \langle N \rangle$ , all considered costs are only different by a constant offset of  $p(N)$ . Therefore, the same child is chosen by the two algorithms. (This holds for both average-optimal and worst-case-optimal calculations. We do however need to demand that ties between equally good moves are broken in the same way.)

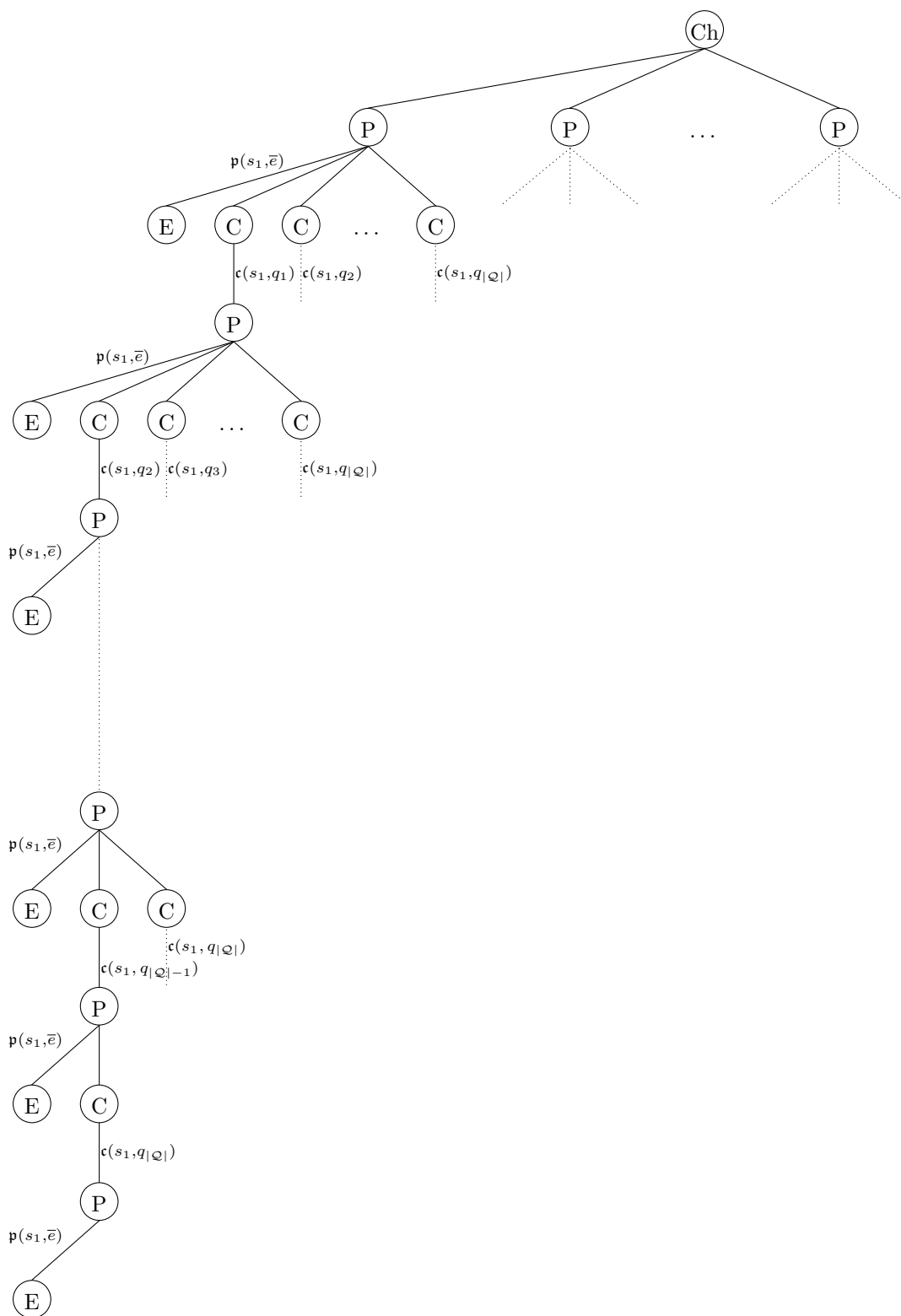


Figure 4.4: Redefined edge costs after Algorithm 4.19. Labels with value 0 are omitted to make the graph simpler.

Now we need to show that the desired equality holds in the player node. Let  $\tilde{C}$  be the child chosen by the algorithm(s). Looking at Algorithm 4.19 above, we see that for each type of outgoing edge, this edge is associated with the same costs for all nodes  $\tilde{N} \in \langle N \rangle$ . Therefore, the difference between the expected future cost of a parent node and its child node always equals the cost of the edge between them. Mathematically speaking,  $f(N) = f(\tilde{C}) + x(N, \tilde{C})$ . On the other hand,  $x(N, \tilde{C})$  is also the cost difference between the path from  $\tilde{C}$  to the root and that from  $N$  to the root:  $p(N) = p(\tilde{C}) - x(N, \tilde{C})$ .

We therefore get:

$$\begin{aligned} f(N) + p(N) &= f(\tilde{C}) + x(N, \tilde{C}) + p(\tilde{C}) - x(N, \tilde{C}) \\ &= f(\tilde{C}) + p(\tilde{C}) \\ &= c(\tilde{C}) \end{aligned}$$

Since Algorithm 4.18 would also choose  $\tilde{C}$  and set the expected cost of  $c(N)$  to  $c(\tilde{C})$ , we get  $c(N) = c(\tilde{C}) = f(N) + p(N)$ , the desired equation.

For showing the equality in the computer nodes, we can basically reuse the second part of the proof. Let  $A$  be a computer node. There is only one outgoing edge, so we only have one child  $C$ . Using Algorithm 4.18, we get  $c(A) = c(C)$ . Using Algorithm 4.19, we get  $f(A) = f(C) + x(A, C)$ . Combining these and using  $p(A) = p(C) - x(A, C)$  again, we get

$$\begin{aligned} f(A) + p(A) &= f(C) + x(A, C) + p(C) - x(A, C) \\ &= f(C) + p(C) \\ &= c(C) \\ &= c(A) , \end{aligned}$$

the desired equality.

For all nodes  $B$  on the second level (after Chance's move) we therefore have  $c(B) = f(B) + p(B) = f(B) + 0$ . For those nodes, the costs calculated by the both algorithms are therefore the same, and consequently so are the costs calculated in the root node, since the root's cost is set to the cost of the nodes on the second level.

Therefore, in the root  $R$ , the calculated expected costs  $c(R)$  and expected future costs  $f(R)$  are the same, and in all player nodes the same decisions are taken.  $\square$

#### 4.2.5 Situation subtrees

For convenience we introduce another definition:

**Definition 4.21** (situation subtree). *The SITUATION SUBTREE for a situation  $s \in \mathcal{S}$  is the subtree after Chance's move that corresponds to  $s$ .*

Note that within one situation subtree, the answer for each question  $q$  is always the same.

### 4.3 Decision sets

As we can see, the number of information sets for which we need to calculate best moves (and their expected outcomes) is huge. In order to be able to calculate them (in less than a human lifetime), we need to find simplifications. We therefore introduce the concept of decision sets:

**Definition 4.22** (decision set). *A DECISION SET is a subset of the player nodes with the following properties:*

- *In all nodes the same move is the best move (or among the best moves, if there is more than one best move).*
- *In all nodes the expected future cost is the same.*

Note that we do not demand that the same moves are available in all nodes; it is sufficient if there is one move that is available in all nodes and is the best move in each of these nodes. Also note that the cost of questions that were asked already is irrelevant, only the expected cost of future questions (in perfect play) needs to be the same.

We also do not demand that all nodes in which the same move is optimal are in one decision set, we only vice versa demand that the same move is optimal for all nodes in a decision set.

The idea is this: We want to group the nodes of the game tree into decision sets so that instead of calculating the best move for every node or every information set, we only need to calculate the best move for every decision set. The more nodes we can prove (with theoretical means) to be in the same decision set, the less we have to calculate.

Looking at it from another angle, this is closely related to the idea of dynamic programming: instead of doing the same calculation several times, we want to store the calculation results. However, currently each node is a completely different calculation to us, so we cannot reuse these results (since we access them only once during the calculation anyway). What we want to do now is to prove that certain nodes have the same result, so that we can calculate this result once and then reuse it for all equivalent nodes.

The most trivial choice for decision sets would be to put every node into its own decision set. (Single nodes fulfill the requirements for decision sets for obvious reasons.)

Another still trivial choice for decision sets are the information sets, since we *have* to choose the same move for all nodes in an information set anyway, and the expected future costs for all nodes in an information set are the same as well.

### 4.3.1 Ignoring the order of previously asked questions

Instinctively we would assume that the order in which previous questions were asked is irrelevant. Now we only need to prove that our instincts are right.

In the next section we will prove an even stronger version of this. However, since the topic of this thesis is relatively new, proving this weaker version can be seen as warm-up for later considerations.

**Theorem 4.23.** *Let  $A$  and  $B$  be two information sets in which the same questions have been asked already and the same answers were given. Then in  $A$  and  $B$  the same next move is optimal.*

*Proof.* Since the same questions were asked already, the same remaining questions (and thus moves) are available in  $A$  and  $B$ , and since the same number of questions was asked,  $A$  and  $B$  are on the same height in the tree. Also, their costs up to that point are the same.

We know that for  $A$  and  $B$  the same questions were asked and the same answers were given. Therefore, a situation is still possible in  $A$  if and only if it is still possible in  $B$ . Consequently, each situation subtree contains either both a node from  $A$  and a node from  $B$ , or neither.

Let  $s$  be a situation that is still considered possible in  $A$  and  $B$ , and let  $a \in A$  and  $b \in B$  be the nodes from  $A$  and  $B$ , respectively, that are in the situation subtree of  $s$ . Then the subtree with  $a$  as a root identical to the one with  $b$  as its root, since within one situation subtree, the same questions always have the same answer. Therefore, each possible move in  $a$  exactly corresponds to one possible move in  $b$ , and they have the same cost.

Combining this, we see that each node in  $A$  exactly corresponds to one node in  $B$ . We also know that all nodes in  $A$  have the same types of outgoing edges. Consequently,  $A$  also has the same types of outgoing edges as  $B$ , and every possible move in information set  $A$  exactly corresponds to one possible move in information set  $B$ , again with the same expected cost. Therefore, the same move is optimal in  $A$  and  $B$ .  $\square$

This shows that if for any two information sets the same questions were asked and the same answers were given, then the same next move is optimal, and therefore we can combine these two information sets into one decision set.

### 4.3.2 Considering only the set of still possible situations

We now show a much stronger version of Theorem 4.23.

**Theorem 4.24.** *Let  $A$  and  $B$  be information sets in which the same situations are still considered possible. Then in  $A$  and  $B$  the same next move is optimal.*

*Proof.* In contrast to the previous proof, we now cannot assume that the same questions are still available, or that  $A$  and  $B$  are on the same height in the game tree. We therefore need to show that all questions that are not available any more in either one of the two information sets are not optimal. (Or, more precisely, that there exists an optimal move that is available in both  $A$  and  $B$ .)

A question is available if and only if it has not been asked yet. Let for example  $q$  be a question that is still available in  $A$ , but was asked already in  $B$ . Then for all situations still considered possible in  $B$ , the answer given to that question must be the same. However, the same situations are possible in  $A$  and in  $B$ , so that answer also applies to all situations in  $A$ . If we asked that question in  $A$ , we therefore would not get any new information, but would have to pay the cost for the question.

Note that we do not care about the actual answer, we just needed to show that the answer is the same for all situations in  $A$ . Therefore, such a question cannot be optimal. (At best it is free, but still does not give us any information.)

More mathematically, let us assume that some question  $q$  has the same answer  $a$  for all questions in  $A$ . This means that asking  $q$  does not “split” the information set. By asking the question, we reach an information set on the next player level of the game tree that contains exactly the same situation subtrees; let  $A'$  be that information set. Looking at the sets  $A$  and  $A'$  and all their successors, their only difference is that the questions corresponding to  $q$  do not exist in the latter any more. Let us assume that we find an optimal strategy on information set  $A'$  and its successors, and  $f(A')$  is the expected future cost of that strategy. If we decide to ask question  $q$  in  $A$ , then the expected future cost  $f(A)$  equals  $f(A')$  plus the cost of the question  $q$ . (That is because for each node in  $A$ , asking  $q$  leads to a node in  $A'$ , and all nodes in  $A'$  have the same expected future costs since they are in the same information set.) That is,  $f(A) = f(A') + c(q, a)$ . However, since the optimal strategy used in  $A'$  only contains questions that are also available in  $A$ , we could simply have used this strategy in  $A$  directly, without asking  $q$  first, and get the same cost  $f(A) = f(A')$ . At best,  $c(q, a) = 0$ , in which case the useless question only did not help. At worst, asking it adds unnecessary costs.

For finding the best move we therefore only need to consider questions that were asked neither in  $A$  nor in  $B$  yet. When we ignore all other questions, we can again use the same method as in the previous proof:

Let again  $a \in A$  and  $b \in B$  be the nodes from  $A$  and  $B$  in the situation subtree for a fixed situation  $s \in S$ . Then the subtree with  $a$  as a root is exactly the same as the one with  $b$  as its root, with the exception of some questions that we can ignore. Therefore, each potentially optimal move in  $a$  exactly corresponds to one such move in  $b$ , and they have the same cost. We can again conclude that the same move is optimal in  $A$  and  $B$ .  $\square$

Let us look at an example for clarification. We are playing the number guessing game with numbers from 1 to 100. In information set  $A$ , we asked whether the chosen number is larger than 10 (yes), 30 (yes), 90 (no) and 60 (no). In information set  $B$  we asked whether it is larger than 60 (no), 20 (yes) and 30 (yes). Both in  $A$  and  $B$  we consider the numbers from 31 to 60 still possible. The proof shows that the three questions asking whether it is larger than 10, 20 and 90 would not give us any additional information, so we can ignore them. Therefore it does not matter that these questions are not available any more in one of the two information sets because they would not be optimal anyway.

This means that the only thing we still have to consider is subsets of  $S$  that can occur as sets of still possible situations:

**Corollary 4.25.** *Let  $\hat{S} \subseteq S$  be a subset of the possible situations. Then all information sets in which the situations in  $\hat{S}$  are exactly the situations that are still considered possible can be combined into one decision set.*

*Proof.* This follows directly from the previous considerations.  $\square$

Looking at the example above, we immediately see that 10, 20 and 90 are not the only questions that are not interesting any more once we know that the number is in the range between 31 and 60. All questions that are outside this range (and that would therefore give the same answer for all number in that range) are worthless.

We can draw a nice little corollary from Theorem 4.24 in which we state just that:

**Corollary 4.26.** *Let  $\hat{S} \subseteq S$  be a set of situations, and let  $q \in Q$  be a question that has the same answer for all situations in  $\hat{S}$ . Then in the decision set corresponding to  $\hat{S}$ , the question  $q$  is not optimal (or is at least not the only optimal question).*

*Proof.* Let  $A$  be an information set for which  $\hat{S}$  is the set of still possible questions. If we ask  $q$  now, this does not split the information set, so we get to an information set  $A'$  in which the same situations  $\hat{S}$  are still

possible, but  $q$  is not a valid move any more. According to the previous theorem, the same move is optimal in  $A$  and  $A'$ . Since asking  $q$  is not available in  $A'$  any more, a different question  $q'$  is the optimal question in both  $A'$  and  $A$ . Consequently,  $q$  cannot have been the [only] optimal question in  $A$ . (Note that  $q$  can still be among the optimal questions if it is free. It is nonetheless useless, though.)  $\square$

Looking at the same example as above with numbers from 31 to 60 still possible, we therefore can now ignore all questions below 30 and all above 60, since they would not change the set of possible situations.

## 4.4 An equivalent game

We will now look at a different game and will show that it is equivalent. This will lead to a game tree that has a different form, but the same average outcome and equivalent optimal strategies. We will also find that we can use the same decision sets as before. Having two ways leading to the same decision sets will help us later on to find better upper and lower bounds on the number of decision sets.

This time, the computer does *not* choose the secret situation in the beginning. Instead, the player makes the first move by asking a question. Only now the computer chooses one of the possible *answers* for that question. The player then asks the next question, and again the computer chooses an answer in such a way that it does not contradict the previously given information, and so on. Only when the player finally takes a guess, the computer chooses a situation that is consistent with the answers that were given, and calculates the penalty.

The computer chooses answers in the following way:

**Algorithm 4.27.** Let  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  be the set of situations that correspond to all answers that were already given by the computer, i.e., the situations that are still possible.

Let furthermore  $q$  be the question asked by the player.

For each possible answer  $a \in \mathcal{A}_q$ , calculate the weight  $\hat{w}_a$  by adding the weights of all situations in  $\hat{\mathcal{S}}$  for which  $a$  is the answer to  $q$ :

$$\hat{w}_a := \sum_{\substack{s \in \hat{\mathcal{S}} \\ r(s,q)=a}} w(s)$$

Choose one of the answers in  $\mathcal{A}_q$  using the weights  $\hat{w}_a$ .

Let us give an example:

**Example 4.28.** Once again, we play the number guessing game with numbers from 1 to 100. In the first move, the player can choose which question to ask. Let us assume he asks whether the number is larger than 86.

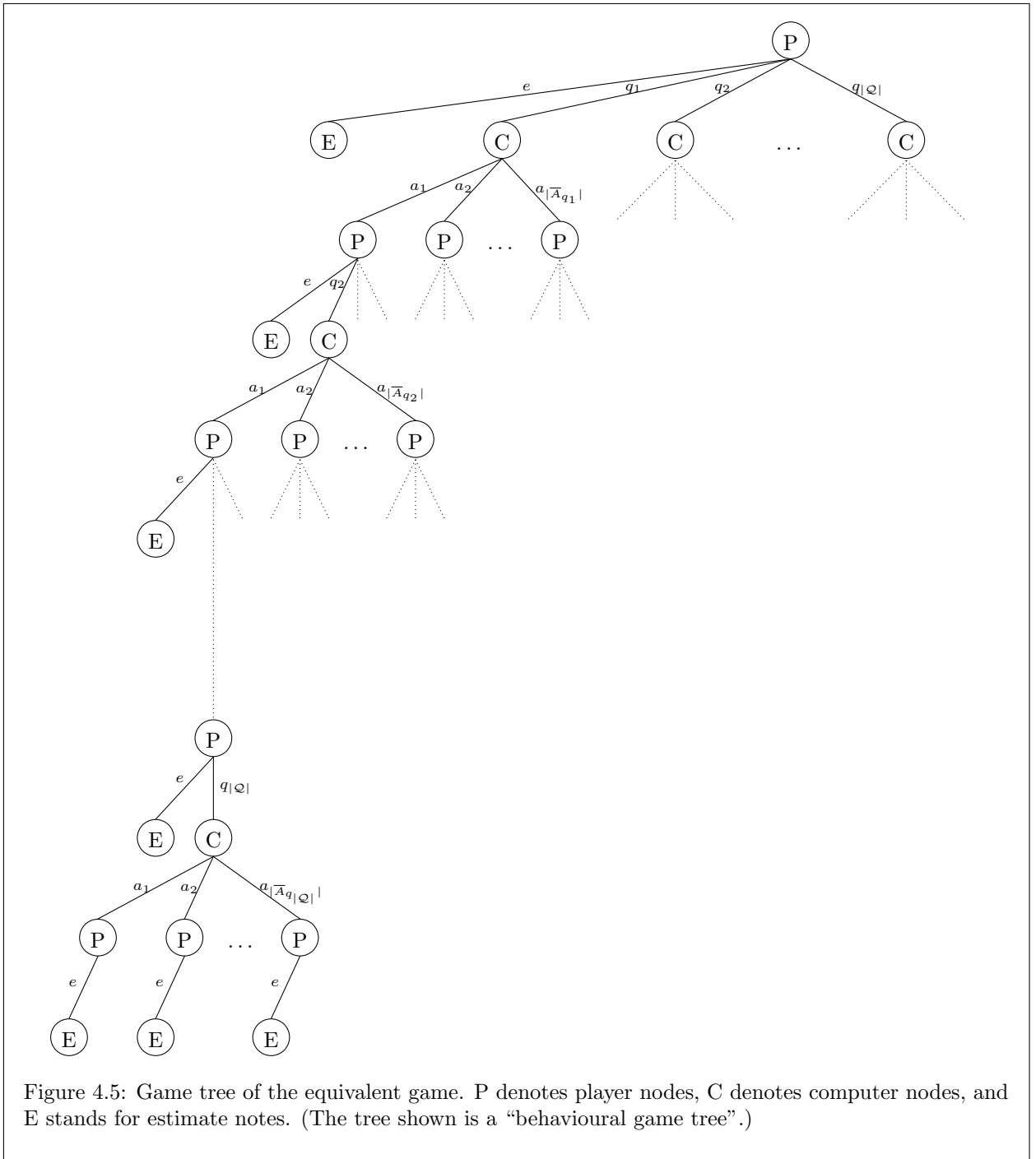
There are 86 questions for which the answer is “no”, and 14 questions for which the answer is “yes”. If the computer had already chosen a situation, the probability that the answer is “no” would therefore be 86%. Based on these probabilities, the computer now chooses an answer, choosing “no” with a probability of 86% and “yes” with a probability of 14%. Let us assume the computer answers “no”.

The remaining situations that are possible are the numbers from 1 to 86. The player can decide on a question again. Let us assume he next asks whether the number is larger than 36. There are 36 remaining situations for which the answer is “no”, and 50 situations for which the answer is “yes”. If the computer had chosen a situation at the beginning, the probability that “no” is the answer would therefore be  $\frac{36}{86} \approx 42\%$ , and the probability that the answer is “yes” would be  $\frac{50}{86} \approx 58\%$ . The computer again chooses an answer according to these probabilities (but still does not decide on a situation).

**Definition 4.29.** For simplicity, we introduce the following notations for the remaining situations and remaining possible answers, respectively:

$$\begin{aligned} \hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k)) &:= \{s \in \mathcal{S} \mid r(s, q_1) = a_1, r(s, q_2) = a_2, \dots, r(s, q_k) = a_k\} \\ \hat{\mathcal{A}}_q((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k)) &:= \{a \in \mathcal{A} \mid \exists s \in \hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k)) : r(s, q) = a\} \end{aligned}$$

We now have a very different game tree (which we see in Figure 4.5):



First of all, the first move is now done by the player, not by the computer. The player has the choice between  $|Q|$  questions, or taking a guess. After that, the computer moves, using the algorithm above. Contrary to the previous game tree, the computer node now has not only one outgoing edge, but rather one for every answer  $a$  for which the weight  $\hat{w}_a$  is not 0. Then the player moves again and can choose any question except for the one that was asked already. Then the computer moves again, again using the algorithm above, and so on.

Basically, the computer now represents Chance in each of its moves (in addition to answering), whereas in the original game it was only choosing randomly once at the beginning of the game.

Counting nodes is a lot harder this time.

- There is one root node with  $|Q| + 1$  outgoing edges.
- In the next level, we have one leaf for taking a guess, and  $|Q|$  computer nodes. The number of outgoing edges of each computer node is the number of possible answers for the question that was asked,  $|\overline{A_q}|$ .
- On the third level, we therefore have  $\sum_{q \in Q} |\overline{A_q}|$  player nodes. Each of them has  $|Q| - 1$  outgoing edges for questions and one for taking a guess.
- Accordingly, we have  $\sum_{q \in Q} |\overline{A_q}|$  estimation nodes on the next level, and  $(|Q| - 1) \cdot \left(\sum_{q \in Q} |\overline{A_q}|\right)$  computer nodes.

The number of outgoing edges in each of these computer nodes now depends on the number of different answers that are still possible after the answer given to the first question. Let  $q_1$  be the first question asked by the player (i.e., the one in the node three levels above on the path from the root to the node we are looking at),  $a_1$  the given answer, and  $q_2$  the second question. The set of still possible situations is now  $\hat{S}((q_1, a_1))$ . The number of possible answers therefore is  $|\hat{A}_{q_2}((q_1, a_1))|$ .

- On the next level, we therefore altogether have

$$\sum_{q_1 \in Q} \sum_{a_1 \in \overline{A_{q_1}}} \sum_{q_2 \in Q \setminus \{q_1\}} |\hat{A}_{q_2}((q_1, a_1))|$$

player nodes. Each of them has  $|Q| - 2$  outgoing question edges and one guessing edge.

- Accordingly, on the next level we have the same number of estimate leafs and  $|Q| - 2$  times that number of computer nodes.
- Generally, on the level  $2k + 1$  we have

$$\sum_{q_1 \in Q} \sum_{a_1 \in \overline{A_{q_1}}} \sum_{q_2 \in Q \setminus \{q_1\}} \sum_{a_2 \in \hat{A}_{q_2}((q_1, a_1))} \sum_{q_3 \in Q \setminus \{q_1, q_2\}} \sum_{a_3 \in \hat{A}_{q_3}((q_1, a_1), (q_2, a_2))} \cdots \sum_{q_{k-1} \in Q \setminus \{q_1, q_2, \dots, q_{k-2}\}} \sum_{a_{k-1} \in \hat{A}_{q_{k-1}}((q_1, a_1), (q_2, a_2), \dots, (q_{k-2}, a_{k-2}))} \sum_{q_k \in Q \setminus \{q_1, q_2, \dots, q_{k-1}\}} |\hat{A}_{q_k}((q_1, a_1), (q_2, a_2), \dots, (q_{k-1}, a_{k-1}))|$$

player nodes.

On level  $2k + 2$  we have the same number of estimate leafs and  $|Q| - k$  times that number of computer nodes.

In Section 5.3.3 we will try to find more readable upper bounds for the amount of nodes in this tree. Other than in the original problem, the number now depends a lot on the game.

**Example 4.30.** *Let us first look at the number guessing game for numbers from 1 to 100. Obviously, if the answer to “Is the number larger than 50?” was “yes”, then for the question “Is the number larger than 30?” the answer “no” is not available any more. Therefore, the computer node for answering this question only has one child.*

*Now look on the other hand at the number guessing game with bit questions for numbers from 1 to 127 with 7 bit questions, one for each bit. Even if we have asked 6 of these questions already, for the remaining bit question both answers (“yes” and “no”) are still possible.*

*The number of children of a computer node, and thus the total number of nodes in this game tree, does therefore depend on the problem.*

Note that this time, each information set contains only one node – the player has exactly the same information as the computer.

The payoffs in the estimate leafs are calculated in the same way as in Algorithm 4.16.

**Definition 4.31.** *For consistency with terminology in related fields of game theory we will call this game tree of the alternative game the BEHAVIOURAL GAME TREE [16]. We will refer to the original game tree defined earlier as the STANDARD GAME TREE.*



### 4.4.1 Equivalence to original game

First of all, let us prove that this new game is indeed equivalent to the previous one.

**Theorem 4.32.** *The standard game tree and the behavioural game tree are equivalent from the perspective of the player. (That is, even in repeated play, the player can not determine whether the computer plays the original or the alternative game. Consequently, the expected outcome for every strategy of the player is the same in both cases.)*

*Proof.* We will prove this iteratively for all subgames. In order to do so, let us look at an intermediate version of the game in which we swap the order of the first two steps: Like in the behavioural game tree, the player starts by asking a question. After that, the computer chooses one of the situations to be the secret situation and gives the corresponding answer. From there on, the game continues like in the original game. (See Figure 4.6.)

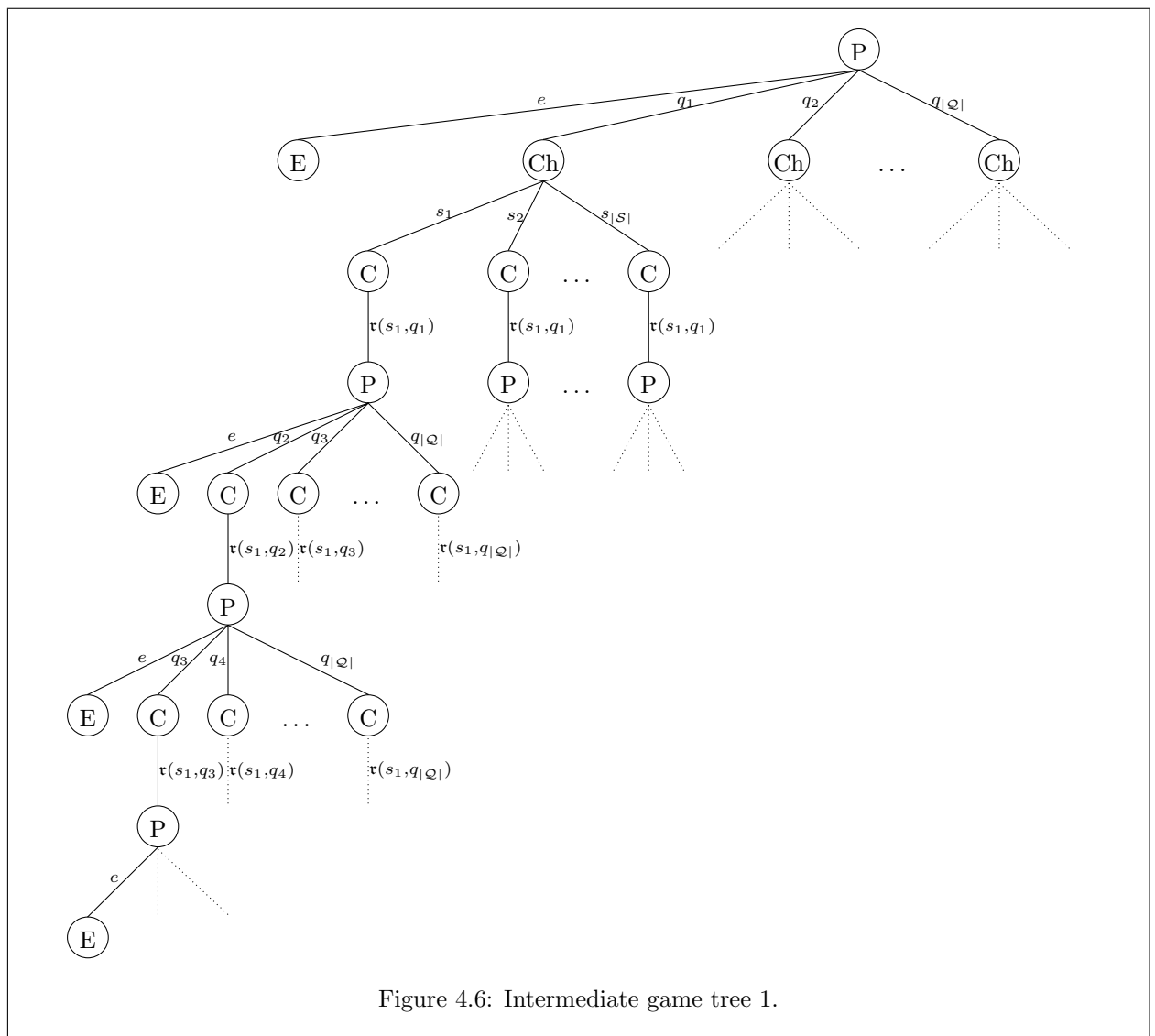


Figure 4.6: Intermediate game tree 1.

Looking at the game tree of this intermediate version, we see the following:

- In the root node, the player chooses to either take a guess or ask one of the available questions.
- On the next level, we have one estimate leaf and  $|Q|$  computer nodes. In each computer node, the computer chooses one of the situations in  $S$  and gives an answer.

- On level three, we therefore have  $|Q| \cdot |S|$  computer nodes. Each of those exactly corresponds to one of the  $|Q| \cdot |S|$  computer nodes on the third level of the standard game tree (in which the same secret situation was chosen and the same question was asked).
- From there on, the game tree is exactly the same as the standard game tree.

Since all subgames with roots in the third level are identical to those in the standard game tree, we only need to show that also the first three levels are equivalent. This is obvious though, since the move of the computer is not influenced by the question that was asked, and the player does not know anything about the outcome of the computer move. There is no way the player can tell whether the computer chose the situation before or afterwards. Therefore we can switch the order of those two moves.

More mathematically, we see that each subtree with a root on level 3 in the intermediate game is identical to one subtree with root at level 3 in the standard game. Therefore, also their expected costs are identical. (The expected future costs anyway because they only depend on the children of a node, and also the traditional expected costs, since the sum of the costs from the root to each estimate leaf stayed the same.) In the upper three levels, the player node that is now root is equivalent to the information set of all player nodes on level 2 in the standard game tree. Looking at how the expected costs were calculated there as average or maximum of all nodes in the information set, we see that now the Chance nodes on the second level, following definitions from Section 4.1.4, calculate expected costs exactly the same way.

Now let us consider another intermediate form that is one more step closer to the behavioural game tree, in which we split the (Chance) move of the computer into two steps: Again, the player first chooses a question. Then the computer chooses an answer using Algorithm 4.27. After that, the computer moves again and chooses one question that corresponds to the given answer. After that, the game continues like in the standard game tree. (See Figure 4.7.)

Let us study this game tree as well:

- In the root node, the player chooses to either take a guess or ask one of the available questions.
- On the next level, we have one estimate leaf and  $|Q|$  computer nodes. In each computer node, the computer chooses one of the answers using Algorithm 4.27.
- On the third level, we have  $\sum_{q \in Q} |\overline{A}_q|$  computer nodes. In each of them, the computer chooses one of the still possible situations.
- On level four, we have computer nodes again, in which the computer gives the answer for the chosen situation. Let  $q$  be a question, then the Chance node on the second level that corresponds to  $q$  has  $\overline{A}_q$  children. However, each situation  $s \in \mathcal{S}$  is possible in exactly one of these children. Therefore, all this children together have exactly  $\mathcal{S}$  outgoing edges. Therefore, there are again  $|Q| \cdot |S|$  computer nodes on the fourth level, each of which again exactly corresponds to one of the computer nodes on the third level of the standard game tree.
- Again, the game tree is exactly the same as the standard game tree from there on.

Basically we just split the move of the computer (“choose a random situation”) into two moves (“choose a random answer and then choose a corresponding situation”). The equivalence follows immediately from our choice of weights in Algorithm 4.27.

Let us finally assume that the computer already announces the chosen answer immediately after choosing it, then we do not need the computer nodes on level 4 for announcing the answer any more and can simply remove them (so that the Chance nodes on level 3 have outgoing edges that lead directly to the player nodes on level 5).

Let us now have a closer look at the subgames beginning on the third level of the second intermediate form. They start with the computer choosing a situation from a given set of available situations, then the player asking a question and the computer answering, and so on. This, however, is simply a standard game tree for a smaller set of available situations and one question less.

We therefore can transform this subtree into an equivalent one using the same method. Repeating this step iteratively all the way down to the leafs, we get the behavioural game tree.  $\square$

Note that we will also call the Chance nodes simply computer nodes from now on, since from the player’s point of view it is not obvious anyway where Chance is moving and where only the computer is answering.

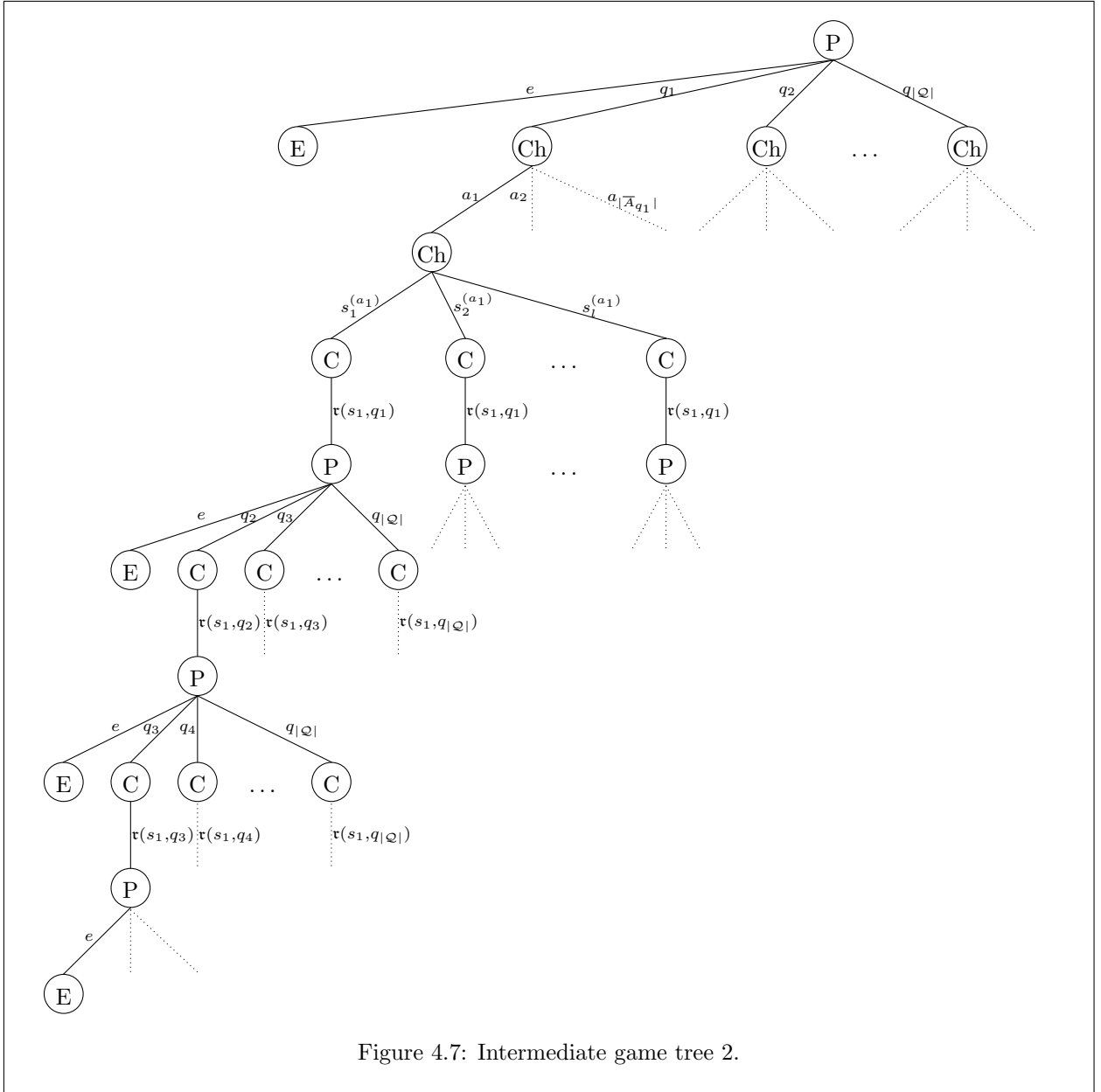


Figure 4.7: Intermediate game tree 2.

A short observation also shows how these trees are related:

**Lemma 4.33.** *Let  $A$  be an information set in the standard game tree. An information set consists of player nodes for which the same questions were asked in the same order, and the same answers were given. Let  $((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$  be this sequence of questions and answers.*

*If we start in the root of the behavioural game tree and follow those questions and answers in this order, we get to a player node  $B$ .*

*The set of remaining situations in  $B$  is the same as the set of remaining situations  $A$ .*

*Proof.* This follows directly from the construction: Both in  $A$  and in  $B$ , a situation  $s$  is still considered possible if and only if it satisfies all (question, answer) pairs.  $\square$

### 4.4.2 Calculating optimal strategies

Again we have two possible ways to calculate optimal strategies. First of all, we need a simplified version of the payoff calculation from Algorithm 4.16. In that algorithm we had to take all the nodes in the same information set into account. Now, on the other hand, each node is an information set by itself and already contains the list of remaining situations.

**Algorithm 4.34.** *Let  $E$  be an estimate leaf and let  $\hat{S} \subseteq \mathcal{S}$  be the subset of the remaining situations in  $E$ . Let furthermore  $x$  be the sum of the answer costs that occurred on the path to  $E$ .*

*If we want to find an average-optimal solution for the game, choose the estimate  $\bar{e} \in \mathcal{E}$  with the best expected average over all situations  $s \in \hat{S}$  using the weight function  $\mathbf{w}$ , and use this expected average plus  $x$  as the payoff in  $E$ . If we want a worst-case-optimal solution, choose  $\bar{e}$  as the estimate with the lowest worst-case outcome over all situations in  $\hat{S}$  and set the payoff to the cost in that worst case plus  $x$ .*

We will now again define two very similar algorithms for finding optimal strategies. Algorithm 4.35 will follow standard game theory and is thus proven to calculate optimal strategies, whereas Algorithm 4.36 will be easier to calculate in practice. We will show that these two algorithms produce the same results and thereby prove that also Algorithm 4.36 finds optimal strategies.

**Algorithm 4.35.** *We calculate the expected overall costs by recursively calculating the expected costs of each node.*

*For each estimate leaf  $E$ , calculate the payoffs as described in Algorithm 4.34. The expected costs equal the payoff.*

*For each player node  $N$ , choose the child with the lowest expected costs. (Since we do not have any information sets spanning nodes, it is sufficient to look at one player node at a time.) Set the expected costs of  $N$  to the expected costs of that child.*

*For computer nodes  $C$ , calculate the expected costs of all children. If calculating an average-optimal solution, return the weighted average of these costs, using the weight function  $\mathbf{w}$ . If calculating a worst-case-optimal solution, return the highest expected costs among the children.*

Note that in contrast to Algorithm 4.18, we now have to think about the difference between average-optimal and worst-case-optimal strategies in the computer nodes instead of the player nodes. Also note that this algorithm is easier to describe (and calculate) than Algorithm 4.18.

Now we will again define a second algorithm that, like Algorithm 4.19, moves the costs to the edges instead of the leafs:

**Algorithm 4.36.** *As in Algorithm 4.19, remove the payoffs that were calculated for the estimate leafs. We instead label the edges with costs in the following way:*

- *For each edge leading from a player node to an estimate leaf, calculate the expected payoff for the estimate the same way as in Algorithm 4.34, but do not add the cost  $x$  of the questions asked until that point.*
- *For each edge leading from a computer node to a player node, label the edge with the cost corresponding to the answer.*
- *Label all other edges with 0.*

(See Figure 4.8.)

*Also like in Algorithm 4.19, we will now calculate expected future costs for each node.*

*For each estimate leaf  $E$ , set the expected future cost to 0.*

*Let  $N$  be a player node for which we want to calculate the expected future cost. For each child, calculate the sum of that child's expected future cost plus the cost of the edge leading to the child. Choose the child for which this sum is smallest. This sum is the expected future cost of  $N$ .*

*For computer nodes  $C$ , again calculate for each child the sum of that child's expected future cost plus the cost of the edge leading to it. If looking for an average-optimal solution, calculate the weighted average of these costs, using the weight function  $\hat{\mathbf{w}}$  from Algorithm 4.27. If looking for a worst-case-optimal solution, use the maximum of these costs.*

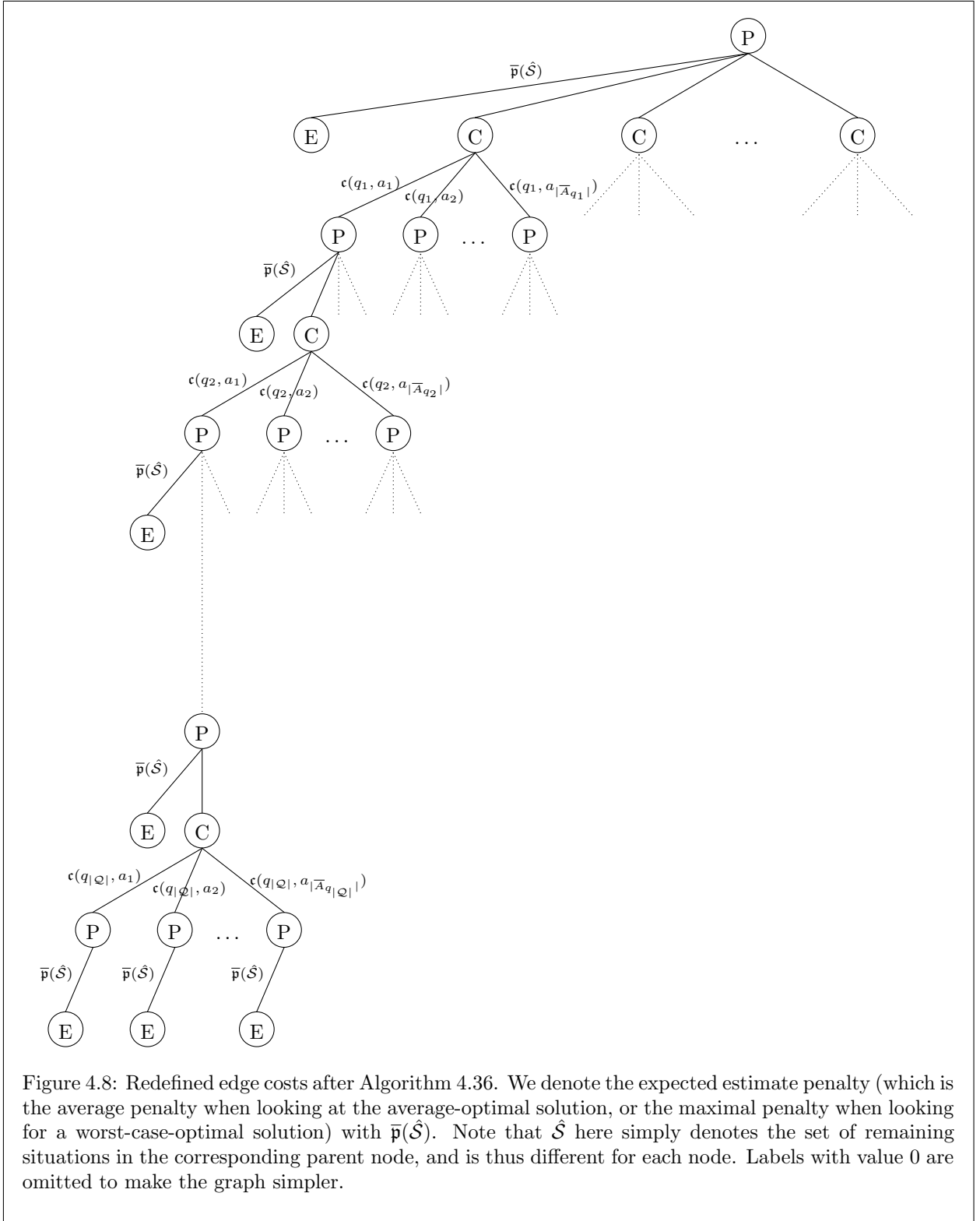


Figure 4.8: Redefined edge costs after Algorithm 4.36. We denote the expected estimate penalty (which is the average penalty when looking at the average-optimal solution, or the maximal penalty when looking for a worst-case-optimal solution) with  $\bar{p}(\hat{\mathcal{S}})$ . Note that  $\hat{\mathcal{S}}$  here simply denotes the set of remaining situations in the corresponding parent node, and is thus different for each node. Labels with value 0 are omitted to make the graph simpler.

As announced earlier, we will show that these two algorithms are equivalent.

**Theorem 4.37.** *Algorithms 4.35 and 4.36 will result in the same strategy and the same expected outcome in the root node.*

*Proof.* We use the same proof idea as in the proof for Theorem 4.20, and will see that this tree structure makes the first part of the proof much easier, but the second part harder. Let again  $c(A)$  be the expected total costs,  $f(A)$  the expected future costs and  $p(A)$  the past costs in a node  $A$ , and let  $x(A, B)$  be the cost of the edge from  $A$  to  $B$ . We once more show that for each node,  $f(A) + p(A) = c(A)$ .

For the estimate leafs, this is again obvious from the definitions.

Let  $N$  be a player node for which all children satisfy the condition. Algorithm 4.35 compares the costs  $c(C)$  of the children  $C \in \text{children}(N)$ . Algorithm 4.36 on the other hand compares  $f(C) + x(N, C)$ . Using the equality  $x(N, C) = p(C) - p(N)$ , we again get:

$$\begin{aligned} f(C) + x(N, C) &= f(C) + p(C) - p(N) \\ &= c(C) - p(N) \end{aligned}$$

Once again, the compared costs only differ by a constant offset, so the same child is chosen.

Let  $\tilde{C}$  denote that chosen child. Then Algorithm 4.35 gives  $c(N) = c(\tilde{C})$ , while Algorithm 4.36 gives  $f(N) = f(\tilde{C}) + x(N, \tilde{C})$ . Using  $p(N) = p(\tilde{C}) - x(N, \tilde{C})$ , we get

$$\begin{aligned} f(N) + p(N) &= f(\tilde{C}) + x(N, \tilde{C}) + p(\tilde{C}) - x(N, \tilde{C}) \\ &= f(\tilde{C}) + p(\tilde{C}) \\ &= c(\tilde{C}) \\ &= c(N) , \end{aligned}$$

the desired equality.

Let  $A$  be a computer node for which all children satisfy the condition. Algorithm 4.35 takes the average or maximum of  $c(C)$  of all children  $C \in \text{children}(A)$ , whereas Algorithm 4.36 takes the average or maximum of  $f(C) + x(A, C)$ . Again we get

$$\begin{aligned} f(C) + x(A, C) &= f(C) + p(C) - p(A) \\ &= c(C) - p(A) . \end{aligned}$$

These values are again offset by the constant  $p(A)$ . If we take the maximum, the same node  $\bar{C}$  is therefore chosen. Therefore  $c(A) = c(\bar{C})$  from Algorithm 4.35,  $f(A) = f(\bar{C}) + x(A, \bar{C})$  from Algorithm 4.36,  $p(A) = p(\bar{C}) - x(A, \bar{C})$  from the definition of  $p(X)$ , and thus

$$\begin{aligned} f(A) + p(A) &= f(\bar{C}) + x(A, \bar{C}) + p(\bar{C}) - x(A, \bar{C}) \\ &= f(\bar{C}) + p(\bar{C}) \\ &= c(\bar{C}) \\ &= c(A) \end{aligned}$$

shows the desired equality.

If on the other hand we have to calculate the weighted average, then let  $q$  be the question that led to  $A$  and let  $C(a)$  denote the child of  $A$  that corresponds to answer  $a$ . With these definitions, Algorithm 4.35 leads to

$$c(A) = \frac{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a \cdot c(C(a))}{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a} ,$$

whereas using

$$\begin{aligned} f(C(a)) + x(A, C(a)) &= f(C(a)) + p(C(a)) - p(A) \\ &= c(C(a)) - p(A) , \end{aligned}$$

Algorithm 4.36 leads to

$$\begin{aligned}
f(A) &= \frac{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a \cdot (f(C(a)) + x(A, C(a)))}{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a} \\
&= \frac{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a \cdot (c(C(a)) - p(A))}{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a} \\
&= \frac{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a \cdot c(C(a))}{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a} - p(A) \cdot \frac{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a}{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a} \\
&= \frac{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a \cdot c(C(a))}{\sum_{a \in \bar{\mathcal{A}}_q} \hat{w}_a} - p(A) \\
&= c(A) - p(A),
\end{aligned}$$

again proving the desired equality.  $\square$

### 4.4.3 Decision sets

The obvious but trivial choice for decision sets are once again the player nodes (which in this case already are the information sets). Given that now the player nodes already contain the information about the remaining possible situations, the following theorem comes naturally:

**Theorem 4.38.** *Let  $A$  and  $B$  be two player nodes in which the same situations are still possible. Then the same move is optimal in both of them (or is at least among the optimal moves if more than one exists).*

*Proof.* The proof is almost analogue to the one of Theorem 4.24. First, we again see that any question that has already been asked in one node, but is still available in the other, does not give any new information but might potentially cause costs. Therefore we ignore such questions. Once we do that, the remaining subtrees are again identical.  $\square$

We also get almost the same two conclusions as before:

**Corollary 4.39.** *Let  $\hat{S} \subseteq S$  be a subset of the possible situations. Then all player nodes in which the situations in  $\hat{S}$  are exactly the situations that are still considered possible can be combined into one decision set.*

*Proof.* This follows directly from the previous considerations.  $\square$

**Corollary 4.40.** *Let  $\hat{S} \subseteq S$  be a set of situations, and let  $q \in Q$  be a question that has the same answer for all situations in  $\hat{S}$ . Then in the decision set corresponding to  $\hat{S}$ , the question  $q$  is not optimal (or is at least not the only optimal question).*

*Proof.* The proof is identical to the one of Corollary 4.26.  $\square$

## 4.5 Decision set graph

As we have seen in Corollary 4.25 and Corollary 4.39, player nodes with the same set of remaining situations can be treated exactly the same way. (In the standard game tree, the set of remaining situation depended on the information set we were in. In the behavioural game tree, the set of remaining situations depended on the situations not ruled out yet on the way there. Lemma 4.33 also showed the connection between those.)

We therefore want to create a new data structure in which all nodes that belong to the same decision set are combined into one single node. We will first describe a way to roughly imagine what is going on. After that, we will formally define this new data structure.

To combine all nodes of one decision set into one node, we simply take the behavioural game tree and merge all player nodes that have the same set of remaining situations. We do have a little problem though: What should we do if we want to merge two player nodes, but some question is not available in one of those nodes any more? Fortunately, Corollary 4.40 has already shown that such a question cannot be optimal (or can at least never be the only optimal question). We will therefore simply omit outgoing edges for such questions.

Consequently, in the merged node we will only have outgoing edges for questions that can still give at least two different answers among the remaining situations: Let  $\hat{S}$  be the set of remaining situations, and let us assume that there exists a question  $q$  that gives the same answer for all situations  $s \in \hat{S}$ . Let  $A$  be any node from the decision set in which  $\hat{S}$  is the set of remaining situations. Either question  $q$  has already been asked on the path to  $A$ , then it is not available there any more, so it is omitted from the merged node. Or it has not been asked yet, then we can ask it and reach a node  $A'$  in which  $\hat{S}$  still is the set of remaining situations, since the answer to  $q$  did not tell us anything new. Consequently,  $A'$  is also merged into the same node, and  $q$  is not available there any more. Therefore,  $q$  is again omitted. For the other direction, we note that if the set of remaining situations still contains situations that can give two different answers for some question  $q$ , then this question certainly has not been asked yet. Therefore, the outgoing edge corresponding to  $q$  still exists in all merged nodes.

The other problem that we face is that the nodes we merged had edges to different computer nodes. However, once we merge them, we cannot differentiate between those any more. Therefore, we have to merge corresponding computer nodes as well.

Similar to the player nodes, we want to merge all computer nodes that have the same set of remaining situations *and* the same question to be answered. Here, all such nodes trivially have the same occurring answers and thus also the same outgoing edges. Again, those computer nodes had edges to different player nodes, which we now cannot differentiate any more. Fortunately for us, all such player nodes were already merged: For a set of computer nodes that were merged, let  $\hat{S}$  be the set of remaining situations in each of these merged computer nodes, let  $q$  be the question to be answered, and let  $a$  be the answer. Then each merged computer node had a child corresponding to answer  $a$ . However, the set of remaining situations in that child was  $\hat{S}((q, a))$  for each of these computer nodes. Therefore, the corresponding player nodes have been merged already in the previous step.

Finally, we also need to merge corresponding estimate leafs. Fortunately, the expected cost of an estimate leaf (or cost of the edge leading to it) only depended on the set of remaining situations anyway.

Since each node from the behavioural game tree is merged into exactly one of these new nodes – we merge all player nodes that have the same set of remaining situations, so each player node can be part of exactly one such group –, we can say that such a node is “represented” or “encompassed” by the node it was merged into.

We will not describe the merging process in much more detail than that, since it is not how we will in practice create the decision set graph. It simply is an abstract concept that helps to understand what the decision set graph is.



Let us now look at the more formal definition:

**Definition 4.41** (decision set graph). *The DECISION SET GRAPH of a deduction game is a directed, connected graph with the following properties:*

- Each node can be categorized as “computer node”, “player node” or “estimate leaf”.
- Each player node is uniquely identified by a set  $\hat{S} \subseteq \mathcal{S}$  of remaining situations.<sup>1</sup>
- Each computer node is uniquely identified by a set  $\hat{S} \subseteq \mathcal{S}$  of remaining situations and a question  $q \in \mathcal{Q}$  to be answered.<sup>2</sup>
- Each estimate leaf is uniquely identified by a set  $\hat{S} \subseteq \mathcal{S}$  of remaining situations.<sup>3</sup>
- We have one player node in which all situations are possible (i.e., the root node of the original game, in which  $\mathcal{S}$  itself is the set of remaining situations). This player node is the only node in the decision set graph that has no incoming edge. We will call this node the root of the decision set graph.
- Each player node has one outgoing edge for every question that will give at least two different answers for the remaining situations (and is thus not useless by Corollary 4.40), as well as one outgoing edge for taking a guess.

Let  $\hat{S} \subseteq \mathcal{S}$  be the set of remaining situations in the player node, and let  $q$  be one question that can still give at least two different answers among the situations in  $\hat{S}$ . Then the outgoing edge from the player node that corresponds to  $q$  leads to the computer node in which  $\hat{S}$  is the set of remaining situations as well, and  $q$  is the question to be answered.

The outgoing edge for the estimate leads to the estimate leaf in which  $\hat{S}$  is the set of remaining situations.

- Each computer node has one outgoing edge for every answer that occurs. Each of these outgoing edges leads to player node corresponding to the subset of the remaining situations that will give this answer.

That is, let  $\hat{S} \subseteq \mathcal{S}$  be the set of remaining situations in the computer node, let  $q$  be the question to be answered, and let  $a_1$  be one answer. Then the outgoing edge corresponding to  $a_1$  leads to the player node in which  $\hat{S}((q, a_1)) = \{s \in \hat{S} \mid \mathfrak{r}(s, q) = a_1\} \subseteq \hat{S}$  is the set of remaining situations.

- Nodes for taking a guess are the only nodes with no outgoing edges. We will refer to them as leafs or estimate leafs.
- There are no nodes in the decision set graph that cannot be reached from the root, and nodes do not have any incoming or outgoing edges other than those described above. (That is, the graph only contains those nodes and edges that resulted from the “merging” described above, and does not arbitrarily add useless new ones.)

(See Figure 4.9 for an example.)

<sup>1</sup>That is, for each subset  $\hat{S} \subseteq \mathcal{S}$ , there exists at most one player node in the decision set graph that has this set of remaining situations.

<sup>2</sup>That is, for each combination of a subset  $\hat{S} \subseteq \mathcal{S}$  and a question  $q \in \mathcal{Q}$ , there exists at most one computer node in the decision set graph that has  $\hat{S}$  as set of remaining situations and  $q$  as the question to be answered.

<sup>3</sup>That is, for each subset  $\hat{S} \subseteq \mathcal{S}$ , there exists at most one estimate leaf in the decision set graph that has this set of remaining situations.

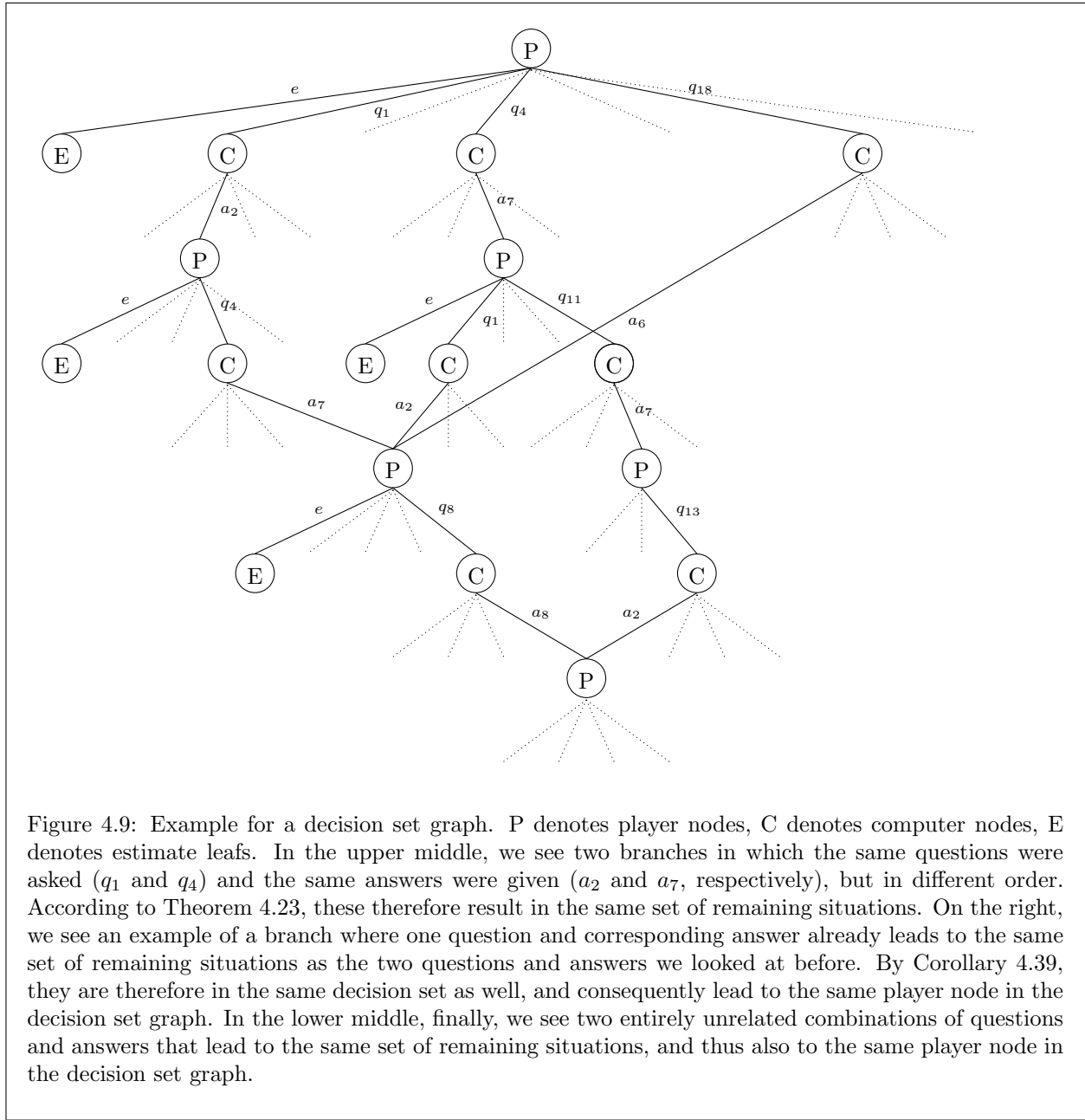


Figure 4.9: Example for a decision set graph. P denotes player nodes, C denotes computer nodes, E denotes estimate leaves. In the upper middle, we see two branches in which the same questions were asked ( $q_1$  and  $q_4$ ) and the same answers were given ( $a_2$  and  $a_7$ , respectively), but in different order. According to Theorem 4.23, these therefore result in the same set of remaining situations. On the right, we see an example of a branch where one question and corresponding answer already leads to the same set of remaining situations as the two questions and answers we looked at before. By Corollary 4.39, they are therefore in the same decision set as well, and consequently lead to the same player node in the decision set graph. In the lower middle, finally, we see two entirely unrelated combinations of questions and answers that lead to the same set of remaining situations, and thus also to the same player node in the decision set graph.

We will start with a number of observations:

**Theorem 4.42.** *On every path from the root to a leaf, player and computer nodes alternate.*

*Proof.* This follows directly from the definition above, since player nodes only lead to computer nodes (and estimate leaves), and computer nodes only lead to player nodes.  $\square$

**Theorem 4.43.** *Each computer node has exactly one incoming edge. Likewise, each estimate leaf has exactly one incoming edge.*

*Proof.* Computer nodes can only have incoming edges from player nodes. By construction, all children of a player node have the same set of remaining situations  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  as the player node itself.

Let us assume that a computer node had two incoming edges, and let  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  be the set of remaining situations in the computer node. Then this also has to be the set of remaining situations in both player

nodes that lead to it. However, different player nodes must by definition have different sets of remaining situations, which contradicts our assumption.

The same argument also holds for estimate leaves.  $\square$

**Theorem 4.44.** *The decision set graph is acyclic (i.e., it does not contain any directed cycles).*

*Proof.* Let us assume the graph contained a directed cycle. Then there would exist an infinite path beginning from the root, which can be obtained by going from the root to a node in the cycle (which is possible according to the definition), and then following the cycle indefinitely.

For any player node  $A$ , let  $|A|$  be the number of remaining situations. For any computer node  $B$ , let  $|B|$  be the number of remaining situations. For both kinds of nodes we call this the SIZE of the node.

Using this definition, every player node has only outgoing edges to computer nodes of the same size. (In addition, they have one outgoing edge leading to a leaf which by definition has no outgoing edges. Therefore it cannot be part of a cycle.)

Computer nodes have outgoing edges for every answer that is still available. Every situation gives exactly one of those answers, therefore the sum of the sizes of all the successors of a computer node equals the size of the computer node. Since players can only choose questions for which at least two answers are still possible, we have at least two children. Each remaining situation is contained in exactly one of the children (and each child has to contain at least one situation). Therefore, each of the children of the computer node has a size smaller than its parent.

Consequently, every outgoing edge only leads to nodes with smaller or equal size, and whenever we take two steps along outgoing edges from any node, we get to a node with a strictly smaller size. Since the size of the root is finite and all sizes are positive integers, every path from the root has only finitely many steps. Therefore, the graph cannot contain any cycles.  $\square$

Note that we needed to forbid pointless questions in order to avoid cycles.

We can now extend our definitions of children, parents and leaves in the decision set graph:

**Definition 4.45.** *Let  $E$  be a node in the decision set graph with no outgoing edges, then we call  $E$  a leaf. Estimate nodes are the only nodes with this property.*

*Let  $N$  be a node in the decision set graph. All nodes to which  $N$  has outgoing edges we call children of  $N$ , and use  $\text{children}(N)$  to denote them.*

*Let  $N$  be a node in the decision set graph that has exactly one incoming edge (and thus is child to exactly one other node). Then we call the node leading to  $N$  the parent of  $N$  and use  $\text{parent}(N)$  to denote it. All computer nodes have (unique) parents.*

So far, we only looked at the decision set graph by itself. However, we know that it is closely related to the behavioural game tree. As a next step, we will formally define how nodes in the behavioural game tree are represented by nodes in the decision set graph:

**Definition 4.46** (encompassing). *For each node  $A$  in the behavioural game tree, we define a node  $A'$  in the decision set graph that ENCOMPASSES  $A$ :*

- *Let  $N$  be a player node of the behavioural game tree. Let  $\hat{S} \subseteq S$  be the set of remaining situations in  $N$ . By our construction, there exists exactly one player node  $N'$  in the decision set graph corresponding to  $\hat{S}$ . We call  $N'$  the node in the decision set graph that encompasses  $N$  and write  $N' = \text{encompassing}(N)$ .*
- *Let  $E$  be an estimate leaf in the behavioural game tree. Let again  $\hat{S} \subseteq S$  be the set of remaining situations in  $E$ . Then there exists exactly one estimate leaf  $E'$  in the decision set graph that corresponds to  $\hat{S}$ . We call  $E'$  the node in the decision set graph that encompasses  $E$  and write  $E' = \text{encompassing}(E)$ .*
- *Let  $C$  be a computer node in the behavioural game tree that has at least two children. Let  $\hat{S} \subseteq S$  be the set of remaining situations in  $C$ , and let  $q \in \mathcal{Q}$  be the question that is to be answered by the computer in  $C$ . Then there exists exactly one computer node  $C'$  in the decision set graph in which also  $\hat{S}$  is the set of remaining situations and the asked question is  $q$ . We call  $C'$  the node in the decision set graph that encompasses  $C$ , and write  $C' = \text{encompassing}(C)$ .*

Note that for each node  $A$  in the behavioural game tree the encompassing node  $A'$  is unique, whereas every node  $B'$  in the decision set graph can encompass any number of nodes from the behavioural game tree.

Even though we already wrote above that each node in the behavioural game tree (with the exception of computer nodes that have only a single child) is encompassed by a node in the decision set graph “by construction”, let us formally prove this in case we still have doubts. Until proven that an encompassing node always exists, let us consider  $\text{encompassing}(A)$  to be undefined if  $A$  does not have an encompassing node.

First of all we prove in the following three lemmata that our definition of encompassing behaves on the graph structures the way we expect it to:

**Lemma 4.47.** *Let  $N$  be a player node in the behavioural game tree and  $N' = \text{encompassing}(N)$  its encompassing node in the decision set graph. Let  $q$  be a question denoting one outgoing edge of  $N$ , and let  $C(q)$  denote the child that is reached by asking that question.*

*If  $N'$  has an outgoing edge corresponding to  $q$ , then let  $C'(q)$  be the child of  $N'$  that is reached by asking the same question  $q$ . Then  $C(q)$  is encompassed by  $C'(q)$ .*

*Otherwise, if  $N'$  has no outgoing edge corresponding to  $q$ , then the computer node  $C(q)$  has exactly one outgoing edge. Let  $N(q)$  be the only child of that computer node, then  $N'$  itself encompasses  $N(q)$  again.*

*Proof.* Let  $\hat{S} \subseteq S$  be the set of remaining situations in  $N$  (and consequently the set of remaining situations in  $N'$ ). Then in  $C(q)$  the set of remaining situations is  $\hat{S}$  as well (since a computer node has the same set of remaining situations as its parent), and the asked question is  $q$ . Likewise  $\hat{S}$  is the set of remaining situations in  $C'(q)$ , and question  $q$  was asked. Therefore by Definition 4.46,  $C'(q)$  encompasses  $C(q)$ .

That  $N'$  does not have an outgoing edge for a question can only be the case if this question has the same answer for all situations in  $\hat{S}$ . Consequently, the computer node  $C(q)$  has to give the same answer for all situations in  $\hat{S}$ , so it has only one child. This child  $N(q)$  has still the same set of remaining situations  $\hat{S}$ . Therefore,  $N(q)$  is also encompassed by  $N'$ .  $\square$

**Lemma 4.48.** *Let  $N$  be a player node in the behavioural game tree and  $N' = \text{encompassing}(N)$  the encompassing node in the decision set graph. Let  $E$  be the estimate leaf with  $N$  as parent, and let  $E'$  be the estimate leaf in the decision set graph with  $N'$  as its parent. Then  $E$  is encompassed by  $E'$ .*

*Proof.* Let  $\hat{S} \subseteq S$  be the set of remaining situations in  $N$  (and consequently the set of remaining situations in  $N'$ ). Then  $\hat{S}$  is also the set of remaining situations in both  $E$  and  $E'$ , and thus  $E$  is encompassed by  $E'$  by definition.  $\square$

**Lemma 4.49.** *Let  $C$  be a player node in the behavioural game tree and  $C'$  the encompassing node in the decision set graph, and let  $q$  be the question that is to be answered in both of them. Let  $\hat{S}$  be the set of remaining situations, and let  $a_1$  be an answer that occurs among them. Let  $N(a_1)$  be the child of  $C$  that is reached with answer  $a_1$ , and let  $N'(a_1)$  be the child of  $C'$  reached with the same answer. Then  $N(a_1)$  is encompassed by  $N'(a_1)$ .*

*Proof.* Let  $\hat{S}((q, a_1)) \subseteq \hat{S}$  be the set of situations in  $\hat{S}$  for which  $a_1$  is the answer to  $q$ , i.e.,  $\hat{S}((q, a_1)) := \{s \in \hat{S} \mid r(s, q) = a_1\}$ . Then  $\hat{S}((q, a_1))$  is the set of remaining situations both in  $N(a_1)$  and  $N'(a_1)$ . Therefore by Definition 4.46,  $N'(a_1)$  encompasses  $N(a_1)$ .  $\square$

Using these three lemmata that show how encompassment is passed on to the children of a node, we can now formally prove that indeed (almost) every node is encompassed:

**Theorem 4.50.** *Every player node, every computer node with at least two possible answers, and every estimate leaf in the behavioural game tree is encompassed by a node in the decision set graph. Vice versa, each node in the decision set graph encompasses at least one node in the behavioural game tree.*

*In short, player nodes in the decision set graph exist if and only if they are needed to encompass a node from the behavioural game tree.*

*Proof.* The root of the behavioural game tree contains the entire set  $\mathcal{S}$  as set of remaining situations. So does the root of the decision set graph. Therefore, the root of the behavioural game tree is encompassed by the root of the decision set graph.

We will show that for each node that is encompassed, its children are encompassed as well, with the exception of computer nodes that have only one possible answer. For those computer nodes we will have to show in addition that even though they themselves are not encompassed, their children are.

Let  $N$  be a player node that is encompassed by  $N'$ . By Lemma 4.47, we know that each computer node that is child of  $N$  and has at least two answers is encompassed. By the same lemma, for each computer node that has only one child, this child is encompassed as well.

By Lemma 4.48, the estimate leaf that is child of  $N$  is encompassed as well. Finally, by Lemma 4.49, all children of an encompassed computer node are encompassed as well.

Starting from the root and going downward, we can therefore show that all nodes in the behavioural game tree are encompassed, skipping only computer nodes with a single child (but encompassing that child).

Vice versa, let us assume that there existed a player node  $A$  in the decision set graph that is not encompassing any node from the behavioural game tree. Let  $((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$  be one possible sequence of questions and answers that leads to  $A$  when starting from the root of the decision set graph. Starting in the root of the behavioural game tree and following the same sequence of questions and answers, we get to a player node that is encompassed by  $A$  (as can be easily seen by applying Lemmata 4.47 and 4.49 in each step).  $\square$

For convenience in later proofs, we will show one more useful property of the encompassing relationship:

**Definition 4.51** (encompassing set). *Let  $\hat{A}$  be a set of nodes in the behavioural game tree, then we define an ENCOMPASSING SET of nodes  $\hat{A}'$  in the decision set graph as the set containing exactly the nodes that encompass the nodes in  $\hat{A}$ . We write again*

$$\hat{A}' = \text{encompassing}(\hat{A}) = \bigcup_{a \in \hat{A}} \{\text{encompassing}(a)\} .$$

This trivially gives us the following equality:

**Theorem 4.52.** *Let  $\hat{A}$  and  $\hat{B}$  be sets of nodes in the behavioural game tree, then*

$$\text{encompassing}(\hat{A} \cup \hat{B}) = \text{encompassing}(\hat{A}) \cup \text{encompassing}(\hat{B}) .$$

*Proof.*

$$\begin{aligned} \text{encompassing}(\hat{A} \cup \hat{B}) &= \bigcup_{a \in \hat{A} \cup \hat{B}} \{\text{encompassing}(a)\} \\ &= \bigcup_{a \in \hat{A}} \{\text{encompassing}(a)\} \cup \bigcup_{a \in \hat{B}} \{\text{encompassing}(a)\} \\ &= \text{encompassing}(\hat{A}) \cup \text{encompassing}(\hat{B}) \end{aligned}$$

$\square$

A final observation, using these definitions:

**Lemma 4.53.** *Let  $N$  be the set of all player nodes in the behavioural game tree, then  $\text{encompassing}(N)$  is the set of all player nodes in the decision set graph.*

*Proof.* This follows directly from Theorem 4.50.  $\square$

### 4.5.1 Solving the game on the decision set graph

We will first describe an algorithm to find “best moves” in the decision set graph, will then show how to transform the resulting strategy back into the behavioural tree, and prove that it is optimal there. (Since the decision set graph is not an actual game tree, we have not really defined what “best” moves mean. We therefore simply define the moves calculated by the following algorithm as “best moves”.)

Since this is the central algorithm of this thesis, we will give it in two forms, the first with a little more text, and then again immediately afterwards with some more formulas.

**Algorithm 4.54** (solving the game on the decision set graph).

Label the edges with costs as follows:

- For each edge leading to an estimate leaf, calculate the expected cost for the guessing using Algorithm 4.34, but do not add the cost  $x$  of the questions asked until that point.
- For each edge leading from a computer to a player node, label the edge with the cost corresponding to the answer.
- Set all remaining edges to cost 0.

We will now again calculate expected future costs for all nodes:

- For estimate leafs, set the expected future costs to 0.
- For a player node  $N$ , calculate the expected future costs as follows:
  - For each child, calculate the sum of expected future costs of that child and the cost of the edge leading to it.
  - Choose the child for which this sum is lowest. This sum is the expected future cost for node  $N$ . Remember the move corresponding to that child.
- For computer nodes  $C$ , the expected future costs are calculated as follows:
  - For each child, calculate the sum of expected future costs of that child and the cost of the edge leading to it.
  - If looking for average-optimal solutions, calculate the weighted average of these costs of all children using the weight function from Algorithm 4.27. This is the expected future cost for node  $C$ .
  - If looking for worst-case-optimal solutions, find the child for which this sum is highest, and set the expected outcome for  $C$ .

Using these functions, we now simply calculate the expected future costs of the root node. All other nodes will be calculated recursively in the process. The algorithm is finite, since the decision set graph is acyclic and all functions only refer to children of the node they are currently looking at.

(This can of course be optimized using dynamic programming and other tricks. We will have a closer look at those once we start actually programming it in Chapter 6.)

The above is a rather prosaic version of the algorithm. We can also define it in a more mathematical way, which will come in handy later:

**Algorithm 4.55** (solving the game on the decision set graph).

Label the edges with costs as follows:

- Let  $E$  be an estimate leaf, and let  $\hat{\mathcal{S}}$  be the set of remaining situations in  $E$ .  
If looking for an average-optimal solution, then choose an estimate  $\bar{e} \in \mathcal{E}$  such that

$$\hat{\mathbf{p}}(e) = \frac{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot \mathbf{p}(s, e)}{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)}$$

is minimal. Label the edge from  $\text{parent}(E)$  to  $E$  with cost  $\hat{\mathbf{p}}(\bar{e})$ .

If looking for a worst-case-optimal solution, choose an estimate  $\bar{e}' \in \mathcal{E}$  such that

$$\hat{\mathbf{p}}'(e) = \max(\mathbf{p}(s, e))$$

is minimal. Label the edge from  $\text{parent}(E)$  to  $E$  with cost  $\hat{\mathbf{p}}'(\bar{e}')$ .

Do this for all estimate leafs.

- Let  $C$  be a computer node, and let  $q$  be the question that is to be answered in that node. Let  $A$  be a child of  $C$ , and let  $a$  be the answer that leads from  $C$  to  $A$ . Label the edge between  $C$  and  $A$  with cost  $\mathfrak{c}(q, a)$ .

Do this for all outgoing edges of all computer nodes.

- Set all remaining edges to cost 0.

Let  $x(X, Y)$  denote the cost of the edge from node  $X$  to node  $Y$  (if such an edge exists). Let  $\hat{\mathfrak{w}}(\hat{\mathcal{S}}, q, a)$  denote the weight as defined in Algorithm 4.27, that is,

$$\hat{\mathfrak{w}}(\hat{\mathcal{S}}, q, a) := \sum_{\substack{s \in \hat{\mathcal{S}} \\ \mathfrak{r}(s, q) = a}} \mathfrak{w}(s) .$$

We will use  $f(N)$  to denote the future cost of a node  $N$ , and calculate  $f(N)$  as follows:

- If  $N$  is an estimate leaf, then

$$f(N) := 0 .$$

- If  $N$  is a player node, then

$$f(N) := \min_{C \in \text{children}(N)} (f(C) + x(N, C)) .$$

- Let  $N$  be a computer node. Let  $\hat{\mathcal{S}}$  be the set of remaining situations in  $N$ , and let  $q$  be the question to be answered. For each possible answer  $a \in \mathcal{A}_q$ , let  $C(a)$  be the child of  $N$  that corresponds to  $a$ .

- If we are looking for an average-optimal solution, then

$$f(N) := \frac{\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}, q, a) \cdot (f(C(a)) + x(N, C(a)))}{\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}, q, a)} .$$

- If we are looking for a worst-case-optimal solution, then

$$f(N) := \max_{a \in \mathcal{A}_q} (f(C(a)) + x(N, C(a))) .$$

Since the decision set is acyclic and each definition of  $f(N)$  only refers to nodes that are children of  $N$ , the function  $f(N)$  is well-defined. The expected future cost in the root  $R$  therefore simply is  $f(R)$ .

We now transform this strategy back into one on the behavioural game tree. To do so, we simply introduce an algorithm to determine the move in a player node of the behavioural game tree, based on the node in the decision set graph that the player node is encompassed by.

**Algorithm 4.56.** Create the decision set graph and solve it using Algorithm 4.54.

Let  $N$  be a player node of the behavioural game tree for which we want to know the best move. Choose the question that was chosen in the encompassing node  $N' = \text{encompassing}(N)$ .

Let  $E$  be an estimate leaf in the behavioural game tree. Choose the estimate that was chosen in the encompassing node  $E' = \text{encompassing}(E)$ .

In order to prove that this indeed gives an optimal strategy, as a next step we modify this algorithm to also store the associated costs  $f'(\cdot)$  that were calculated by Algorithm 4.55:

**Algorithm 4.57.** Let  $N$  be a player node of the behavioural game tree and  $N'$  the encompassing node in the decision set graph. Choose the question that was chosen in  $N'$  using Algorithm 4.54, and set  $f'(N)$  to the expected future costs in  $N'$ .

Let  $E$  be an estimate node in the behavioural game tree and  $E'$  the encompassing leaf in the decision set graph. Choose the estimate that was chosen there, and set  $f'(E)$  to 0.

Let  $C$  be a computer node in the behavioural game tree and  $C'$  the encompassing computer node in the decision set graph. Set  $f'(C)$  to the expected future costs in  $C'$ .

Now we are ready to prove that Algorithm 4.56 actually gives an optimal strategy:

**Theorem 4.58.** *Algorithm 4.56 gives an optimal strategy for the player, and the expected future costs calculated for the root of the decision set graph equal the expected future cost of the root of the behavioural game tree. (If the decision set graph was solved for average-optimal solutions, this gives an average-optimal strategy of the deduction game. If the decision set graph was solved for worst-case-optimal solutions, it gives a worst-case-optimal strategy.)*

*Proof.* We will show that this algorithm simulates Algorithm 4.36. This time we start our proof from the leafs and go upwards. We will show that in each node  $X$  of the behavioural game tree, the expected future costs  $f(X)$  and  $f'(X)$  are the same.

For each estimate leaf  $E$  it holds by construction that  $f(E) = f'(E) = 0$ , and that the same estimate is chosen.

Let  $N$  be a player node so that for each child of  $N$  the condition holds. Since the encompassing node  $N'$  in the decision set graph only has outgoing edges for questions for which at least two different answers are still possible among the remaining situations, we first have to show that other questions can not be optimal. This however follows directly from Corollary 4.40 whereby questions that have the same answer for all remaining situations do not need to be considered.

Among the remaining questions, each question  $q$  leads to one computer node  $C(q)$ , and as we have shown in Lemma 4.47, this node  $C(q)$  is encompassed by the child  $C'(q)$  of  $N'$  in the decision set graph. Both in the behavioural game tree and the decision set graph the edges between  $N$  and  $C(q)$  and between  $N'$  and  $C'(q)$ , respectively, have cost 0. In addition, there is one outgoing edge leading from  $N$  to an estimate leaf  $E$ , and an edge leading from  $N'$  to an estimate leaf  $E'$  that encompasses  $E$  as we have shown in Lemma 4.48. By our construction, the costs of these edges and nodes are equal as well.

In summary we therefore have:

- edges from  $N$  to computer nodes that by Corollary 4.40 are not optimal and thus are not chosen by Algorithm 4.36;
- edges from  $N$  to computer nodes that correspond to edges from  $N'$  to computer nodes, having the same costs for the edges and the same expected future costs in the child nodes; and
- one edge from  $N$  to an estimate leaf that corresponds to one edge from  $N'$  to an estimate leaf, again having the same costs for the edge and the child node.

Since both Algorithms 4.36 and 4.54 choose the cheapest child in terms of edge cost plus expected future costs of the child, the same move is chosen.

Let  $A$  be the chosen child (which can be either a computer node or an estimate leaf), and  $A'$  the encompassing node belonging to it. Algorithm 4.36 calculates  $f(N)$  as the sum of  $f(A)$  and the cost of the edge between  $N$  and  $A$ . Algorithm 4.56, on the other hand, sets  $f'(N)$  to  $f(N')$ . Looking back at Algorithm 4.54 though, we see that  $f(N')$  equals  $f(A')$  plus the cost of the edge between  $N'$  and  $A'$ , which as we saw above is equal to  $f(A)$  and the cost of the edge between  $N$  and  $A$ , respectively. Therefore,  $f(N) = f'(N)$ .

Let finally  $C$  be a computer node so that for each child of  $C$  the desired equality holds, and let  $C'$  be the encompassing computer node in the decision set graph. Both  $C$  and  $C'$  have one outgoing edge per answer that occurs among the remaining situations, and for each such answer  $a$ , the child  $N(a)$  of  $C$  is encompassed by the child  $N'(a)$  of  $C'$ . Also, the costs of the edges to those children are equivalent again, as are the used weights. Combining this we again get that  $f(C) = f'(C)$ .  $\square$

For the remainder of this thesis, we will therefore now concentrate on finding efficient ways of calculating these “best moves” in the decision set graph, and finding good upper bounds for the number of nodes in the decision set graph.



## Chapter 5

# Upper bounds for the number of nodes in the decision set graph

In the worst case, we have to calculate and store the expected future costs of all nodes in the decision set graph. The number of such nodes therefore is an interesting upper bound for the calculation effort and memory consumption.

For this entire section, let  $n$  denote the number of player nodes in the decision set graph and  $c$  the number of computer nodes. Let  $q := |\mathcal{Q}|$ ,  $e := |\mathcal{E}|$ ,  $s := |\mathcal{S}|$ , for each question  $q \in \mathcal{Q}$  let  $a_q = |\overline{\mathcal{A}}_q|$ , and let  $a := \max_{q \in \mathcal{Q}} |\overline{\mathcal{A}}_q| = \max_{q \in \mathcal{Q}} a_q$ .

### 5.1 Relationship between number of player, computer and estimate nodes

**Theorem 5.1.** *Let  $n$  be the number of player nodes in the decision set graph. Then there are also  $n$  estimate nodes.*

*Proof.* This follows directly from the construction. □

**Theorem 5.2.** *Let  $n$  be the number of player nodes in the decision set graph. Then there are at most  $n \cdot q$  computer nodes.*

*Proof.* Each computer node is child to exactly one player node, and each player node has at most  $q$  computer nodes as children, i.e., at most one for each question. □

**Corollary 5.3.** *Let  $c$  be the number of computer nodes in the decision set graph. Then there are at least  $\frac{c}{q}$  player nodes.*

*Proof.* This follows directly from the previous theorem. □

**Theorem 5.4.** *Let  $c$  be the number of computer nodes in the decision set graph. Then there are at most  $c \cdot a + 1$  player nodes.*

*Proof.* Each player node except for the root node is child to at least one computer node, and each computer node has at most  $a$  children, i.e., at most one for each possible answer. □

**Corollary 5.5.** *Let  $n$  be the number of player nodes in the decision set graph. Then there are at least  $\frac{n-1}{a}$  computer nodes.*

*Proof.* This follows directly from the previous theorem. □

**Corollary 5.6.** *Let  $n$  be the number of player nodes and  $c$  the number of computer nodes in the decision set graph, then  $c = O(nq)$ .*

*If we consider  $q$  and  $a$  to be constants, then  $n = \Theta(c)$  (and  $c = \Theta(n)$ ).*

*Proof.* This follows directly from the theorems above, since  $\frac{c}{q} \leq n \leq c \cdot a + 1$ , and  $\frac{n-1}{a} \leq c \leq n \cdot q$ . □

## 5.2 Relationship between number of nodes and calculation effort

**Theorem 5.7.** *For calculating the expected future costs of each node in the decision set graph, the calculation effort of Algorithm 4.54 is composed of the sum of the following:*

- $O(n \cdot q)$  for calculating player nodes;
- $O(c \cdot (a + s))$  for the computer nodes in an average-optimal solution; or
- $O(c \cdot a)$  for the computer nodes in a worst-case-optimal solution; and
- $O(n \cdot e \cdot s)$  for the estimate leafs.

*(This is under the assumption that we store calculated future costs of nodes rather than recursively recalculating them every time. This takes  $O(nf)$  memory space, where  $f$  is the space needed for storing the expected future costs of one node.)*

*Proof.* The costs of the recursive calculation of the children is already accounted for, since by definition we calculate each node exactly once. We therefore only need to count the other calculation costs.

For each of the  $n$  player nodes, we have to retrieve the costs of at most  $q + 1$  children (at an effort of  $O(1)$  per child) and find their minimum (taking  $O(q + 1)$  time), leading to a total time of  $O(q + 1) = O(q)$ ; and thus an effort of  $O(n \cdot q)$  in sum over all player nodes.

For each of the  $c$  computer nodes, we have to retrieve the costs of at most  $a$  children (at an effort of  $O(1)$  per child). For a worst-case-optimal solution we have to find their maximum (taking  $O(a)$  time). For an average-optimal solution we have to calculate the weights using Algorithm 4.27, which can be implemented in such a way that it takes  $O(1)$  time per situation still considered possible. There are at most  $s$  situations still possible, leading to a calculation effort of  $O(s)$ . In addition we need to calculate the weighted average, taking  $O(a)$  time. Altogether, this therefore takes  $O(a)$  time for a worst-case-optimal and  $O(a + s)$  time for an average-optimal solution for each of the  $c$  computer nodes.

Finally we have the same number of estimate leafs as we have player nodes. There might be better algorithms available, but in the worst case, we have to compare each estimate with each situations and find the estimate that gives the best average (or maximum), leading to  $e \cdot s$  comparisons.  $\square$

Since by Corollary 5.6 the number of player and computer nodes are in the same order of magnitude, it makes sense to combine those costs as follows:

**Corollary 5.8.** *The calculation effort for calculating the expected future costs of each node in the decision set graph is  $O(n \cdot (q + qa + qs + es))$ .*

*Proof.* This follows directly from the previous theorem and Corollary 5.6, using  $c = O(nq)$ .  $\square$

First, note that the calculation of the estimates gives a rather big contribution to those costs. For many games, it is possible to find more efficient algorithms for finding best estimates, some being as quick as  $O(1)$  (instead of  $O(es)$ ).

Second, keep in mind that, as we will soon see, the number of player nodes  $n$  will usually be orders of magnitude larger than the other factor,  $(q + qa + qs + es)$ .

## 5.3 Upper bounds for the number of player nodes

Since the number of player nodes is therefore important for estimating the calculation effort, we will now try to find good upper bounds for it.

Since we have more than one dimension to consider –  $s$ ,  $q$ ,  $a$  and the individual values of  $|\overline{A}_{q_i}|$  are all variable –, we will find several different bounds with different combinations of these dimensions. For example, if we have a very large number of situations, but only one question, the upper bound of  $2^s - 1$  (which we will show below) is high above the actual number of player nodes, whereas the upper bound of  $2^q s$  (which will also be shown) can be strict. If on the other hand we have hundreds of questions but only two situations,  $2^s - 1 = 3$  is a potentially strict bound, whereas  $2^q s$  is much higher.

### 5.3.1 Upper bound: $n \leq 2^s - 1$

**Theorem 5.9.**  $n \leq 2^s - 1$ .

*Proof.* Remember that we have exactly one player node for each set of remaining situations (i.e., subset of  $\mathcal{S}$ ) that occurs in at least one player node of the behavioural game tree (or one information set of the standard game tree). There are  $s$  situations, thus there are at most  $2^s$  different subsets of them. We can subtract one subset from that since the empty set does not occur.  $\square$

**Theorem 5.10.** *For each  $s$  there exists a game with  $s$  situations,  $2^s - 1$  questions and  $2^s - 1$  nodes in the decision set graph.*

*Proof.* Let  $\mathcal{S} = \{s_1, s_2, \dots, s_s\}$  be a set containing  $s$  situations. We define questions as follows: For each non-empty subset  $\hat{\mathcal{S}} \subseteq \mathcal{S}$ , let  $q(\hat{\mathcal{S}})$  be a question with

$$\tau(s, q(\hat{\mathcal{S}})) = \begin{cases} \text{“true”} & s \in \hat{\mathcal{S}} \\ \text{“false”} & \text{otherwise} \end{cases}$$

as the evaluation function. (In short, we take any subset of the situations and ask whether  $s$  is in that subset.)

Each question is answered with “true” for at least one situation  $s$ . Thus, after asking question  $q(\hat{\mathcal{S}})$  and getting “true” as an answer, exactly the situations in  $\hat{\mathcal{S}}$  are still possible. Since we have one such question for each non-empty subset  $\hat{\mathcal{S}} \subseteq \mathcal{S}$ , all such subsets occur as sets of remaining questions.  $\square$

**Theorem 5.11.** *For each  $s$  there exists a game with  $s$  situations,  $s$  questions and  $2^s - 1$  nodes in the decision set graph.*

*Proof.* Let  $\mathcal{S} = \{s_1, s_2, \dots, s_s\}$  be a set containing  $s$  situations. We define questions as follows: For each situation  $\bar{s} \in \mathcal{S}$ , let  $q(\bar{s})$  be a question with

$$\tau(s, q(\bar{s})) = \begin{cases} \text{“true”} & s = \bar{s} \\ \text{“false”} & \text{otherwise} \end{cases}$$

as evaluation function. (This means that we take any situation and ask if it is the correct one.)

We need to show that each subset  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  can occur as set of remaining situations. Let  $\hat{\mathcal{S}}$  be one such subset. We will show that we can reach a player node having  $\hat{\mathcal{S}}$  as remaining situations by asking all questions *not* belonging to situations in  $\hat{\mathcal{S}}$  and getting “no” as answer every time. That is, we simply rule out all situations that are not in  $\hat{\mathcal{S}}$ .

For demonstrating that a certain question can still be asked, we will need to show that both answers “yes” and “no” are still possible for it. We choose any situation  $\hat{s} \in \hat{\mathcal{S}}$  that will help us with that.

Let  $\hat{\mathcal{S}}' = \mathcal{S} \setminus \hat{\mathcal{S}} = \{s_1, s_2, \dots, s_k\}$  be the set of situations *not* in  $\hat{\mathcal{S}}$ . (Thus,  $\hat{\mathcal{S}}$  and  $\hat{\mathcal{S}}'$  are disjoint, and their union is  $\mathcal{S}$ .)

In the root of the behavioural game tree, we ask  $q(s_1)$  and get to the computer node for answering the question. Since  $s_1 \in \hat{\mathcal{S}}'$  and  $\hat{s} \in \hat{\mathcal{S}}$ , both answers are still possible, and the answer  $\tau(\hat{s}, q(s_1))$  is “false”. Thus we can follow the edge corresponding to “false” to the next player node. We can see from our definition of the question evaluation function that in this player node, all situations except  $s_1$  are still possible.

Now we repeat from there with question  $s_2$ . Again,  $\tau(s_2, \hat{s}) = \text{“false”}$ , and we follow the corresponding edge, leading to a player node in which  $\mathcal{S} \setminus \{s_1, s_2\}$  are still possible.

By repeating this for all questions in  $\hat{\mathcal{S}}'$ , we get to a player node in which

$$\mathcal{S} \setminus \{s_1, s_2, \dots, s_k\} = \mathcal{S} \setminus \hat{\mathcal{S}}' = \mathcal{S} \setminus (\mathcal{S} \setminus \hat{\mathcal{S}}) = \hat{\mathcal{S}}$$

is the set of remaining situations.  $\square$

### 5.3.2 Upper bound: $n \leq s \cdot q! \cdot e$

**Theorem 5.12.**  $n \leq s \cdot \sum_{k=0}^q \frac{q!}{k!} \leq s \cdot q! \cdot e$  (where  $e$  in this case denotes the Euler number).

*Proof.* Remember that we have exactly one player node for each set of remaining situations that occurs in at least one information set of the standard game tree. Therefore, there can be at most as many player nodes in the decision set graph as there are information sets in the standard game tree, and since information sets are disjoint and non-empty, there can be at most as many information sets as there are player nodes in the standard game tree.

In Section 4.2 we have shown that there are exactly  $s \cdot \sum_{k=0}^q \frac{q!}{k!}$  player nodes, which in turn is smaller than  $s \cdot q! \cdot e$ .  $\square$

### 5.3.3 Upper bound: $q! \cdot a^q \cdot e^{\frac{1}{a}}$

**Theorem 5.13.**  $n \leq \sum_{k=0}^q \frac{a^{q-k} q!}{k!} \leq q! \cdot a^q \cdot e^{\frac{1}{a}}$  (where  $e$  in this case denotes the Euler number).

*Proof.* Remember that we also have exactly one player node for each set of remaining situations that occurs in at least one player node of the behavioural game tree. Therefore, there can be at most as many player nodes in the decision set graph as there are player nodes in the behavioural game tree.

In Section 4.4 we have shown a very unwieldy way to count the number of player nodes in the behavioural game tree. Let us now look at upper bounds for that number.

- There is one root node with  $q + 1$  outgoing edges.
- In the next level, we have one estimate leaf and  $q$  computer nodes. The number of outgoing edges of the computer nodes is the number  $|A_q|$  of possible answers for the question that was asked. Thus, each computer node has at most  $a$  outgoing edges.
- On the third level, we therefore have at most  $q \cdot a$  player nodes. Each of them has  $q - 1$  outgoing edges for questions and one for taking a guess.
- Accordingly, we have at most  $q \cdot a$  estimation nodes on the next level, and at most  $q \cdot (q - 1) \cdot a$  computer nodes. Each of these computer nodes again has at most  $a$  outgoing edges.
- On the next level, we therefore altogether have at most  $q \cdot (q - 1) \cdot a^2$  player nodes.
- Accordingly, on the next level we have the same number of estimate leafs and  $q - 2$  times that number of computer nodes.
- Generally, on the level  $2k + 1$  we have at most  $q \cdot (q - 1) \cdot \dots \cdot (q - k + 1) \cdot a^k$  player nodes. On level  $2k + 2$  we have the same number of estimate leafs and  $q - k$  times that number of computer nodes.
- On level  $2q + 1$  finally, we have at most  $q \cdot (q - 1) \cdot \dots \cdot 3 \cdot 2 \cdot 1 \cdot a^q$  player nodes.
- On level  $2q + 2$  we have the same number of estimate leafs.

Counting only the player nodes, this altogether gives at most

$$\begin{aligned}
 n' &= 1 + qa + q(q - 1)a^2 + q(q - 1)(q - 2)a^3 + \dots + q(q - 1) \cdot \dots \cdot 3 \cdot 2 \cdot 1 \cdot a^q \\
 &= \sum_{k=0}^q \frac{a^{q-k} q!}{k!} \\
 &= q! \cdot a^q \cdot \sum_{k=0}^q \frac{a^{-k}}{k!} \\
 &= q! \cdot a^q \cdot \sum_{k=0}^q \frac{\left(\frac{1}{a}\right)^k}{k!} \\
 &\leq q! \cdot a^q \cdot \sum_{k=0}^{\infty} \frac{\left(\frac{1}{a}\right)^k}{k!} \\
 &= q! \cdot a^q \cdot e^{\frac{1}{a}}
 \end{aligned}$$

player nodes, and consequently at most the same number of player nodes in the decision set graph.  $\square$

### 5.3.4 Upper bound: $n \leq 2^q s + 1$

For the next upper bound, we will for each subset  $\hat{Q} \subseteq Q$  of questions look at the set of player nodes in which exactly the questions from  $\hat{Q}$  have been asked. This obviously splits the set of player nodes into disjoint classes, and there are exactly  $2^q$  such classes.

We will show that for each class, the player nodes in that class have at most  $s$  different sets of remaining situations.

A short example that will help to understand the following proof:

**Example 5.14.** *We play a kind of two-dimensional number guessing game: The computer hides one dot on a  $10 \times 10$  grid. The player can ask questions of the type “Is the  $x$ -coordinate larger than  $a$ ?” and “Is the  $y$ -coordinate larger than  $b$ ?”.*

*Let us look at all player nodes we can reach by asking “Is the  $x$ -coordinate larger than 5?”, “Is the  $y$ -coordinate larger than 3?” and “Is the  $y$ -coordinate larger than 6?”, in any order.*

*These questions dissect the grid into 6 rectangles, as shown in Figure 5.1.*

*Let us look at the player node in which the questions were asked in the order “Is the  $y$ -coordinate larger than 3?”, “Is the  $x$ -coordinate larger than 5?”, and “Is the  $y$ -coordinate larger than 6?”, and the answers were “yes”, “yes”, “no”, respectively. Then the fields where the dot could be hidden are exactly the fields in the middle right rectangle.*

*We see that similarly, for each player node in which exactly these three questions were asked, exactly the fields in one of the 6 rectangles could still contain the dot hidden by the computer. Which rectangle that is depends on the answers given.*

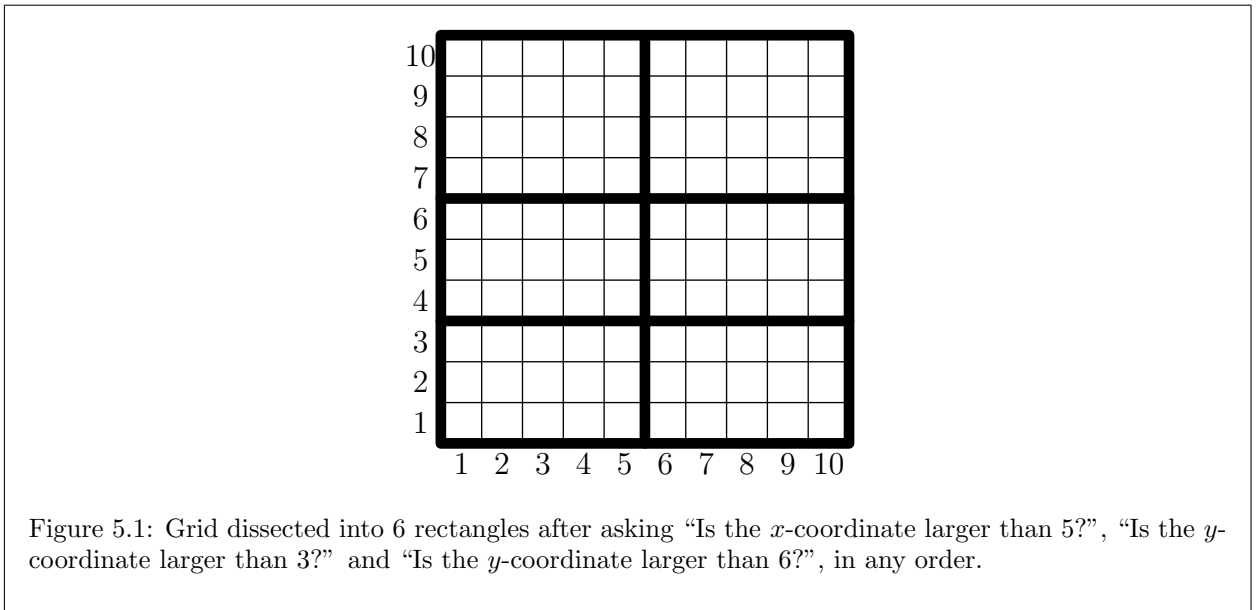


Figure 5.1: Grid dissected into 6 rectangles after asking “Is the  $x$ -coordinate larger than 5?”, “Is the  $y$ -coordinate larger than 3?” and “Is the  $y$ -coordinate larger than 6?”, in any order.

As we can see in the example, even if asking the questions in a different order, and even if getting different answers, there are only a few sets of remaining situations that can occur. We also see that each situation is contained in at most one of these sets – one of the 6 rectangles in our example –, so there can be at most  $s$  sets that are not empty.

Consequently, each such class can “create” (during the merging) at most  $s$  player nodes in the decision set graph. In other words, each player node in the decision set graph needs a “license to exist” from some player node in the behavioural game tree (since the decision set graph only contains nodes that encompass at least one player node from the behavioural game tree). However, each such class of player nodes in the behavioural game tree can give at most  $s$  such “licenses”.

Now we will just express this in a more formal way. There are two different ways to do this, and since they make use of slightly different properties, both of them seem interesting enough to be included in the thesis.

For the first method, we start with a lemma:

**Lemma 5.15.** *Let  $\hat{Q} \subseteq Q$  be a set of asked questions. Let  $\hat{N}$  be the subset of the player nodes of the behavioural game tree in which exactly the questions in  $\hat{Q}$  have been asked already, and let  $\hat{N}' = \text{encompassing}(\hat{N})$  be the set of nodes in the decision set graph that encompass the nodes of  $\hat{N}$ .*

*Then each situation  $s \in \mathcal{S}$  is contained in exactly one of the nodes of  $\hat{N}'$ .*

*Proof.* First we will show that each situation  $s \in \mathcal{S}$  is contained in at least one of the nodes in  $\hat{N}$ :

Let us define any order of the elements in  $\hat{Q}$ , i.e.,  $\hat{Q} = \{q_1, q_2, \dots, q_m\}$ . In the root node of the behavioural game tree, all situations are still possible, including  $s$ . We follow the path for asking question  $q_1$  and reach the computer node for answering this question. This computer node has an outgoing edge for each answer that occurs at least once. Let  $a_1 = \tau(s, q_1)$  be the answer to  $q_1$  in situation  $s$ , then there exists an outgoing edge for  $a_1$ . In the player node that this answer leads to, situation  $s$  is still among the possible situations.

We now repeat that with question  $q_2$ : In the player node we reached by asking  $q_1$  and following answer  $a_1$ , the situation  $s$  is still possible. We ask  $q_2$  and reach a computer node. Let  $a_2 = \tau(s, q_2)$  be the answer for  $s$  to question  $q_2$ , then (since there must exist an outgoing edge for each answer that occurs) we follow the edge corresponding to answer  $a_2$ . This leads to a player node in which  $q_1$  and  $q_2$  have been asked already, and  $s$  is still possible.

We repeat again with question  $q_3$ , and then  $q_4, q_5, q_6$ , and so on, until reaching  $q_m$ . Following that path, we reach a player node in which exactly the questions in  $\hat{Q}$  have been asked (and which therefore is in  $\hat{N}$ ), and in which  $s$  is still possible, thus showing that such a node exists.

Using this, we can now show that each situation  $s$  is contained in exactly one of the nodes in  $\hat{N}'$ :

- As shown above, there exists at least one player node  $N$  in the behavioural game tree in which  $s$  is still possible. Let  $N'$  be the node in the decision set graph that encompasses  $N$ , then  $s$  is possible in that node as well.  $N'$  is an element of  $\hat{N}'$ , therefore  $s$  is possible in at least one node in  $\hat{N}'$ .
- Let us assume that there exist two distinct nodes  $N'_1$  and  $N'_2$  in  $\hat{N}'$  in which  $s$  is still possible. Then according to the definition of  $\hat{N}'$  there exist two player nodes  $N_1$  and  $N_2$  in  $\hat{N}$  such that  $N_1$  is encompassed by  $N'_1$  and  $N_2$  is encompassed by  $N'_2$ . Consequently,  $s$  is still possible in both  $N_1$  and  $N_2$ .

We know that in both  $N_1$  and  $N_2$  exactly the questions from  $\hat{Q}$  were asked. They might however either have the same answers to all questions, or different answers.

- If the same questions were asked *and* the same answers were given on the path to  $N_1$  and  $N_2$ , then also exactly the same situations would be still possible in  $N_1$  and  $N_2$ . This would however mean that  $N_1$  and  $N_2$  would be encompassed by the same node in the decision set graph, which contradicts our assumption.
- Let us assume there exists at least one question  $q \in \hat{Q}$  for which different answers were given on the paths to  $N_1$  and  $N_2$ . Since all player nodes contain exactly the situations to which all of the answers given on the path there apply, and  $s$  is contained in both  $N_1$  and  $N_2$ , this would in particular mean that there exists a question such that two different answers to the same question apply to  $s$ , which is not possible.

Therefore, each situation  $s$  can only be among the remaining situations of one node in  $\hat{N}'$ .

□

**Theorem 5.16.**  $n \leq 2^q s$ .

*Proof.* Let  $N$  be the set of player nodes in the behavioural game tree.

There are  $q$  questions, so there are  $2^q$  possibilities which questions have already been asked. For each subset  $\hat{Q} \subseteq Q$ , let  $\hat{N}(\hat{Q}) \subseteq N$  be the set of player nodes in the behavioural game tree in which exactly the questions in  $\hat{Q}$  have been asked. Then these sets  $\hat{N}(\hat{Q})$  are disjoint, and the union of all  $2^q$  of these sets is  $N$ :

$$N = \bigcup_{\hat{Q} \subseteq Q} \hat{N}(\hat{Q})$$

By Lemma 4.53 it holds that the set of player nodes in the decision set graph  $N'$  is the encompassing set of all player nodes in the behavioural game tree  $N$ , and thus

$$\begin{aligned} N' &= \text{encompassing}(N) \\ &= \text{encompassing}\left(\bigcup_{\hat{Q} \subseteq Q} \hat{N}(\hat{Q})\right) \\ &= \bigcup_{\hat{Q} \subseteq Q} \text{encompassing}(\hat{N}(\hat{Q})) \end{aligned}$$

(using Theorem 4.52 for the last step).

Using the triangle inequality, we get

$$|N'| = \left| \bigcup_{\hat{Q} \subseteq Q} \text{encompassing}(\hat{N}(\hat{Q})) \right| \leq \sum_{\hat{Q} \subseteq Q} |\text{encompassing}(\hat{N}(\hat{Q}))|.$$

As we have just shown in Lemma 5.15, for each set of questions  $\hat{Q} \subseteq Q$ , each situation  $s$  is contained in exactly one of the nodes of  $\text{encompassing}(\hat{N}(\hat{Q}))$ . Since, on the other hand, each node in  $\text{encompassing}(\hat{N}(\hat{Q}))$  must by construction contain at least one remaining situation, there can be at most  $s$  such nodes, i.e.,  $|\text{encompassing}(\hat{N}(\hat{Q}))| \leq s$ .

We therefore get

$$|N'| \leq \sum_{\hat{Q} \subseteq Q} |\text{encompassing}(\hat{N}(\hat{Q}))| \leq \sum_{\hat{Q} \subseteq Q} s = 2^q s.$$

□

**Corollary 5.17.**  $n \leq 2^q s - s + 1$ .

*Proof.* We will basically use the same proof idea, but with one small optimization. As in the previous proof, we get

$$N' = \bigcup_{\hat{Q} \subseteq Q} \text{encompassing}(\hat{N}(\hat{Q})).$$

However, we now treat  $\hat{Q} = \emptyset$ , i.e., the case in which no question was asked, differently. There is only one node in which no question was asked yet, namely the root node. Thus our upper limit of  $s$  is a bit too sloppy there. Using the limit of 1 instead, we get the desired limit:

$$\begin{aligned} N' &= \bigcup_{\hat{Q} \subseteq Q} \text{encompassing}(\hat{N}(\hat{Q})) \\ &= \text{encompassing}(\hat{N}(\emptyset)) \cup \bigcup_{\substack{\hat{Q} \subseteq Q \\ \hat{Q} \neq \emptyset}} \text{encompassing}(\hat{N}(\hat{Q})) \\ |N'| &\leq |\text{encompassing}(\hat{N}(\emptyset))| + \sum_{\substack{\hat{Q} \subseteq Q \\ \hat{Q} \neq \emptyset}} |\text{encompassing}(\hat{N}(\hat{Q}))| \\ &\leq 1 + (2^q - 1) \cdot s \\ &= 1 + 2^q s - s \end{aligned}$$

□

We will now prove exactly the same result again, using a different approach (though it still follows the same underlying principles):

**Theorem 5.18.**  $n \leq 2^q s - s + 1$ .

*Proof.* If no question has been asked, then all situations are still possible.

Let us look at what happens after asking one question  $q_1 \in \mathcal{Q}$ : There are  $a_1 = |\bar{A}_{q_1}|$  possible answers; let us use  $\bar{A}_{q_1} = \{a_{1,1}, a_{1,2}, \dots, a_{1,a_1}\}$  to denote them. Then for each answer  $a_{1,i}$  we look at the subset of the situations to which this answer applies. Using the notation from Definition 4.29, these are the sets  $\hat{\mathcal{S}}((q_1, a_{1,1})), \hat{\mathcal{S}}((q_1, a_{1,2})), \dots, \hat{\mathcal{S}}((q_1, a_{1,a_1}))$ . Each situation  $s \in \mathcal{S}$  is contained in exactly one of these subsets, thus their disjoint union is  $\mathcal{S}$ :

$$\begin{aligned} \hat{\mathcal{S}}((q_1, a_{1,x})) \cap \hat{\mathcal{S}}((q_1, a_{1,y})) &= \emptyset \quad (x \neq y) \\ \bigcup_{i=1}^{a_1} \hat{\mathcal{S}}((q_1, a_{1,i})) &= \mathcal{S} \end{aligned}$$

After asking only  $q_1$ , these are the subsets of remaining situations that can occur. Since there are only  $s$  situations, at most  $s$  of these are not empty. Analogously, after asking any other single question, we get at most  $s$  non-empty subsets of  $\mathcal{S}$ . There are  $q$  ways to select one question, therefore at most  $s \cdot q$  non-empty sets that can occur after one asked question.

Next, let us assume that we ask two questions,  $q_1$  and  $q_2$ . Let again  $\bar{A}_{q_1} = \{a_{1,1}, a_{1,2}, \dots, a_{1,a_1}\}$  denote the possible answers to  $q_1$  and let  $\bar{A}_{q_2} = \{a_{2,1}, a_{2,2}, \dots, a_{2,a_2}\}$  denote the possible answers to question  $q_2$ . Then for each combination of answers  $(a_{1,i_1}, a_{2,i_2}) \in \bar{A}_{q_1} \times \bar{A}_{q_2}$  we again look at the subset of situations to which these answers apply. We get the following system of disjoint subsets:

$$\begin{aligned} \hat{\mathcal{S}}((q_1, a_{1,x_1}), (q_2, a_{2,x_2})) \cap \hat{\mathcal{S}}((q_1, a_{1,y_1}), (q_2, a_{2,y_2})) &= \emptyset \quad (x_1 \neq y_1 \vee x_2 \neq y_2) \\ \bigcup_{(a_{1,i_1}, a_{2,i_2}) \in \bar{A}_{q_1} \times \bar{A}_{q_2}} \hat{\mathcal{S}}((q_1, a_{1,i_1}), (q_2, a_{2,i_2})) &= \mathcal{S} \end{aligned}$$

Therefore, at most  $s$  of these sets are non-empty. There are  $\binom{q}{2}$  ways to select two questions, leading to at most  $s \cdot \binom{q}{2}$  different such systems.

We now repeat this for more questions. After asking  $m$  questions  $q_1, q_2, \dots, q_m$ , we consider all tuples  $(a_{1,i_1}, a_{2,i_2}, \dots, a_{m,i_m}) \in \bar{A}_{q_1} \times \bar{A}_{q_2} \times \dots \times \bar{A}_{q_m}$ , where again  $\bar{A}_{q_k} = \{a_{k,1}, a_{k,2}, \dots, a_{k,a_k}\}$  denotes the possible answers to  $q_k$  for every  $k$ .

This leads to the already familiar system of disjoint subsets; that is, each intersection of two subsets is empty (except for intersection of a set with itself of course), and the union of all subsets is  $\mathcal{S}$ .

$$\begin{aligned} \hat{\mathcal{S}}((q_1, a_{1,x_1}), (q_2, a_{2,x_2}), \dots, (q_m, a_{m,x_m})) \cap \hat{\mathcal{S}}((q_1, a_{1,y_1}), (q_2, a_{2,y_2}), \dots, (q_m, a_{m,y_m})) &= \emptyset \\ \bigcup_{(a_{1,i_1}, a_{2,i_2}, \dots, a_{m,i_m}) \in \bar{A}_{q_1} \times \bar{A}_{q_2} \times \dots \times \bar{A}_{q_m}} \hat{\mathcal{S}}((q_1, a_{1,i_1}), (q_2, a_{2,i_2}), \dots, (q_m, a_{m,i_m})) &= \mathcal{S} \end{aligned}$$

Even though this can be a rather large number of such subsets by now – up to  $a^m$  to be exact –, again at most  $s$  of these are not empty. There are  $\binom{q}{m}$  ways to select  $m$  questions, therefore situations where  $m$  questions were asked can contribute at most  $s \cdot \binom{q}{m}$  different non-empty subsets of  $\mathcal{S}$ .

Altogether, we therefore get at most

$$\begin{aligned} n' &= 1 + s \cdot q + s \cdot \binom{q}{2} + s \cdot \binom{q}{3} + \dots + s \cdot \binom{q}{q-1} + s \cdot \binom{q}{q} \\ &= 1 + s \cdot \left( \binom{q}{1} + \binom{q}{2} + \dots + \binom{q}{q} \right) \\ &= 1 - s + s + s \cdot \left( \binom{q}{1} + \binom{q}{2} + \dots + \binom{q}{q} \right) \\ &= 1 - s + s \cdot \left( \binom{q}{0} + \binom{q}{1} + \binom{q}{2} + \dots + \binom{q}{q} \right) \\ &= 1 - s + s \cdot 2^q \end{aligned}$$

different non-empty subsets of remaining situations that can occur in the behavioural game tree and thus at most the same number of player nodes in the decision set graph.  $\square$



### 5.3.5 Upper bound: $n \leq 2^{aq}$ .

As a next idea, we will look at the subsets of  $\hat{\mathcal{S}}$  that we can get after asking only one question. In Example 5.14, after asking only “Is the  $x$ -coordinate larger than 5?”, we either get the left or the right half of the grid as set of remaining situations. After asking “Is the  $y$ -coordinate larger than 3?”, one possible set of remaining situations is the one containing the lowest 3 rows, and the other is the one containing the 7 rows above it. If we had a question that gave three different answers, for example, “What is the remainder of the  $y$ -coordinate modulo 3?”, then we would get three sets of remaining situations, one containing rows 3, 6 and 9, one containing rows 1, 4, 7 and 10, and a third containing rows 2, 5 and 8.

As we can see, every set of remaining situations that we can get after asking two or more questions is the intersection of some sets of remaining situations after asking one question. There are at most  $aq$  sets of remaining situations after one question, and there are, consequently, at most  $2^{aq}$  ways to select some of them and intersect them.

Somewhat more formal:

**Lemma 5.19.** *Let  $q_1$  and  $q_2$  be two questions, let  $a_1$  be one possible answer to  $q_1$  and let  $a_2$  be one possible answer to  $q_2$ . Then*

$$\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2)) = \hat{\mathcal{S}}((q_1, a_1)) \cap \hat{\mathcal{S}}((q_2, a_2))$$

holds.

*Proof.*

$$\begin{aligned} \hat{\mathcal{S}}((q_1, a_1), (q_2, a_2)) &= \{s \in \mathcal{S} \mid \tau(s, q_1) = a_1 \wedge \tau(s, q_2) = a_2\} \\ &= \{s \in \mathcal{S} \mid \tau(s, q_1) = a_1\} \cap \{s \in \mathcal{S} \mid \tau(s, q_2) = a_2\} \\ &= \hat{\mathcal{S}}((q_1, a_1)) \cap \hat{\mathcal{S}}((q_2, a_2)) \end{aligned}$$

□

**Corollary 5.20.** *For any questions  $q_1, q_2, \dots, q_k$  and corresponding answers  $a_1, a_2, \dots, a_k$ ,*

$$\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k)) = \hat{\mathcal{S}}((q_1, a_1)) \cap \hat{\mathcal{S}}((q_2, a_2)) \cap \dots \cap \hat{\mathcal{S}}((q_k, a_k))$$

holds.

*Proof.* This follows directly from Lemma 5.19. □

**Theorem 5.21.**  $n \leq 2^{aq}$ .

*Proof.* First of all, we see that question  $q_1$  splits the set of situations into  $a_1$  sets of remaining situations of the form  $\hat{\mathcal{S}}((q_1, a_i))$ . Similarly,  $q_2$  creates  $a_2$  subsets of the form  $\hat{\mathcal{S}}((q_2, a_i))$ , and so on. Altogether, there are  $\sum_{i=1}^q a_i \leq aq$  such sets.

We need to count all sets of remaining situations that can occur. Since the order of question is irrelevant, we can describe the remaining situation in each player node in the standard and behavioural game tree as  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$  (simply by looking at the questions and answers on the path to that node). We therefore just need to count how many such sets exist.

However, as we just saw in Corollary 5.20, this is equivalent to the number of sets of the form  $\hat{\mathcal{S}}((q_1, a_1)) \cap \hat{\mathcal{S}}((q_2, a_2)) \cap \dots \cap \hat{\mathcal{S}}((q_m, a_m))$ .

All of these sets can be created by taking some of the sets  $\hat{\mathcal{S}}((q_x, a_y))$  and intersecting them. There are  $2^{aq}$  ways to select a subset of these  $aq$  sets. □

**Corollary 5.22.**  $n \leq 2^{sq}$ .

*Proof.* This follows from the previous theorem and  $a \leq s$ :

$$n \leq 2^{aq} \leq 2^{sq}.$$

□

### 5.3.6 Upper bound: $n \leq (1 + a)^q$

We now show an upper bound that is strictly better than the previous one (except for  $a = 1$ , which would mean that all questions have only one answer). To do so, we will look at the proof method of Theorem 5.18 again and deduct a slightly different limit. In Theorem 5.18, we showed that after asking  $m$  questions, there are at most  $s$  combinations of answers for which at least one situation exists that satisfies all these answers.

However, we can also simply count how many combinations of answers there are at all: Each set of remaining situations occurring in the behavioural game tree can be described with a list of (question, answer) tuples, since the set of remaining situations in a node contains exactly the situations from  $\mathcal{S}$  that satisfy the answers retrieved on the path from the root to the node. Counting the number of ways to select some questions and some answers for them therefore gives another upper limit for the number of different sets of remaining situations that can occur.

**Theorem 5.23.** *Let  $\{q_1, q_2, \dots, q_q\}$  be the set of questions, and let  $a_i = |\overline{A}_{q_i}|$  for all  $i$ . Then  $n \leq (1 + a_1)(1 + a_2) \cdots (1 + a_q)$ .*

*Proof.* As in the proof to Theorem 5.18, we look at sets of situations that have certain answers. After asking  $m$  questions  $q_{i_1}, q_{i_2}, \dots, q_{i_m}$ , we look at all tuples  $(a_{i_1, j_1}, a_{i_2, j_2}, \dots, a_{i_m, j_m}) \in \overline{A}_{q_{i_1}} \times \overline{A}_{q_{i_2}} \times \cdots \times \overline{A}_{q_{i_m}}$  and the corresponding subsets of situations that gave the following system:

$$\begin{aligned} \hat{\mathcal{S}}((q_{i_1}, a_{i_1, x_1}), (q_{i_2}, a_{i_2, x_2}), \dots, (q_{i_m}, a_{i_m, x_m})) \cap \hat{\mathcal{S}}((q_{i_1}, a_{i_1, y_1}), (q_{i_2}, a_{i_2, y_2}), \dots, (q_{i_m}, a_{i_m, y_m})) &= \emptyset \\ \bigcup_{(a_{i_1, j_1}, a_{i_2, j_2}, \dots, a_{i_m, j_m}) \in \overline{A}_{q_{i_1}} \times \overline{A}_{q_{i_2}} \times \cdots \times \overline{A}_{q_{i_m}}} \hat{\mathcal{S}}((q_{i_1}, a_{i_1, j_1}), (q_{i_2}, a_{i_2, j_2}), \dots, (q_{i_m}, a_{i_m, j_m})) &= \mathcal{S} \end{aligned}$$

In the previous proof, we used the fact that there are only  $s$  situations, so at most  $s$  of these subsets can be non-empty. We can, however, also just directly count how many such subsets there are in this system: We get one subset for each combination of answers to the asked questions. There are  $a_{i_1} \cdot a_{i_2} \cdots a_{i_m}$  combinations of answers in  $\overline{A}_{q_{i_1}} \times \overline{A}_{q_{i_2}} \times \cdots \times \overline{A}_{q_{i_m}}$ , so the system contains  $a_{i_1} \cdot a_{i_2} \cdots a_{i_m}$  sets.

Now we just need to find all ways to select  $m$  questions, and calculate for each of them how many sets the system contains. Each occurring set of remaining situations is contained in one of these systems, so the total number of sets in the systems is an upper bound for the number of nodes in the decision set graph.

For the case with no question asked, this trivially gives one possible set.

For all cases in which one question was asked together, there are  $a_1 + a_2 + \cdots + a_q$  sets.

For cases with two asked questions, we get  $a_1 a_2 + a_1 a_3 + \cdots + a_1 a_q + a_2 a_3 + a_2 a_4 + \cdots + a_2 a_q + a_3 a_4 + \cdots + a_{q-1} a_q$  sets.

Generally, for cases with  $m$  asked questions, we get the sum of all possible ways to multiply  $m$  of the  $a_i$ .

Finally, the cases with  $q$  asked questions contribute up to  $a_1 a_2 \cdots a_q$  sets.

Altogether, these are

$$\begin{aligned} n' &\leq 1 \\ &+ a_1 + a_2 + \cdots + a_q \\ &+ a_1 a_2 + a_1 a_3 + \cdots + a_1 a_q + a_2 a_3 + a_2 a_4 + \cdots + a_2 a_q + a_3 a_4 + \cdots + a_{q-1} a_q \\ &+ a_1 a_2 a_3 + \cdots + a_{q-2} a_{q-1} a_q \\ &+ \cdots \\ &+ a_1 a_2 \cdots a_q \\ &= (1 + a_1)(1 + a_2) \cdots (1 + a_q) \end{aligned}$$

sets.

We could also prove this more directly: We want to look at all subsets of  $\mathcal{S}$  that are defined by demanding certain answers for some of the questions. That is, for each question  $q$  we can either demand it to have one of its  $a_q$  answers, or we can not restrict it. Therefore, for each question  $q$  we have  $a_q + 1$  choices, and we can choose for each question independently. This leads again to the same product.  $\square$

**Corollary 5.24.**  $n \leq (1 + a)^q$ .

*Proof.* This follows directly from the previous theorem and the fact that  $a_i \leq a$  for all  $i$ , and consequently

$$n \leq (1 + a_1)(1 + a_2) \cdots (1 + a_q) \leq (1 + a)(1 + a) \cdots (1 + a) = (1 + a)^q .$$

□

**Corollary 5.25.**  $n \leq (1 + s)^q$ .

*Proof.* This follows directly from the previous theorem and  $a \leq s$ , and consequently

$$n \leq (1 + a)^q \leq (1 + s)^q .$$

□

We will now show that there exist games for which the limit from Theorem 5.23 is indeed reached.

**Theorem 5.26.** *For each  $q$  and each set of values  $a_1, a_2, \dots, a_q$ , there exists a game with*

- $q$  questions such that  $|\overline{A}_{q_i}| = a_i$  for all  $i$ ;
- $a_1 a_2 \cdots a_q$  situations; and
- $(1 + a_1)(1 + a_2) \cdots (1 + a_q)$  player nodes in the decision set graph.

*Proof.* We define  $\mathcal{Q} = \{q_1, q_2, \dots, q_q\}$ . (The  $q_i$  don't have any special meaning here, we just use them to denote  $q$  different questions.) For each  $i$ , let  $A_{q_i} = \overline{A}_{q_i} = \{1, 2, \dots, a_i\}$ .

We create one situation for each possible tuple  $(a_1, a_2, \dots, a_q) \in \overline{A}_{q_1} \times \overline{A}_{q_2} \times \cdots \times \overline{A}_{q_q}$ , and denote that situation with  $s(a_1, a_2, \dots, a_q)$ . This gives us  $a_1 a_2 \cdots a_q$  different situations.

We define the evaluation function as follows:

$$\tau(s(a_1, a_2, \dots, a_q), q_i) = a_i$$

That is, the  $i$ -th question returns the value of the  $i$ -th element of the tuple belonging to the situation.

As in the proof of Theorem 5.23, we look at the situation after asking  $m$  questions, which leads to the following system:

$$\begin{aligned} \hat{\mathcal{S}}((q_{i_1}, a_{i_1, x_1}), (q_{i_2}, a_{i_2, x_2}), \dots, (q_{i_m}, a_{i_m, x_m})) \cap \hat{\mathcal{S}}((q_{i_1}, a_{i_1, y_1}), (q_{i_2}, a_{i_2, y_2}), \dots, (q_{i_m}, a_{i_m, y_m})) &= \emptyset \\ \bigcup_{(a_{i_1, j_1}, a_{i_2, j_2}, \dots, a_{i_m, j_m}) \in \overline{A}_{q_{i_1}} \times \overline{A}_{q_{i_2}} \times \cdots \times \overline{A}_{q_{i_m}}} \hat{\mathcal{S}}((q_{i_1}, a_{i_1, j_1}), (q_{i_2}, a_{i_2, j_2}), \dots, (q_{i_m}, a_{i_m, j_m})) &= \mathcal{S} \end{aligned}$$

However, we now can show that each of the subsets  $\hat{\mathcal{S}}((q_{i_1}, a_{i_1, x_1}), (q_{i_2}, a_{i_2, x_2}), \dots, (q_{i_m}, a_{i_m, x_m}))$  contains at least one situation. To do so, we define values  $b_j$  in the following way: If the list  $((q_{i_1}, a_{i_1, x_1}), (q_{i_2}, a_{i_2, x_2}), \dots, (q_{i_m}, a_{i_m, x_m}))$  contains a tuple  $(q_j, a_{j, y_j})$  (that is, the question  $j$  is among the asked questions and has to be answered with  $a_{j, y_j}$ ), we set  $b_j$  to  $a_{j, y_j}$ . Otherwise, we set  $b_j$  to 1 (which is a valid answer for every question). The tuple  $(b_1, b_2, \dots, b_q)$  therefore is included in  $\hat{\mathcal{S}}((q_{i_1}, a_{i_1, x_1}), (q_{i_2}, a_{i_2, x_2}), \dots, (q_{i_m}, a_{i_m, x_m}))$ .

Consequently, all sets in the system are non-empty, and the limit  $(1 + a_1)(1 + a_2) \cdots (1 + a_q)$  is reached. □

Finally, let us combine the last two approaches:

**Corollary 5.27.** *Let  $a_i$  be defined as in Theorem 5.23. Then*

$$\begin{aligned} n' &\leq 1 \\ &+ a_1 + a_2 + \cdots + a_q \\ &+ \min(s, a_1 a_2) + \min(s, a_1 a_3) + \cdots + \min(s, a_1 a_q) + \min(s, a_2 a_3) + \cdots + \min(s, a_{q-1} a_q) \\ &+ \min(s, a_1 a_2 a_3) + \cdots + \min(s, a_{q-2} a_{q-1} a_q) \\ &+ \cdots \\ &+ \min(s, a_1 a_2 \cdots a_q) . \end{aligned}$$

*Proof.* Theorem 5.18 showed that each system can contain at most  $s$  sets that are not empty, and Theorem 5.23 showed that each system can contain at most  $a_1 a_2 \cdots a_q$  sets (if  $q_1, q_2, \dots, q_m$  are the questions for which the system is created). Consequently, they are both upper limits, and each system can at most contain the *lower* of these two. □

### 5.3.7 Upper bounds in the hypercuboid

We will now look at another approach to derive upper bounds. This requires some theory first, and will then give us some already familiar and some new upper bounds.

#### Two-dimensional hypercuboids (planes)

For starters, let us look at a game with only two questions. We draw a grid with the answers to question  $q_1$  on the  $x$ -axis and the answers to question  $q_2$  on the  $y$ -axis, as shown in Figure 5.2.

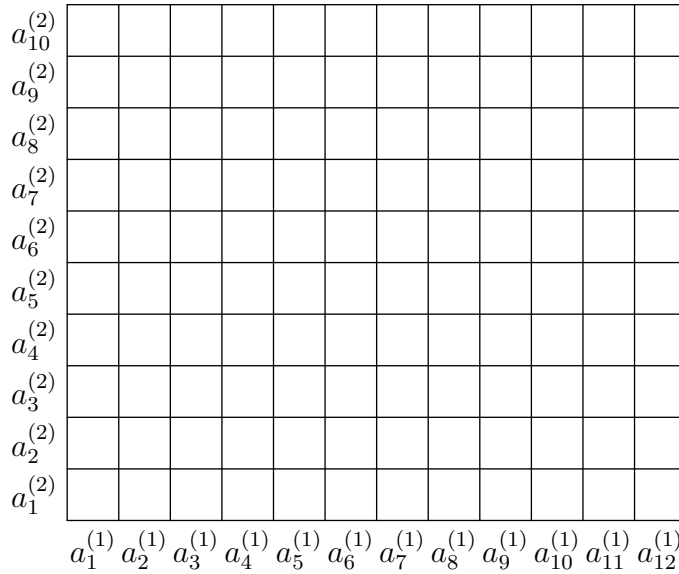


Figure 5.2: Grid for two questions.

In the next step, we draw one dot for each situation, in the cell corresponding to the answers to that situation. (That is, for each situation  $s$ , we draw a dot in cell  $(\tau(s, q_1), \tau(s, q_2))$ , labeled with the corresponding situation.) See Figure 5.3 for an example; the labeling of the nodes was omitted in that graph.

In the proof for Theorem 5.23, we looked at all possible sets that can occur after asking a subset of the questions. The sets after asking  $q_1$  now correspond exactly to the columns in this graph: the set  $\hat{\mathcal{S}}((q_1, a_7^{(1)}))$  is exactly the set of the situations (dots) in the 7th column. Likewise, the sets after asking  $q_2$  correspond to the rows. The sets after asking both  $q_1$  and  $q_2$  correspond to individual cells in the grid, and the set before asking any question corresponds to the set of all situations in the grid.

**Example 5.28.** *To give an example, we give numbers to the situation dots in Figure 5.3 and get Figure 5.4. We therefore get the following (non-empty) sets:*

<i>From the entire grid (i.e., no questions asked):</i>			
• {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}			
<i>From columns (i.e., only <math>q_1</math> asked):</i>			
• {1}	• {5, 6, 7, 8}	• {11, 12}	• {16}
• {2, 3, 4}	• {9, 10}	• {13, 14, 15}	
<i>From rows (i.e., only <math>q_2</math> asked):</i>			
• {8}	• {1, 3}	• {12, 14}	• {5, 6, 7}
• {4}	• {2, 9, 10, 15}	• {11, 13}	• {16}



For example, if we ask question  $q_1$  and get answer  $a_7^{(1)}$ , then we know that situations 5, 6, 7 and 8 are still possible, that is, exactly the situations in the column for  $a_7^{(1)}$  (since we put each situation  $s$  into cell  $(\tau(s, q_1), \tau(s, q_2))$ , so column  $a_7^{(1)}$  contains exactly the situations for which  $\tau(s, q_1) = a_7^{(1)}$ ). If we ask  $q_2$  next and get answer  $a_2^{(2)}$ , then only situation 8 is possible any more.

We therefore see that after asking  $q_1$ , we can restrict the remaining situations to one column, and after asking  $q_2$ , we can restrict it to one row. After asking both, we can restrict it to one cell, and before having asked anything, all situations in the grid are still considered possible.

In the example, the set of situations in row  $a_9^{(2)}$  contains situations 5, 6 and 7. However, also the cell at the intersection of row  $a_9^{(2)}$  and column  $a_7^{(1)}$  contains the same situations. In the game tree this would mean that the player node after asking both questions and getting answers  $a_7^{(1)}$  and  $a_9^{(2)}$  has the same set of remaining situations as the one after asking only  $q_2$  and getting answer  $a_9^{(2)}$ . Therefore, even though these player nodes could have different sets of remaining situations, they have the same sets in this case, so they are encompassed by the same node in the decision set graph.

If however row  $a_9^{(2)}$  contained another situation in a different column, then the node after asking only  $q_2$  and getting answer  $a_9^{(2)}$  would have a different set of remaining situations, and there would be another node in the decision set graph for it.

**Example 5.29.** As another example, let us assume that all situations were in the same row  $a_3^{(2)}$ ; that is, all situations in the entire game have the same answer to question  $q_2$ . Then the set of all situations  $\mathcal{S}$  would be the same as the set of situations after asking  $q_2$  and getting answer  $a_3^{(2)}$ . That is, there is only one player node in which only question  $q_2$  has been asked, and this player node has  $\mathcal{S}$  as set of remaining situations. Therefore, it does not contribute a new unique set of remaining situations. Likewise, for each column we see that asking  $q_1$  and getting some answer  $a_i^{(1)}$  has the same set of remaining situations as asking  $q_1$  and getting answer  $a_i^{(1)}$  and asking  $q_2$  and getting answer  $a_3^{(2)}$ . Therefore, also these player nodes in the behavioural game tree contribute less different sets of remaining situations than they could in a different configuration of situations.

**Example 5.30.** As the opposite extreme, let us look at a game in which each cell contains a situation. (For example, let us assume that the computer hides a dot on a  $10 \times 10$  grid again. For now, we can only have two questions (but any number of answers), so let us assume our two questions are “What is the  $x$ -coordinate of the dot?” and “What is the  $y$ -coordinate of the dot?” (which is of course a very boring game). Then each situation  $(x, y)$  also produces a dot in cell  $(x, y)$  of our situation grid. Before asking any questions, all 100 situations are possible. After asking for the  $x$ -coordinate and getting answer “3”, only 10 situations are possible, the 10 with coordinates of the form  $(3, *)$ , which are represented in the third column. After asking for the  $x$ -coordinate and getting answer “4”, other 10 situations are remaining. In total, we have 10 possible answers, and for each, we get a set of 10 remaining situations. These sets are disjoint (since a dot that has  $x$ -coordinate 3 cannot at the same time have  $x$ -coordinate 7). The same applies to asking for the  $y$ -coordinate. After asking both questions, we know the exact situation, so the set of remaining situations only contains that one situation. In the grid, this situation is represented in one of the cells. We therefore get altogether 100 different sets of remaining situations after having asked both questions, 10 sets of remaining situations after having asked for the  $x$ -coordinate, 10 sets of remaining situations after having asked for the  $y$ -coordinate, and one set of remaining situations before asking any questions. This exactly corresponds to the 100 cells, 10 columns, 10 rows, and 1 entire grid, and results in a total of 121 player nodes.

Consequently, if we count how many different sets of situations we get when counting the sets for rows, for columns, for cells, and the one set containing all situations, then this is exactly the number of player nodes in the decision set graph.

For the remainder of this section, we will therefore try to find smart ways to count how many different sets we can at most get, look at different distributions of these dots, and generalize it to more than two questions.

**Lemma 5.31.** Two non-empty sets of the same “type” (row, column, or cell) are never duplicates of each other.

*Proof.* This follows directly from the fact that situations are distinguishable. If, for example, the (non-empty) sets for two columns were duplicates of each other, there would have to be at least one situation that is contained in both columns, which would be a contradiction to our construction.  $\square$

We count the number of sets in the following order:

**Algorithm 5.32.** *Let  $R$  be a data structure that contains sets of remaining situations.  $R$  is initially empty. For each cell: Add the corresponding set to  $R$ . Then, for each column: If the corresponding set has not been added to  $R$  yet, add it to  $R$ . Then, for each row: If the corresponding set has not been added to  $R$  yet, add it to  $R$ . Then, for the entire grid: If the corresponding set has not been added to  $R$  yet, add it to  $R$ .*

**Theorem 5.33.** *The number of sets in  $R$  after running Algorithm 5.32 equals the number of player nodes in the decision set graph.*

*Proof.* See argumentation above. □

We make the following observation:

**Lemma 5.34.** *When adding elements in the order described in Algorithm 5.32, then:*

- *a row or column adds a new set if and only if it contains at least two cells that are non-empty; and*
- *the entire grid adds a new set if and only if it contains at least two rows and two columns that are non-empty.*

*Proof.* For rows and columns this is obvious. If a column contains only one non-empty cell, then that set has already been added in the first step when we added the cells. If it contains two cells, then it was not added in the first step yet.

For rows, it is equally obvious that the set of a row containing two cells has neither been added in the step where we added cells nor in the step where we added columns.

Finally, for the entire grid, if it contains only one non-empty row, then that row contains all the elements in the grid and has already been added when adding the rows (or even earlier, if it already was a duplicate at that point). Likewise, if there is only one column, it has already been added. If there are at least two non-empty rows and at least two non-empty columns, on the other hand, then none of the previous steps could have added it already, because none of them added sets with two dimensions yet. □

**Theorem 5.35.** *The number  $n$  of player nodes in the game tree can be calculated as*

$$\begin{aligned} n = & \#[\text{non-empty cells}] \\ & + \#[\text{rows with at least two non-empty cells}] \\ & + \#[\text{columns with at least two non-empty cells}] \\ & + \begin{cases} 1 & \text{grid has at least two non-empty rows and two non-empty columns} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

*Proof.* This follows directly from the previous lemma. □

We now want to find out how high that number can be at most, given the number  $s$  of situations.

**Lemma 5.36.** *Adding an additional situation to any given configuration never decreases the number of counted sets.*

*Proof.* For each term in the sum in Theorem 5.35, adding a situation either increases the number (if it turns a row/column/grid that previously contained into only one non-empty cell/row/column into one that has two) or leaves it unchanged. It never decreases one of these summands. □

**Lemma 5.37.** *If we want to distribute situations into the cells in such a way that we maximize the number of sets counted in Algorithm 5.32, then adding two situations to the same cell is never an optimal solution (or at least not the only optimal solution).*

*Proof.* Let us assume that we have one cell that contains at least two situations. We remove one of these. This does not change anything about the number of sets counted.

Now we can add this situation anywhere else, which according to Lemma 5.36 does not decrease the sum, and possibly increases it. □

**Theorem 5.38.** For  $s$  situations, two questions and any number of possible answers, there can never exist more than  $s + 2 \lfloor \frac{s}{2} \rfloor + 1$  player nodes in the decision set graph.

*Proof.* We look at the individual terms in the sum in Theorem 5.35.

$\#[\text{non-empty cells}]$  can be at most  $s$ .

At most  $\lfloor \frac{s}{2} \rfloor$  columns can contain 2 or more situations. Therefore,

$$\#[\text{rows with at least two non-empty cells}] \leq \lfloor \frac{s}{2} \rfloor .$$

Analogously,

$$\#[\text{columns with at least two non-empty cells}] \leq \lfloor \frac{s}{2} \rfloor .$$

The grid finally can contribute at most 1.

In total, this gives at most  $s + 2 \lfloor \frac{s}{2} \rfloor + 1$  player nodes. □

Now that we have this upper limit, we try to find games that get close to it.

**Theorem 5.39.** For each  $s > 2$ , there exists a configuration with  $s$  situations, two questions and  $s + 2 \lfloor \frac{s}{2} \rfloor + 1$  player nodes in the decision set graph.

*Proof.* For even  $s = 2k$ , we build a “stair” like the one in Figure 5.5.

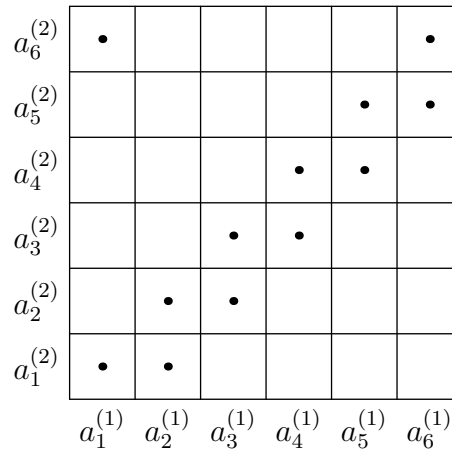


Figure 5.5: One way to create configurations with the maximum number of player nodes.

The number of player nodes in this case is

$$\begin{aligned}
 n &= \#[\text{non-empty cells}] \\
 &\quad + \#[\text{rows with at least two non-empty cells}] \\
 &\quad + \#[\text{columns with at least two non-empty cells}] \\
 &\quad + \begin{cases} 1 & \text{grid has at least two non-empty rows and two non-empty columns} \\ 0 & \text{otherwise} \end{cases} \\
 &= 2k + k + k + 1 \\
 &= s + 2 \lfloor \frac{s}{2} \rfloor + 1
 \end{aligned}$$

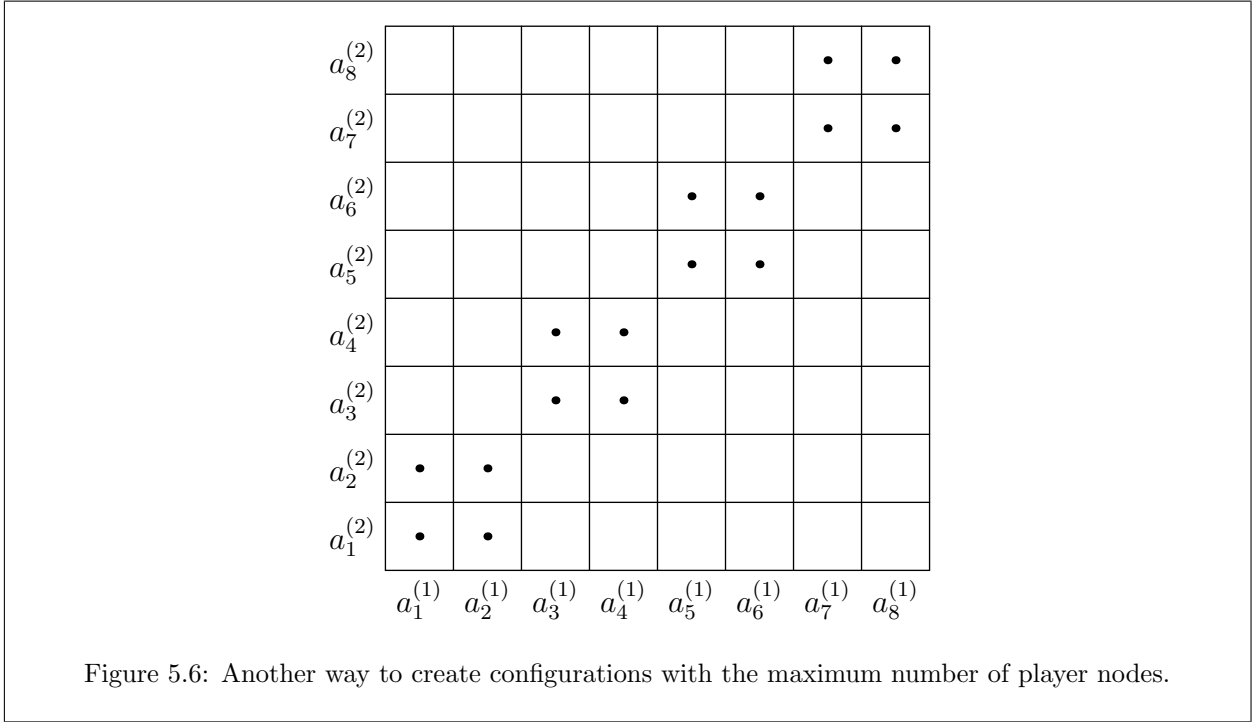
For odd  $s = 2k + 1$ , take the first  $s - 1 = 2k$  points to build the “stair” as above, resulting in  $2k + 2 \lfloor \frac{2k}{2} \rfloor + 1 = 2k + 2 \lfloor \frac{2k+1}{2} \rfloor + 1$  player nodes. Now we add the last situation to any empty cell. This increases  $\#[\text{non-empty cells}]$  by 1 and leaves the other terms unchanged. Consequently, we get  $2k + 2 \lfloor \frac{2k+1}{2} \rfloor + 1 + 1 = s + 2 \lfloor \frac{s}{2} \rfloor + 1$  player nodes. □



We will now show a slightly weaker version of this, which is however easier to generalize.

**Theorem 5.40.** *Let  $s = 4k$  be a multiple of 4, then there exists a configuration with  $s$  situations, two questions and  $2s + 1 = s + 2 \lfloor \frac{s}{2} \rfloor + 1$  player nodes in the decision set graph.*

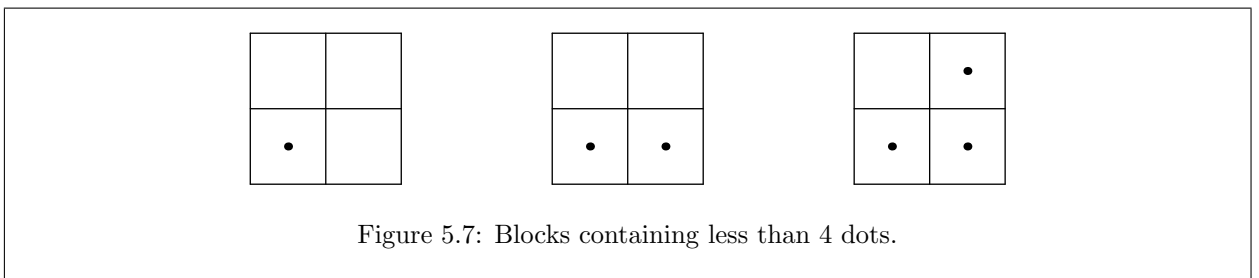
*Proof.* We add  $2 \times 2$  blocks of situations as shown in Figure 5.6.



Each  $2 \times 2$  block contributes 4 cells plus 2 rows plus 2 columns. We have  $k$  such blocks, and therefore altogether  $8k$  player nodes for cells, rows and columns. In addition, we get 1 from the entire grid, leading to the desired sum. □

**Corollary 5.41.** *For each  $s > 2$ , there exists a configuration with  $s$  situations, two questions and  $s + 2 \lfloor \frac{s}{2} \rfloor$  player nodes in the decision set graph.*

*Proof.* We use the same approach as in the proof of Theorem 5.40. If  $s$  is not a multiple of 4, then we add one incomplete block with the remaining (up to 3) dots as shown in Figure 5.7. The block with one dot increases the sum by 1. The block with 2 dots increases it by 3 (two for the cells and one for the row), and the block with 3 dots increases the sum by 5 (three for the cells, one for the row, one for the column).



For  $s = 4k + 1$ , we therefore get

$$\begin{aligned} 8k + 1 + 1 &= 4k + 1 + 2 \left\lfloor \frac{4k}{2} \right\rfloor + 1 \\ &= 4k + 1 + 2 \left\lfloor \frac{4k + 1}{2} \right\rfloor + 1 \\ &= s + 2 \left\lfloor \frac{s}{2} \right\rfloor + 1 \end{aligned}$$

player nodes.

For  $s = 4k + 2$ , we get

$$\begin{aligned} 8k + 1 + 3 &= 4k + 2 + 2 \left\lfloor \frac{4k}{2} \right\rfloor + 2 \\ &= 4k + 2 + 2 \left\lfloor \frac{4k + 2}{2} \right\rfloor \\ &= s + 2 \left\lfloor \frac{s}{2} \right\rfloor \end{aligned}$$

player nodes.

For  $s = 4k + 3$  finally, we get

$$\begin{aligned} 8k + 1 + 5 &= 4k + 3 + 2 \left\lfloor \frac{4k}{2} \right\rfloor + 2 + 1 \\ &= 4k + 3 + 2 \left\lfloor \frac{4k + 3}{2} \right\rfloor + 1 \\ &= s + 2 \left\lfloor \frac{s}{2} \right\rfloor + 1 \end{aligned}$$

player nodes.

For  $s \not\equiv 2 \pmod{4}$  we therefore even get the strict bound; only for  $s \equiv 2 \pmod{4}$ , this approach results in 1 player node less than would be possible.  $\square$

### Three-dimensional hypercuboids (cuboids)

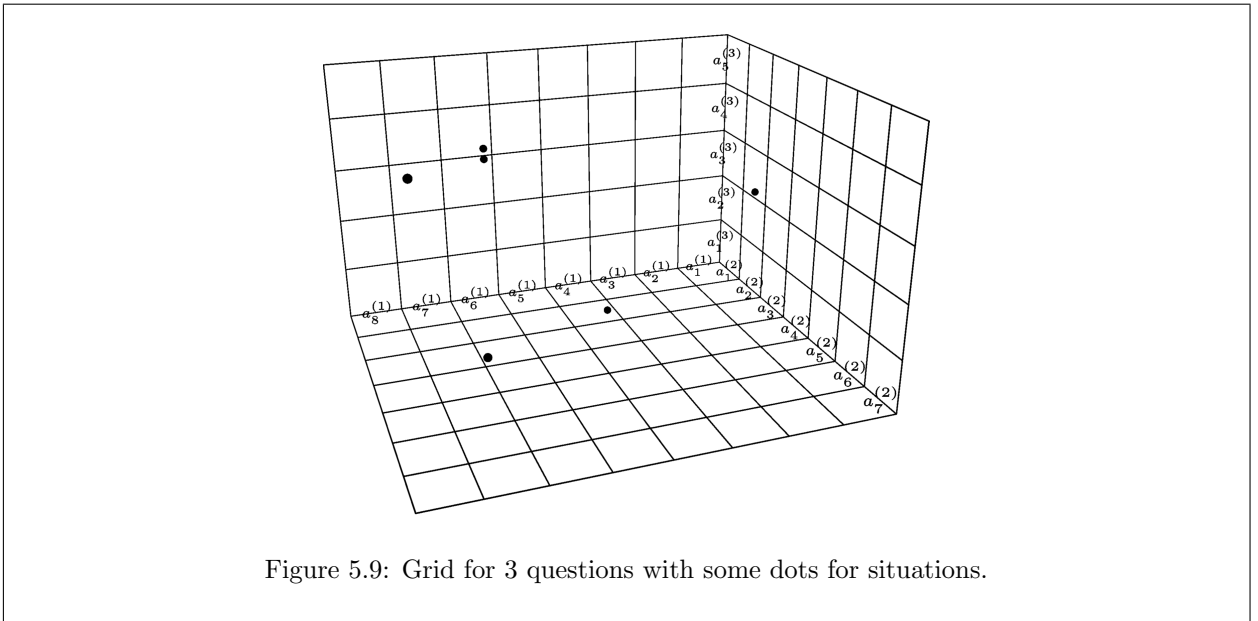
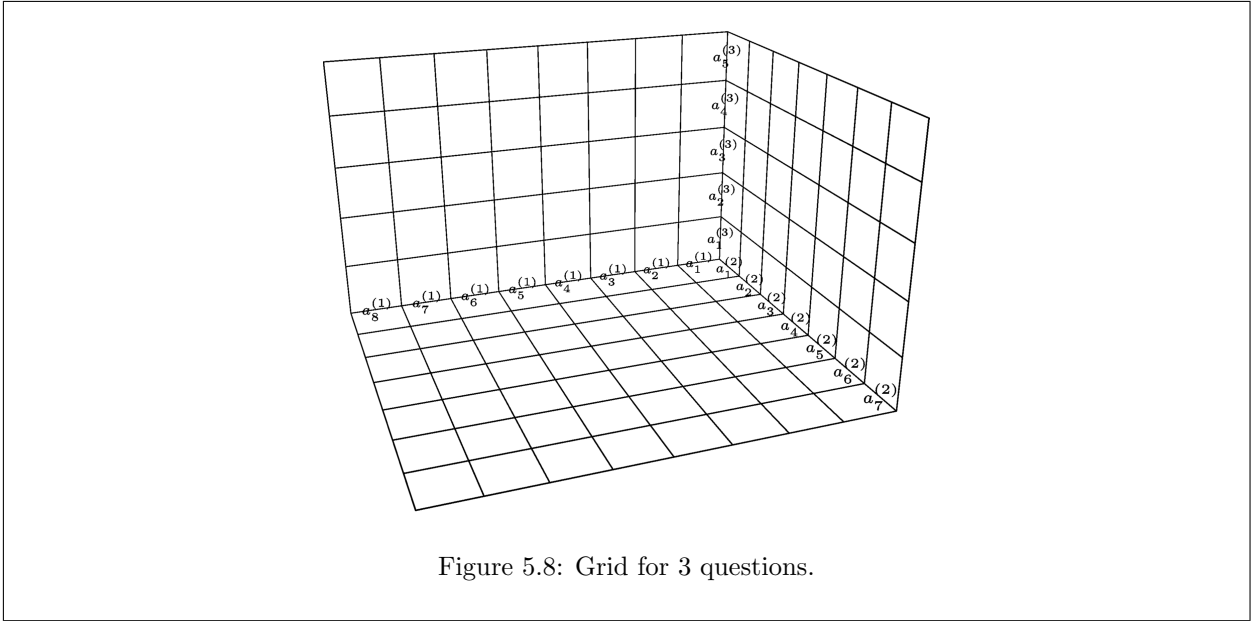
Now that we have sufficiently studied the situation for 2 questions, let us advance to games with 3 questions. This time we mark the answers to question  $q_1$  on the  $x$ -axis, the answers to question  $q_2$  on the  $y$ -axis and the answers to question  $q_3$  on the  $z$ -axis, as shown in Figure 5.8.

Again, for each situation  $s$  we add one point in the cell  $(\mathbf{r}(s, q_1), \mathbf{r}(s, q_2), \mathbf{r}(s, q_3))$ , as shown in Figure 5.9.

As we can see, the three-dimensional image is not very easy to read. We will therefore now stop drawing graphs and just try to imagine things.

Also here we get different subsets depending on the number of questions already asked:

- Again, individual cells correspond to the sets we get after asking all three questions.
- When asking two of the three questions, we get rows (after asking  $q_2$  and  $q_3$ ), columns (after asking  $q_1$  and  $q_3$ ) and pillars (after asking  $q_1$  and  $q_2$ ).
- After asking one question, we get planes: For each answer  $a_l^{(1)}$  to question  $q_1$ , we get the set of all situation dots in the cells with coordinates  $(a_l^{(1)}, *, *)$ , where  $*$  indicates that the coordinate is allowed to have any value. Likewise, we get planes of the form  $(*, a_k^{(2)}, *)$  and  $(*, *, a_m^{(1)})$  after asking only  $q_2$  or  $q_3$ , respectively.
- Finally, before asking any questions, we have the set of all situations in the cuboid.



We therefore need to count the number of different sets that we get that way. We use basically the same algorithm as before, just with one additional step. By “column” we will now mean any set with two fixed answers, and by “plane” we mean any set with one fixed answer. Like before, we will use “cell” to denote a set in which all answers are fixed. Finally, we will use “cuboid” to denote the set before asking any questions.

**Algorithm 5.42.** *Let  $R$  be a data structure that contains sets of remaining situations.  $R$  is initially empty. For each cell: Add the corresponding set to  $R$ . Then, for each column: If the corresponding set has not been added to  $R$  yet, add it to  $R$ . Then, for each plane: If the corresponding set has not been added to  $R$  yet, add it to  $R$ . Then, for the entire cuboid: If the corresponding set has not been added to  $R$  yet, add it to  $R$ .*

We get the same conclusions as in the two-dimensional case:

**Theorem 5.43.** *The number of sets in  $R$  after running Algorithm 5.32 equals the number of player nodes in the decision set graph.*

*Proof.* See argumentation above. □

**Lemma 5.44.** *When adding elements in the order described in Algorithm 5.42, then*

- *a column adds a new set if and only if it contains at least two cells that are non-empty;*
- *a plane adds a new set if and only if it contains at least two rows and two columns that are non-empty (where by “rows” and “columns” we mean columns in different directions); and*
- *the entire cuboid adds a new set if and only if for each question it contains at least two planes that are non-empty.*

*Proof.* The proof is analogous to the one for Lemma 5.34, just with one more dimension. □

**Theorem 5.45.** *The number  $n$  of player nodes in the game tree can be calculated as*

$$\begin{aligned}
 n = & \#[\text{non-empty cells}] \\
 & + \#[\text{columns with at least two non-empty cells}] \\
 & + \#[\text{planes with at least two non-empty rows and at least two non-empty columns}] \\
 & + \begin{cases} 1 & \text{cuboid has at least two non-empty planes for each question} \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

*Proof.* This follows from the definition of Algorithm 5.42. □

### **$q$ -dimensional hypercuboids**

We could now as a next step look at the four-dimensional hypercuboid, but we will skip that step and instead look immediately at the  $q$ -dimensional hypercube.

For each question  $q_l$ , we mark the possible answers to that question on the  $l$ -axis. For each situation  $s$ , we draw a corresponding dot in cell  $(\tau(s, q_1), \tau(s, q_2), \dots, \tau(s, q_q))$ .

Player nodes (and thus sets of remaining situations) in which all  $q$  questions have been asked correspond to cells in the hypercuboid. Sets in which all but one question have been asked correspond to columns, i.e., sets of cells for which all coordinates but one are fixed. Sets in which all but two questions have been asked correspond to planes, i.e., sets of cells for which all but two coordinates are fixed. Sets with all but three questions asked correspond to cuboids, sets with all but 4 questions asked to 4-dimensional hypercuboids, and so on.

In short, situations in which  $q - m$  questions have been asked correspond to  $m$ -dimensional paraxial sub-hypercuboids.

We use  $H\left((i_1, a_{j_1}^{(i_1)}), (i_2, a_{j_2}^{(i_2)}), \dots, (i_m, a_{j_m}^{(i_m)})\right)$  to denote the sub-hypercuboid containing the cells in which the  $i_k$ -coordinate equals  $a_{j_k}^{(i_k)}$  for each  $k$ , or, in other words, the cells that can contain situations from  $\hat{S}\left((q_{i_1}, a_{j_1}^{(i_1)}), (q_{i_2}, a_{j_2}^{(i_2)}), \dots, (q_{i_m}, a_{j_m}^{(i_m)})\right)$ .

We now simply adapt our algorithm correspondingly and generalize our other observations.

**Algorithm 5.46.**

- *Let  $R$  be a data structure that contains sets of remaining situations.  $R$  is initially empty.*
- *For each 0-dimensional paraxial sub-hypercuboid (“cell”): Add the corresponding set to  $R$ .*
- *Then, for each 1-dimensional paraxial sub-hypercuboid (“column”): If the corresponding set has not been added to  $R$  yet, add it to  $R$ .*
- *Then, for each 2-dimensional paraxial sub-hypercuboid (“plane”): If the corresponding set has not been added to  $R$  yet, add it to  $R$ .*

- Then, for each 3-dimensional paraxial sub-hypercuboid: If the corresponding set has not been added to  $R$  yet, add it to  $R$ .
- Then, for each 4-dimensional paraxial sub-hypercuboid: If the corresponding set has not been added to  $R$  yet, add it to  $R$ .
- ...
- Then, for each  $(q - 1)$ -dimensional paraxial sub-hypercuboid: If the corresponding set has not been added to  $R$  yet, add it to  $R$ .
- Then, for the entire cuboid (which is the only  $q$ -dimensional sub-hypercuboid): If the corresponding set has not been added to  $R$  yet, add it to  $R$ .

In the two-dimensional case, we checked whether the entire grid contained at least two non-empty rows and two non-empty columns. In the three dimensional case, we checked whether each plane contained at least two non-empty columns in both possible directions, and whether the entire cuboid contained at least two non-empty planes in each direction.

If a plane (which is two-dimensional) has only one column that contains dots, then one dimension would already be sufficient to contain all dots. If a cuboid (which is three-dimensional) had only one plane that contains dots, then two dimensions would be sufficient to contain all dots. If even all dots in the cuboid are in fact in one column, then one dimension would be sufficient.

We can see that in the two- and three-dimensional cases, we only counted planes/cuboids that indeed contained dots in all two/three dimensions. (We can easily see that, for example, if a cuboid has in on direction only one plane that contains points, then this one plane vice versa already holds all dots of the cuboid, and has already been added when we added all the planes. On the other hand, if we need two planes in each direction, then no plane that contains all points can exist, so it has not been added yet.)

We now generalize this concept as follows:

**Definition 5.47** (actual dimension, full actual dimension). *Let  $M$  be an  $m$ -dimensional paraxial sub-hypercuboid that corresponds to the situation in which all questions except  $\{q_{i_1}, q_{i_2}, \dots, q_{i_m}\}$  have been asked. For each  $k$  with  $1 \leq k \leq m$ , we look at the  $(m - 1)$ -dimensional paraxial sub-hypercuboids of  $M$  in which also  $q_{i_k}$  has been asked – there exists one such sub-hypercuboid for each possible answer to  $q_{i_k}$  – and count how many of them contain at least one situation. (That is, we count how many different answers for  $q_{i_k}$  are still possible.) Let  $a(k)$  be that number.*

*In other words,  $M$  is a sub-hypercuboid of the entire  $q$ -dimensional hypercuboid in which  $q - m$  of the coordinates are fixed. We now fix one additional coordinate, the one corresponding to question  $q_{i_k}$ . We can fix it as any of the answers in  $\overline{A}_{q_{i_k}}$ , and count for how many of these answers the corresponding  $(m - 1)$ -dimensional sub-hypercuboid contains at least one situation dot.*

*Then, we count for how many of the questions the value  $a(k)$  is greater than or equal to 2, that is  $A := |\{k \in \{1, 2, \dots, m\} \mid a(k) \geq 2\}|$ .*

*We call  $A$  the ACTUAL DIMENSION of  $M$ . As we can see, the actual dimension is always smaller than or equal to the dimension of  $M$ .*

*We say that a such a sub-hypercuboid  $M$  has FULL ACTUAL DIMENSION if the actual dimension is equal to the dimension, i.e., if and only if there are at least two non-empty sub-hypercuboids for each additional fixed coordinate.*

*Alternatively, we can say that the sub-hypercuboid  $M$  has FULL ACTUAL DIMENSION if and only if for each unasked question  $q_{i_k}$  there exist at least two situations  $s_1$  and  $s_2$  such that  $\tau(s_1, q_{i_k}) \neq \tau(s_2, q_{i_k})$ .*

*Let  $\hat{S}$  be a set of situations. We count the number of questions  $q_k$  for which there exist two situations  $s_1, s_2 \in \hat{S}$  with  $\tau(s_1, q_k) \neq \tau(s_2, q_k)$ . We call this the ACTUAL DIMENSION of  $\hat{S}$ .*

*This leads to an alternative definition of the actual dimension of a sub-hypercuboid: Let  $M$  be an  $m$ -dimensional sub-hypercuboid, and let  $\hat{S}$  be the set of situations in  $M$ . Then the actual dimension of  $M$  equals the actual dimension of  $\hat{S}$ .*

This requires an example:

**Example 5.48.** We have a game in which the computer chooses an animal from a list, and we can ask three questions: “What class of animal is it?” (mammal, fish, bird, reptile), “How big is it?” (small, medium, large), and “Is it a carnivore?” (carnivore, herbivore).

Without fear of biological incorrectness, the list from which the computer can choose is the following (where size is of course a bit arbitrary):

- Tiger (mammal, large, carnivore)
- Wolf (mammal, medium, carnivore)
- Cat (mammal, small, carnivore)
- Shark (fish, large, carnivore)
- Eagle (bird, medium, carnivore)
- Crocodile (reptile, large, carnivore)
- Plaice (fish, small, herbivore)
- Ostrich (bird, large, herbivore)
- Tortoise (reptile, small, herbivore)

In Figure 5.10, we try to plot them in the three-dimensional space.

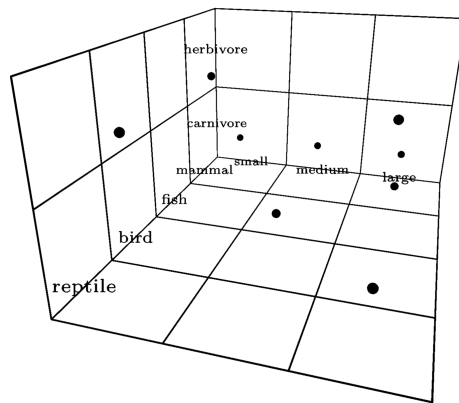


Figure 5.10: Three-dimensional cuboid containing the animals from Example 5.48.

For calculating the actual dimension of the entire 3-dimensional cuboid, we have to look at its 2-dimensional hypercuboids, which are simply planes in this case. We have to do this in each direction; basically we can imagine that we put the cuboid into a CT scan and scan it, layer by layer, in each direction.

Following the description from Definition 5.47, we first fix the first question, “What class of animal is it?”. There are four possible answers to this question. For each answer, we look at the plane in which this answer is fixed. As we can see in Figure 5.11, each of these four planes contains at least one dot. Therefore,  $a(1) = 4$ .

If we fix the answer to the second question, i.e., the size, we get three planes. As we see in Figure 5.12, again all three contain at least one dot, so  $a(2) = 3$ .

Finally, if we fix the answer about food, we get two planes as shown in Figure 5.13. Both contain at least one dot, so  $a(3) = 2$ .

Therefore, the actual dimension is  $A = |\{k \in \{1, 2, 3\} \mid a(k) \geq 2\}| = 3$ , so the cuboid (which also has dimension 3) has full actual dimension.

As another example, let us assume we want to find the full actual dimension of the plane in which the answer to the first question is fixed to “mammal”. This plane is shown as part of Figure 5.11. The questions 2 and 3 can still be fixed, so these are the two we have to look at according to Definition 5.47. For question 2, we can fix it to “small”, “medium” or “large”. The corresponding “sub-hypercuboids” are

in this case the columns of this plane. As we can see, all three of them contain a dot, so  $a(2) = 3$ . For question 3, we can fix the answer as “carnivore” or “herbivore”, and the resulting sub-hypercuboids are the rows of that plane. As we can see, only one of them contains a dot, so  $a(3) = 1$ . Therefore, the actual dimension is  $A = |\{k \in \{2, 3\} \mid a(k) \geq 2\}| = 1$ , so this plane (which has dimension 2, and thus larger than  $A$ ) does not have full actual dimension.

If we had in contrast looked at the plane in which the answer to the first question is fixed as “fish”, then we would have found 2 columns (corresponding to question 2) that contain at least one dot, so  $a(2) = 2$ , and 2 rows (corresponding to question 3) that contain at least one dot, so  $a(3) = 2$ . Therefore,  $A = |\{k \in \{2, 3\} \mid a(k) \geq 2\}| = 2$ , and that plane does have full actual dimension.

Finally, let us look at calculating the actual dimension of one-dimensional sub-hypercuboids of the three-dimensional hypercuboid, which simply are columns. For example, let us calculate the full actual dimension of the column for which the answer to the first question is fixed as “carnivore” and the answer to the second question is fixed as “large”. We can see this column both in Figure 5.13 and in 5.12. There is only question 3 left that we can fix in addition, and we can fix it as one of four values. The corresponding sub-hypercuboids are simple cells, and looking at the column, we see that 3 of the cells are not empty. Therefore,  $a(3) = 3$ , and  $A = |\{k \in \{3\} \mid a(k) \geq 2\}| = 1$ , so the column has full actual dimension.

In contrast, look at the column for which the first question is fixed as “carnivore” and the answer to the second question is fixed as “small”. We can see this column in the same two figures as the previous one. There is again only question 3 left to be fixed in addition, and we can fix it as one of four values. Of the corresponding cells, only one contains a dot. Therefore,  $a(3) = 1$ , and  $A = |\{k \in \{3\} \mid a(k) \geq 2\}| = 0$ , so this column does not have full actual dimension.

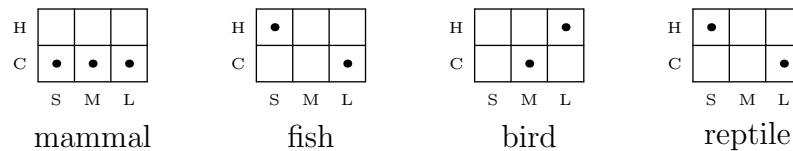


Figure 5.11: Planes of the three-dimensional cuboid from Example 5.48 with fixed answer for class.  $x$ -axis: S = small, M = medium, L = large.  $y$ -axis: C = carnivore, H = herbivore.

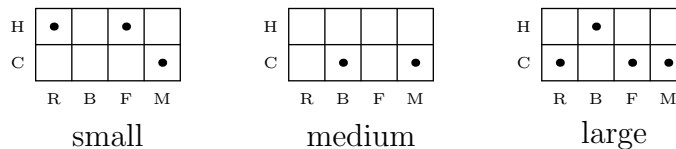


Figure 5.12: Planes of the three-dimensional cuboid from Example 5.48 with fixed answer for size.  $x$ -axis: R = reptile, B = bird, F = fish, M = mammal.  $y$ -axis: C = carnivore, H = herbivore.

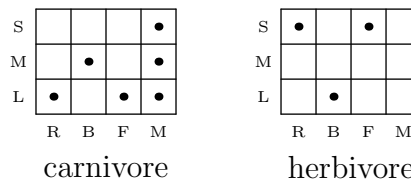


Figure 5.13: Planes of the three-dimensional cuboid from Example 5.48 with fixed answer for food.  $x$ -axis: R = reptile, B = bird, F = fish, M = mammal.  $y$ -axis: L = large, M = medium, S = small.

**Lemma 5.49.** *When adding elements in the order described in Algorithm 5.46, then each  $m$ -dimensional paraxial sub-hypercuboid adds a new set if and only if it has full actual dimension.*

*Proof.* The proof is basically analogous to the one for Lemma 5.34 again.

If there exists at least one question  $q_{i_k}$  such that only the sub-hypercuboid for one of its answers is non-empty, then this set has already been added in the step in which we added all sets for  $(m - 1)$ -dimensional sub-hypercuboids. Otherwise the set has not been added yet, because it has actual dimension  $m$ , and we have not added any sets with actual dimension larger than  $(m - 1)$  yet.  $\square$

**Theorem 5.50.** *The number  $n$  of player nodes in the game tree can be calculated as*

$$\begin{aligned}
 n = & \#[\text{non-empty 0-dimensional paraxial sub-hypercuboids}] \\
 & + \#[\text{1-dimensional paraxial sub-hypercuboids with full actual dimension}] \\
 & + \#[\text{2-dimensional paraxial sub-hypercuboids with full actual dimension}] \\
 & + \#[\text{3-dimensional paraxial sub-hypercuboids with full actual dimension}] \\
 & + \dots \\
 & + \#[\text{(q - 1)-dimensional paraxial sub-hypercuboids with full actual dimension}] \\
 & + \begin{cases} 1 & \text{entire hypercuboid has full actual dimension} \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

*Proof.* This follows from the definition of Algorithm 5.46 and from Lemma 5.49.  $\square$

**Example 5.51.** *We continue Example 5.48 to show how the number of player nodes in the decision set graph can be calculated this way:*

*First, we look at all 0-dimensional paraxial sub-hypercuboids, that is, all cells. 9 of them are not empty.*

*Then we look at all 1-dimensional paraxial sub-hypercuboids, that is, the columns. There are 12 columns in which class and size are fixed, 8 columns in which class and food type are fixed, and 6 columns in which size and food type are fixed. Looking at those independently, we see that only 4 of them have full actual dimension. (We have full actual dimension for the combinations (mammal, carnivore), (large, carnivore), (medium, carnivore), (small, herbivore). Any other combination of two qualities contains at most one animal.*

*Then we look at all 2-dimensional paraxial sub-hypercuboids, that is, the planes. There are 4 planes in which the class is fixed (shown in Figure 5.11), there are 3 planes in which the size is fixed (shown in Figure 5.12), and there are 2 in which the food type is fixed (shown in Figure 5.13). All except the one for mammals have full actual dimension.*

*Finally, the entire cuboid has full actual dimension as shown in Example 5.48.*

*Altogether, this are therefore  $9 + 4 + 8 + 1 = 22$  different sets of remaining situations, and therefore 22 player nodes in the decision set graph.*

As a last note on full actual dimensions, let us show that if a sub-hypercuboid contains 2 dots, this is already sufficient to have full actual dimension, whereas only 1 dot would not be enough:

**Lemma 5.52.** *Let  $M$  be an  $m$ -dimensional paraxial sub-hypercuboid with  $m > 1$ . Then  $M$  can have full actual dimension if and only if it contains at least 2 situation dots.*

*Proof.* That a sub-hypercuboid with only 1 situation dot cannot have full actual dimension is obvious.

We now show that 2 situation dots are sufficient to have full actual dimension. Since each question has at least two answers, we can choose for each question two of its answers and denote them with 0 and 1. Let without loss of generality  $(a_{i_1}^{(1)}, a_{i_2}^{(2)}, \dots, a_{i_{q-m}}^{(q-m)}, *, *, \dots, *)$  be the form of the coordinates of cells in  $M$ , i.e., the first  $(q - m)$  coordinates are fixed to some values.

We can place two situation dots at  $(a_{i_1}^{(1)}, a_{i_2}^{(2)}, \dots, a_{i_{q-m}}^{(q-m)}, 0, 0, \dots, 0)$  and  $(a_{i_1}^{(1)}, a_{i_2}^{(2)}, \dots, a_{i_{q-m}}^{(q-m)}, 1, 1, \dots, 1)$ . Then  $M$  contains for each unanswered question at least two situations that give different answers. Therefore,  $M$  has full actual dimension.  $\square$



### Upper bounds derived from hypercuboids

Now that we have all that theory, we can derive upper bounds from it. A trivial and familiar one first, that had previously been rather laborious to show without using hypercuboids in Theorem 5.23:

**Theorem 5.53.**  $n \leq (1 + a)^q$ .

*Proof.* Let us assume that each question has a different answers. (We can always get this by adding answers even though they do not occur. If anything, this can only increase the number of player nodes, and we are looking for upper bounds anyway.)

Each of the sub-hypercuboids in question contributes at most 1 to the sum in Theorem 5.50. There are  $a^q$  sub-hypercuboids of dimension 1. There are  $\binom{q}{q-1} \cdot a^{q-1}$  sub-hypercuboids of dimension 2. There are  $\binom{q}{q-2} \cdot a^{q-2}$  sub-hypercuboids of dimension 3, and so on.

Altogether, that are

$$\binom{q}{q} \cdot a^q + \binom{q}{q-1} \cdot a^{q-1} + \binom{q}{q-2} \cdot a^{q-2} + \cdots + \binom{q}{2} \cdot a^2 + \binom{q}{1} \cdot a^1 + \binom{q}{0} = (1 + a)^q$$

sub-hypercuboids, and thus at most the same number of player nodes.  $\square$

A definition for convenience:

**Definition 5.54.** Let  $M'$  be a paraxial sub-hypercuboid of the entire hypercuboid for the game. Then we define  $S(M')$  as the set of paraxial sub-hypercuboids of  $M'$ , and  $S(M', d) \subseteq S(M')$  as the set of paraxial sub-hypercuboids of  $M'$  with dimension  $d$ .

For the next upper bound, we make use of the fact that each  $(\hat{q} - k)$ -dimensional sub-hypercuboid is in turn sub-hypercuboid of some  $\hat{q}$ -dimensional sub-hypercuboid. In the three-dimensional cuboid for example, each column is sub-hypercuboid of a plane. Therefore, if instead of counting all  $(\hat{q} - k)$ -dimensional sub-hypercuboids we only count all  $(\hat{q} - 1)$ -dimensional ones *plus their sub-hypercuboids*, then we count each sub-hypercuboid of a smaller dimension also at least once.

As a lemma:

**Lemma 5.55.** Let  $M'$  be a  $\hat{q}$ -dimensional sub-hypercuboid of the entire hypercuboid for the game, then

$$|S(M')| = 1 + \sum_{M'' \in S(M')} 1 \leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} |S(M'')|.$$

*Proof.* This follows directly from the fact that each  $(\hat{q} - k)$ -dimensional sub-hypercuboid is in turn sub-hypercuboid of some  $\hat{q}$ -dimensional sub-hypercuboid. Without loss of generality, let us assume that in  $M'$  the questions  $q_1, q_2, \dots, q_{\hat{q}}$  are still free (and all other questions have fixed answers), and that  $M''$  is a  $(\hat{q} - k)$ -dimensional sub-hypercuboid in which questions  $q_1, q_2, \dots, q_k$  are fixed in addition. Then we define a sub-hypercuboid  $\overline{M}'$  of  $M'$  in which  $q_1$  is fixed to the same answer as in  $M''$ , but  $q_2, q_3, \dots, q_{\hat{q}}$  are free. Then  $\overline{M}'$  is a  $(\hat{q} - 1)$ -dimensional sub-hypercuboid of  $M'$ , and  $M''$  is a sub-hypercuboid of  $\overline{M}'$ .

The only exception is  $k = 0$ , that is,  $M'$  itself (which of course is a sub-hypercuboid of itself). It also is the only  $\hat{q}$ -dimensional sub-hypercuboid, so we just add 1 to the sum for it.  $\square$

Therefore, we can find such upper bounds recursively, and will find an upper bound that we have already seen in Theorem 5.13.

**Theorem 5.56.**  $n \leq q! \cdot a^q \cdot e^{\frac{1}{a}}$ .

*Proof.* We define  $C(\hat{q})$  as an upper bound on the number of sub-hypercuboids that a  $\hat{q}$ -dimensional sub-hypercuboid can have, that is, we demand that for each  $\hat{q}$ -dimensional sub-hypercuboid  $M'$  it holds that  $S(M') \leq C(\hat{q})$ .

From Lemma 5.55 we see that

$$\begin{aligned} |S(M')| &\leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} |S(M'')| \\ &\leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} C(\hat{q} - 1) \\ &= 1 + C(\hat{q} - 1) \cdot |S(M', \hat{q} - 1)|. \end{aligned}$$

We therefore also need a way to calculate or estimate  $|S(M', \hat{q} - 1)|$ . There are  $\hat{q}$  possibilities to choose one additional dimension to fix, and for each of these  $\hat{q}$  ways, there are at most  $\mathbf{a}$  answers to which we can set it. Therefore,  $|S(M', \hat{q} - 1)| \leq \hat{q}\mathbf{a}$ .

Consequently, for each  $\hat{q}$ -dimensional sub-hypercuboid  $M'$  the following inequality holds:

$$\begin{aligned} |S(M')| &\leq 1 + C(\hat{q} - 1) \cdot |S(M', \hat{q} - 1)| \\ &\leq 1 + C(\hat{q} - 1) \cdot \hat{q}\mathbf{a} \end{aligned}$$

Therefore, we can define  $C(\hat{q})$  recursively as

$$\begin{aligned} C(0) &= 1 \\ C(\hat{q}) &= 1 + C(\hat{q} - 1) \cdot \hat{q}\mathbf{a} \end{aligned}$$

and get (for each  $\hat{q}$ -dimensional sub-hypercuboid  $M'$ )

$$|S(M')| \leq C(\hat{q})$$

as desired (because in the argumentation above, we only needed to assume that  $|S(M'')| \leq C(\hat{q} - 1)$  for  $(\hat{q} - 1)$ -dimensional sub-hypercuboids, so we can prove the correctness of the inequality by complete induction).

Solving this recursive equality, we get

$$\begin{aligned} C(\mathbf{q}) &= 1 + \mathbf{q}\mathbf{a} \cdot C(\mathbf{q} - 1) \\ &= 1 + \mathbf{q}\mathbf{a} + \mathbf{q}(\mathbf{q} - 1)\mathbf{a}^2 \cdot C(\mathbf{q} - 2) \\ &= 1 + \mathbf{q}\mathbf{a} + \mathbf{q}(\mathbf{q} - 1)\mathbf{a}^2 + \mathbf{q}(\mathbf{q} - 1)(\mathbf{q} - 2)\mathbf{a}^3 \cdot C(\mathbf{q} - 3) \\ &= \dots \\ &= 1 + \mathbf{q}\mathbf{a} + \mathbf{q}(\mathbf{q} - 1)\mathbf{a}^2 + \mathbf{q}(\mathbf{q} - 1)(\mathbf{q} - 2)\mathbf{a}^3 + \dots + \frac{\mathbf{q}!}{(\mathbf{q} - k + 1)!} \mathbf{a}^{k-1} + \frac{\mathbf{q}!}{(\mathbf{q} - k)!} \mathbf{a}^k \cdot C(\mathbf{q} - k) \\ &= 1 + \mathbf{q}\mathbf{a} + \mathbf{q}(\mathbf{q} - 1)\mathbf{a}^2 + \mathbf{q}(\mathbf{q} - 1)(\mathbf{q} - 2)\mathbf{a}^3 + \dots + \frac{\mathbf{q}!}{1!} \mathbf{a}^{\mathbf{q}-1} + \frac{\mathbf{q}!}{0!} \mathbf{a}^{\mathbf{q}} \cdot C(0) \\ &= 1 + \mathbf{q}\mathbf{a} + \mathbf{q}(\mathbf{q} - 1)\mathbf{a}^2 + \mathbf{q}(\mathbf{q} - 1)(\mathbf{q} - 2)\mathbf{a}^3 + \dots + \frac{\mathbf{q}!}{1!} \mathbf{a}^{\mathbf{q}-1} + \frac{\mathbf{q}!}{0!} \mathbf{a}^{\mathbf{q}} \\ &= \sum_{k=0}^{\mathbf{q}} \frac{\mathbf{q}!}{k!} \mathbf{a}^{\mathbf{q}-k} + \frac{\mathbf{q}!}{0!} \mathbf{a}^{\mathbf{q}} \cdot C(1) \\ &= \mathbf{q}! \mathbf{a}^{\mathbf{q}} \sum_{k=0}^{\mathbf{q}} \frac{\mathbf{a}^{-k}}{k!} \\ &\leq \mathbf{q}! \mathbf{a}^{\mathbf{q}} \sum_{k=0}^{\infty} \frac{\mathbf{a}^{-k}}{k!} \\ &= \mathbf{q}! \mathbf{a}^{\mathbf{q}} e^{\frac{1}{\mathbf{a}}} \end{aligned}$$

□

This seems to be an utterly bad upper bound, but surely there are ways to count in a much smarter way.

In the previous theorem, we only estimated how many sub-hypercuboids  $M'$  can have at all. However, the only ones that are interesting to us are those that also have full actual dimension, since they are the only ones that actually contribute to the number of player nodes. We use the same trick as before, and look only at  $(\hat{q} - 1)$ -dimensional sub-hypercuboids.

In addition, we know from Lemma 5.52 that a sub-hypercuboid can only have full actual dimension if it contains at least two dots. If we fix one question  $q$ , then for each possible answer to  $q$  we get one sub-hypercuboid. However, at most  $\lfloor \frac{\mathbf{a}}{2} \rfloor$  of them can also contain at least 2 dots, and all of them *together* only

contain  $s$ . Therefore, we make the upper bound also dependent on the number of dots that a sub-hypercuboid contains, knowing that a sub-hypercuboid that contains few points cannot contain many sub-hypercuboids with full dimension.

Hopefully, if we use all of these ideas, the mathematical effort (which, as you will see, is not small) might be worth it and produce better upper bounds:

**Theorem 5.57.**  $n \leq 1 + s + s \cdot \frac{q!}{2} \cdot (e + 1)$ .

*Proof.* For each sub-hypercuboid  $M'$  of the game, let  $T(M')$  denote the number of sub-hypercuboids of  $M'$  that have full actual dimension.

Then

$$|T(M')| \leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} |T(M'')| ,$$

using the same argumentation as in Lemma 5.55.

Let  $s(M')$  denote the number of dots in  $M'$ . We define an upper bound  $D(\hat{q}, \hat{s})$  in dependence on the dimension *and* the number of dots, such that  $T(M') \leq D(\hat{q}, s(M'))$  for each  $\hat{q}$ -dimensional sub-hypercuboid  $M'$ .

Therefore

$$\begin{aligned} |T(M')| &\leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} |T(M'')| \\ &\leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} D(\hat{q} - 1, s(M'')) . \end{aligned}$$

We will define  $D(\hat{q}, \hat{s})$  in such a way that it can be expressed as

$$D(\hat{q}, \hat{s}) = \begin{cases} 1 + b_{\hat{q}} \cdot \hat{s} & \hat{s} \geq 2 \\ 0 & \hat{s} \in \{0, 1\} \end{cases} .$$

(Since we only want  $D(\cdot, \cdot)$  to be any upper limit, we can always define it in such a way that it can be expressed at  $1 + b_{\hat{q}} \cdot \hat{s}$ . Additionally, we know from Lemma 5.52 that sub-hypercuboids containing less than two dots cannot have full actual dimension, and if already the entire sub-hypercuboid contains less than two dots, then all sub-hypercuboids of it have at most one dot as well. This justifies setting it to 0 immediately if the number of dots is smaller than two.)

We furthermore know from Theorem 5.38 that for a sub-hypercuboid  $M''$  with two dimensions and  $\hat{s}$  dots,  $T(M'') \leq 2 \lfloor \frac{\hat{s}}{2} \rfloor + 1 \leq \hat{s} + 1$ . We can therefore set  $b_2 = 1$ .

We now need to have a look at the structure of  $S(M', \hat{q} - 1)$ . For each question  $q$  that we can fix in addition to those already fixed in  $M'$ , it contains one sub-hypercuboid for each answer that the question can be set to. In the previous lemma, we concluded that therefore, there are at most  $\hat{q}a$  sub-hypercuboids in  $S(M', \hat{q} - 1)$ .

This time, we split them by question: For each question  $q$  that is not fixed in  $M'$  yet, and for each answer  $a$  in  $\mathcal{A}_q$ , let  $M'(q, a)$  be the sub-hypercuboid of  $M'$  in which question  $q$  has to answer  $a$ . Let for convenience  $\hat{\mathcal{Q}}(M') \subseteq \mathcal{Q}$  denote the set of unfixed situations in  $M'$ . Since  $M'$  has dimension  $\hat{q}$ , this means that  $|\hat{\mathcal{Q}}(M')| = \hat{q}$ . Then

$$S(M', \hat{q} - 1) = \bigcup_{\substack{q \in \hat{\mathcal{Q}}(M') \\ a \in \mathcal{A}_q}} M'(q, a) ,$$

that is, these  $\hat{q}a$  sub-hypercuboids of  $M'$  are exactly the  $(\hat{q} - 1)$ -dimensional sub-hypercuboids that  $M'$  has.

However, we know that for each question, each dot is only contained in exactly one of the sub-hypercuboids for that question, the one that corresponds to that situation's answer. (If in Example 5.48 we take the entire cuboid as  $M'$  and look at all planes in which the question about the class of the animal is fixed, then the tiger appears in exactly one of these planes, the one for mammals.)

Therefore, for any fixed question  $q$  we get

$$\sum_{a \in \mathcal{A}_q} s(M'(q, a)) = s(M') .$$

We now use this in the original inequality that we want to simplify:

$$\begin{aligned}
 |T(M')| &\leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} |T(M'')| \\
 &\leq 1 + \sum_{M'' \in S(M', \hat{q}-1)} D(\hat{q}-1, s(M'')) \\
 &= 1 + \sum_{\substack{q \in \hat{Q}(M') \\ a \in \mathcal{A}_q}} D(\hat{q}-1, s(M'(q, a))) \\
 &= 1 + \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} D(\hat{q}-1, s(M'(q, a))) \\
 &= 1 + \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} \left( \begin{cases} 1 + b_{\hat{q}-1} \cdot s(M'(q, a)) & s(M'(q, a)) \geq 2 \\ 0 & s(M'(q, a)) \in \{0, 1\} \end{cases} \right) \\
 &\leq 1 + \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} \left( b_{\hat{q}-1} \cdot s(M'(q, a)) + \left( \begin{cases} 1 & s(M'(q, a)) \geq 2 \\ 0 & s(M'(q, a)) \in \{0, 1\} \end{cases} \right) \right)
 \end{aligned}$$

Since  $\sum_{a \in \mathcal{A}_q} s(M'(q, a)) = s(M')$ , at most  $\lfloor \frac{s(M')}{2} \rfloor$  of these summands can be greater or equal 2. We use this and continue simplifying:

$$\begin{aligned}
 |T(M')| &\leq 1 + \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} \left( b_{\hat{q}-1} \cdot s(M'(q, a)) + \left( \begin{cases} 1 & s(M'(q, a)) \geq 2 \\ 0 & s(M'(q, a)) \in \{0, 1\} \end{cases} \right) \right) \\
 &\leq 1 + \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} b_{\hat{q}-1} \cdot s(M'(q, a)) + \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} \left( \begin{cases} 1 & s(M'(q, a)) \geq 2 \\ 0 & s(M'(q, a)) \in \{0, 1\} \end{cases} \right) \\
 &\leq 1 + \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} b_{\hat{q}-1} \cdot s(M'(q, a)) + \sum_{q \in \hat{Q}(M')} \left\lfloor \frac{s(M')}{2} \right\rfloor \\
 &= 1 + b_{\hat{q}-1} \cdot \sum_{q \in \hat{Q}(M')} \sum_{a \in \mathcal{A}_q} s(M'(q, a)) + \hat{q} \cdot \left\lfloor \frac{s(M')}{2} \right\rfloor \\
 &= 1 + b_{\hat{q}-1} \cdot \sum_{q \in \hat{Q}(M')} s(M') + \hat{q} \cdot \left\lfloor \frac{s(M')}{2} \right\rfloor \\
 &= 1 + b_{\hat{q}-1} \cdot \hat{q} \cdot s(M') + \hat{q} \cdot \left\lfloor \frac{s(M')}{2} \right\rfloor \\
 &\leq 1 + b_{\hat{q}-1} \cdot \hat{q} \cdot s(M') + \hat{q} \cdot \frac{s(M')}{2} \\
 &\leq 1 + \hat{q} \cdot s(M') \cdot \left( b_{\hat{q}-1} + \frac{1}{2} \right)
 \end{aligned}$$

After all this mathematical hassle, we have therefore now shown that we can simply set

$$\begin{aligned}
 b_2 &= 1 \\
 b_{\hat{q}} &= \hat{q} \cdot \left( \frac{1}{2} + b_{\hat{q}-1} \right)
 \end{aligned}$$

and get an upper limit

$$D(\hat{q}, \hat{s}) = \begin{cases} 1 + b_{\hat{q}} \cdot \hat{s} & \hat{s} \geq 2 \\ 0 & \hat{s} \in \{0, 1\} \end{cases}$$

with

$$T(M') \leq D(d(M'), s(M'))$$

for all sub-hypercuboids  $M'$ , where  $d(M')$  denotes the dimension of  $M'$  and  $s(M')$  denotes the number of dots in  $M'$ .

Solving this recursively, we get

$$\begin{aligned} b_q &= q \cdot \frac{1}{2} + q \cdot b_{q-1} \\ &= q \cdot \frac{1}{2} + q(q-1) \cdot \frac{1}{2} + q(q-1) \cdot b_{q-2} \\ &= \dots \\ &= q \cdot \frac{1}{2} + q(q-1) \cdot \frac{1}{2} + \dots + \frac{q!}{(q-k)!} + \frac{q!}{(q-k)!} \cdot b_{q-k} \\ &= q \cdot \frac{1}{2} + q(q-1) \cdot \frac{1}{2} + \dots + \frac{q!}{2!} + \frac{q!}{2!} \cdot b_2 \\ &= \frac{1}{2} \cdot \sum_{k=2}^{q-1} \frac{q!}{k!} + \frac{q!}{2!} \\ &\leq \frac{q!}{2} \cdot \sum_{k=0}^{\infty} \frac{1}{k!} + \frac{q!}{2!} \\ &\leq \frac{q!}{2} \cdot (e+1), \end{aligned}$$

and thus, adding the up to  $s$  individual cells, at most  $D(q, s) + s = 1 + s + s \cdot \frac{q!}{2} \cdot (e+1)$  player nodes.  $\square$

Well, that did not seem to work too well either; we still have that annoying  $q!$  in the upper bound. Obviously, smarter versions of recursion do not give us significantly better results either, so this kind of recursion might not be the way to go. Let us therefore try something completely different now.

As a next approach, we will count for each point how many sub-hypercuboids with full actual dimension it can potentially part of.

**Theorem 5.58.**  $n \leq 2^{q-1}s + \frac{s}{2}$ .

*Proof.* Any  $m$ -dimensional paraxial sub-hypercuboid  $M'$  for  $m \geq 1$  contributes 1 to the number of player nodes if and only if it has full actual dimension. Let  $M$  be the hypercuboid for the entire game, and let as before  $T(M)$  denote the set of sub-hypercuboids of  $M$  with full dimension, and  $s(M')$  denote the number of situation dots in  $M'$  for every sub-hypercuboid  $M'$ . We now calculate the following: Start with  $x = 0$ . For each sub-hypercuboid  $M'$  of  $M$  that has full dimension (and dimension greater than 0), add the number of situations in  $M$  to  $x$ .

In other words,

$$x = \sum_{M' \in T(M)} s(M') = \sum_{\substack{M' \text{ sub-hypercuboid of } M \\ \text{with full actual dimension}}} [\text{number of situation dots in } M'] .$$

Let us now look at how often a situation  $s$  can be counted that way.  $s$  is contained in a sub-hypercuboid  $M'$  if and only if each coordinate in  $M'$  is either fixed to the value that  $s$  has for that coordinate, or not fixed at all. Therefore, there are two possible states for each of  $q$  questions, and consequently there are altogether at most  $2^q - 1$  sub-hypercuboids of dimension larger than 0 that contain  $s$ . Consequently, each situation can contribute at most  $2^q - 1$  to  $x$ , and we get  $x \leq s \cdot (2^q - 1)$ .

Since sub-hypercuboids with full dimension contain at least two points (by Lemma 5.52), each summand in the formula for  $x$  is at least 2, so  $x$  is at least twice as big as the number of sub-hypercuboids with full actual dimension. The number of player nodes equals the number of sub-hypercuboids with full dimension plus the number of non-empty cells.

We therefore get at most

$$\begin{aligned} n &\leq s + \frac{x}{2} \\ &\leq s + \frac{s \cdot (2^q - 1)}{2} \\ &= s + 2^{q-1}s - \frac{s}{2} \\ &= 2^{q-1}s + \frac{s}{2} \end{aligned}$$

player nodes. □

Finally, similarly to Theorem 5.18 (and the corresponding Example 5.14), we will split the hypercuboid of the game into disjoint sub-hypercuboids and make use of the fact that at most  $\lfloor \frac{s}{2} \rfloor$  of them can contain 2 or more points, and thus have full actual dimension.

**Theorem 5.59.**  $n \leq 2^q \lfloor \frac{s}{2} \rfloor + 2$ .

*Proof.* Any  $m$ -dimensional paraxial sub-hypercuboid  $M$  for  $m > 1$  contributes 1 to the sum if and only if it has full actual dimension. Let us look at sub-hypercuboids in which questions  $q_{i_1}, q_{i_2}, \dots, q_{i_k}$  have been fixed. Then for each combination of answers  $(a_{j_1}^{(i_1)}, a_{j_2}^{(i_2)}, \dots, a_{j_k}^{(i_k)}) \in \overline{\mathcal{A}}_{q_{i_1}} \times \overline{\mathcal{A}}_{q_{i_2}} \times \dots \times \overline{\mathcal{A}}_{q_{i_k}}$ , we get one sub-hypercuboid  $H((i_1, a_{j_1}^{(i_1)}), (i_2, a_{j_2}^{(i_2)}), \dots, (i_k, a_{j_k}^{(i_k)}))$ . These are disjoint, and their union gives the entire hypercuboid.

Therefore, at most  $\lfloor \frac{s}{2} \rfloor$  of them can contain 2 or more situation dots.

There are  $\binom{q}{k}$  ways to choose  $k$  questions to be fixed, and for each such way, we get at most  $\lfloor \frac{s}{2} \rfloor$  sub-hypercuboids with full actual dimension.

If we now do this for all  $0 < k < q$ , we get

$$\begin{aligned} n' &\leq \binom{q}{1} \lfloor \frac{s}{2} \rfloor + \binom{q}{2} \lfloor \frac{s}{2} \rfloor + \dots + \binom{q}{q-1} \lfloor \frac{s}{2} \rfloor \\ &= \lfloor \frac{s}{2} \rfloor \cdot \left( \binom{q}{1} + \binom{q}{2} + \dots + \binom{q}{q-1} \right) \\ &= \lfloor \frac{s}{2} \rfloor \cdot \left( \binom{q}{0} + \binom{q}{1} + \binom{q}{2} + \dots + \binom{q}{q-1} + \binom{q}{q} - \binom{q}{0} - \binom{q}{q} \right) \\ &= \lfloor \frac{s}{2} \rfloor \cdot (2^q - 2) \end{aligned}$$

sets. Now we only need to add 1 for the hypercuboid for the entire game and at most  $s$  for non-empty cells. Altogether, that gives at most

$$\begin{aligned} n &\leq n' + s + 1 \\ &= \lfloor \frac{s}{2} \rfloor \cdot (2^q - 2) + s + 1 \\ &= 2^q \lfloor \frac{s}{2} \rfloor - 2 \lfloor \frac{s}{2} \rfloor + s + 1 \\ &\leq 2^q \lfloor \frac{s}{2} \rfloor - 2 \frac{s-1}{2} + s + 1 \\ &\leq 2^q \lfloor \frac{s}{2} \rfloor - s + 1 + s + 1 \\ &\leq 2^q \lfloor \frac{s}{2} \rfloor + 2 \end{aligned}$$

player nodes. □

These last two upper bound are both similar to the one from Theorem 5.18, but lower by a factor of roughly 2.

Last but not least, we will also try to find some configurations of dots for which the number of player nodes in the decision set graph, which can be calculated as sum of non-empty cells and number of sub-hypercuboids with full dimension, gets close to those upper bounds.

**Theorem 5.60.** *Let  $s = 2^q \cdot b$  be a multiple of  $2^q$  for any integer  $b$ . Then there exists a game with  $q$  questions,  $s$  situations, and  $3^q \cdot b - b + 1$  player nodes in the decision set graph.*

*Proof.* We take the idea with the  $2 \times 2$  boxes from Theorem 5.40 and generalize it to  $q$  dimensions.

We take the first  $2^q$  situations and build a  $2 \times 2 \times \dots \times 2$  hypercube. That is, we assume each question has two answers 0 and 1 and create situations for all combinations  $(c_1, c_2, \dots, c_q)$  with  $c_i \in \{0, 1\}$ . This guarantees that all involved sub-hypercuboids have full actual dimension, since there are two situations in every direction. We therefore count all sub-hyperspaces in which each variable is set either to 0, or to 1, or is not set at all. This gives  $3^q$  sets. From that we subtract the one set in which none of the variables are fixed, i.e., the one for the entire hypercuboid. Altogether, there are  $3^q - 1$  sub-hypercuboids with full actual dimension.

Then, we take the next  $2^q$  situations and repeat this, this time with answers 2 and 3. That is, we create one situation for each  $(c_1, c_2, \dots, c_q)$  with  $c_i \in \{2, 3\}$ . As we can see, the sub-hypercuboids that contain situations from this batch are entirely disjoint from those we counted before. We therefore get another  $3^q - 1$  sets.

We repeat this for the next  $2^q$  situations, and so on, until we have placed all situations.

At the end we can add the one hypercuboid for the entire game again (which we had to subtract for each block to avoid double counting it).

Altogether, we therefore get  $b \cdot (3^q - 1) + 1 = b \cdot 3^q - b + 1$  sets. □

**Corollary 5.61.** *Let  $s = 2^q \cdot b$  be a multiple of  $2^q$  for any integer  $b$ . Then there exist games with  $q$  questions,  $s$  situations and at least  $s \cdot \left(\frac{3}{2}\right)^q - s + 1$  player nodes in the decision set graph.*

*Proof.* As shown above, there exist games with at least  $b \cdot 3^q - b + 1$  player nodes.

$$\begin{aligned} n &\geq b \cdot 3^q - b + 1 \\ &= b \cdot 3^q \cdot \frac{2^q}{2^q} - b + 1 \\ &= b \cdot 2^q \cdot \frac{3^q}{2^q} - b + 1 \\ &= s \cdot \left(\frac{3}{2}\right)^q - b + 1 \end{aligned}$$

We might want to get rid of the  $b$  in the equation, so we can use  $b = \frac{s}{2^q}$  to get

$$n \geq s \cdot \left(\frac{3}{2}\right)^q - b + 1 \geq s \cdot \left(\frac{3}{2}\right)^q - \frac{s}{2^q} + 1$$

as an estimate. □

### 5.3.8 Summary of upper bounds

We found the following upper bounds:

- $n \leq 2^s - 1$
- $n \leq s \cdot q! \cdot e$
- $n \leq q! \cdot a^q \cdot e^{\frac{1}{a}}$
- $n \leq 1 + s + s \cdot \frac{q!}{2} \cdot (e + 1)$
- $n \leq 2^q s$
- $n \leq 2^{q-1} s + \frac{s}{2}$
- $n \leq 2^q s - s + 1$
- $n \leq 2^q \lfloor \frac{s}{2} \rfloor + 2$
- $n \leq 2^{aq}$
- $n \leq (1 + a)^q$
- $n \leq 2^{sq}$
- $n \leq (1 + s)^q$

As we can see, there are basically the following types of upper bounds:

- Those that contain  $2^s$ .
- Those that contain  $q!$  (and are far away from being tight upper bounds).
- Those that contain something of the form  $2^q s$ .
- Those that contain only  $a$  and  $q$ .
- Those in which  $a$  was replaced by  $s$ , which removes  $a$  from the inequality but results in very rough upper bounds.

We found examples for games that produce the following number of player nodes:

- $n = 2^s - 1$
- $n = (1 + a)^q$
- $n = s \cdot \left(\frac{3}{2}\right)^q - \frac{s}{2^q} + 1$

The three interesting pairs we find are:

- $n \leq 2^s - 1$ , and there indeed exist games with  $n = 2^s - 1$ .
- $n \leq (1 + a)^q$ , and there indeed exist games with  $n = (1 + a)^q$ .
- $n \leq s \cdot 2^q - s + 1$ , and there exist games with  $n = s \cdot \left(\frac{3}{2}\right)^q - \frac{s}{2^q} + 1$ .

## 5.4 NP-Hardness

We will now show that solving deduction games is NP-hard. To do so we will look at decision trees, of which deduction games are a more generalized form. Since even the special case of finding an optimal decision tree is known to be NP-hard, deduction games as a generalization thereof are even more so. This approach is interesting because it also shows the close relationship to the field of decision trees (which are commonly used in artificial intelligence), and how to transform a solution of a deduction game back into a decision tree.

In [37], a binary decision tree problem is defined as follows:

**Definition 5.62** (binary decision tree problem [37]). *We have a finite set of objects  $X = \{x_1, x_2, \dots, x_n\}$  and a finite set of tests  $T = \{T_1, T_2, \dots, T_n\}$ . For each test  $T_i \in T$  and each object  $x_j \in X$ , we either have  $T_i(x_j) = \text{true}$  or  $T_i(x_j) = \text{false}$ .*

*The problem is to construct an identification procedure for the objects in  $X$  such that the expected number of tests required to completely identify an element of  $X$  is minimal. An identification procedure is essentially a binary decision tree; at the root and all other internal nodes a test is specified, and the terminal nodes specify objects in  $X$ . To apply the identification procedure one first applies the test specified at the root to the*



unknown object; if it is false one takes the left branch, otherwise the right. This procedure is repeated at the root of each successive subtree until one reaches a terminal node which names the unknown object. Let  $p(x_i)$  be the length of the path from the root of the tree to the terminal node naming  $x_i$ , that is, the number of tests required to identify  $x_i$ . Then the cost of this tree is merely the external path length<sup>1</sup>, that is,  $\sum_{x_i \in X} p(x_i)$ .

Even though the definition only implicitly assumes this, we will furthermore explicitly require that there exist no two objects in  $X$  that have the same answers to all questions in  $T$ , i.e., that are not distinguishable.

That paper then provides the following theorem:

**Theorem 5.63** ([37]). *The binary decision tree problem is NP-complete.*

The similarities to deduction games are fairly obvious – in both we want to find the best order to ask questions –, but really proving that finding optimal decision trees can be reduced (in polynomial time) to solving deduction games takes a bit of work.

**Definition 5.64** (classification game). *Definition 5.62 of a binary decision tree problem already fits nicely into our model. We define situations, questions and answers of a corresponding game as follows:*

- We take  $\mathcal{S} = X = \{x_1, x_2, \dots, x_n\}$  as set of situations, all with weight 1.
- We take  $\mathcal{Q} = T = \{T_1, T_2, \dots, T_n\}$  as questions, and define  $\mathbf{r}(x_j, T_i) = T_i(x_j)$  and  $\mathbf{c}(x_j, T_i) = 1$ .
- We take  $\mathcal{E} = \mathcal{S}$  as the set of estimates with

$$\mathbf{p}(x_i, e_j) = \begin{cases} 0 & x_i = e_j \\ \infty & \text{otherwise} \end{cases}$$

We call this game the CLASSIFICATION GAME (and note that it corresponds to the classification game we had already vaguely defined in Section 2.11).

We see from the definition of question and estimate costs above that

- if there are at least two situations that are still possible, taking an estimate is *never* the best choice, since there is a chance that the penalty is  $\infty$ , and therefore, so is the expected (average) cost; whereas
- if there is only one remaining situation, then taking an estimate is *always* the best choice, since it costs 0 and ends the game whereas any question costs at least 1.

We therefore define reasonable strategies as follows:

**Definition 5.65.** A REASONABLE STRATEGY is a strategy with the following restrictions:

- It is only allowed to take an estimate if there is exactly one remaining situation.
- It is only allowed to ask questions that still can give two different answers.

**Lemma 5.66.** *The average-optimal solution is a reasonable strategy.*

*Proof.* As shown before, taking an estimate when there are still two or more possible situations has expected cost of  $\infty$ . Since all objects are distinguishable, there exists at least one strategy with less than infinite average cost (for example, by simply asking all the questions). Therefore, an average-optimal solution will not use these estimates and therefore fulfills the first restriction.

We know from Corollary 4.26 that questions that have the same answer to all remaining situations are never optimal. (The case that such a question is free does not occur here.) Therefore, all computer nodes that will be used in an average-optimal solution still can give both answers “true” and “false”, and the average-optimal solution also fulfills the second restriction.  $\square$

<sup>1</sup>The external path length of a full binary tree is defined as the sum of the path lengths from the root to each leaf [37].

**Definition 5.67** (strategy subtree). *Consider any strategy in the behavioural game tree of a classification game. The STRATEGY SUBTREE  $T$  is the subtree of the behavioural game tree that contains all nodes that can possibly be reached when playing that strategy. It consequently has the following properties:*

- *The root of the behavioural game tree is included in  $T$ .*
- *For every player node that is included in  $T$ , the child that is chosen in the given strategy is also in  $T$ .*
- *For every computer node in  $T$ , both its children are also included in  $T$ .*
- *There are no nodes contained in  $T$  other than those described above.*

In order to prove that finding average-optimal solutions is NP-complete, we first show the following lemma:

**Lemma 5.68.** *Consider any reasonable strategy in the behavioural game tree of the classification game, and the corresponding strategy subtree  $\hat{T}$ . Then for each situation  $s$ , there exists exactly one estimate leaf in  $\hat{T}$  in which  $s$  is the only remaining situation (and thus also the best estimate).*

*Proof.* First of all we note that the remaining situations in any node are always a subset of the remaining situations in its parent.

Consider any computer node with  $\hat{S}$  as the set of remaining situations. Then the question to be answered there splits  $\hat{S}$  into two disjoint sets  $\hat{S}_1$  and  $\hat{S}_2 = \hat{S} \setminus \hat{S}_1$ . In one child,  $\hat{S}_1$  is the set of remaining situations, in the other it is  $\hat{S}_2$ . Consequently, the left and right subtree after that computer node will always have disjoint sets of remaining situations. Therefore, each situation can only appear as the only remaining situation in either the left or the right subtree of the computer node, but not both.

For player nodes in  $\hat{T}$ , the child that is also in  $\hat{T}$  trivially has the same set of remaining situations as its parent.

Let  $s$  be any situation, then we start at the root, and in each node take the outgoing edge that leads to a node in which  $s$  is also still among the remaining situations. (As we have shown, this edge is unique.) By our construction, this furthermore means that we pass all nodes in which  $s$  is still possible. The tree is finite, therefore this path will eventually end in an estimate leaf.

Therefore there can be at most one leaf in which  $s$  is the only remaining situation, since there is only one leaf in which  $s$  is among the remaining situations at all.

Now we only need to show that in that leaf it will also be the only remaining situation. This, however, follows directly from the restriction that we may only take a guess if there is only one remaining situation.  $\square$

**Theorem 5.69.** *Each reasonable strategy for the behavioural game tree can be transformed into a decision tree, and each decision tree can be transformed into a reasonable strategy.*

*Proof.* Let  $T$  be the strategy subtree of the behavioural game tree for any given reasonable strategy. (See Figure 5.14 and 5.15 for an example of such a tree.) Since each player node has only one outgoing edge in  $T$ , we combine each player node with its child to get the graph  $T'$  depicted in Figure 5.16: If a player node in  $T$  leads to a computer node (i.e., we ask a question), then we merge the player node with that computer node into a “decision node”. We write the question to be asked into the decision node. If a player node leads to taking an estimate, we merge the player node with that estimate leaf and get a “situation leaf”.

This graph  $T'$  therefore is a valid decision tree.

We now show that vice versa, each decision tree  $\tilde{T}'$  can be transformed into a reasonable strategy on the behavioural game tree. Basically, we just take the above steps in reverse order: We replace each situation leaf with a player node that has one outgoing edge leading to an estimate leaf. We replace each decision node with a player node that has exactly one outgoing edge, labeled with the asked question, and leading to a computer node.

The resulting tree  $\tilde{T}''$  is a subtree of the behavioural game tree with the following properties:

- The root of the behavioural game tree is included in  $\tilde{T}''$ .
- For every player node that is included in  $\tilde{T}''$ , exactly one child is in  $\tilde{T}''$ .
- For every computer node in  $\tilde{T}''$ , both its children are also included in  $\tilde{T}''$ .

We can therefore define a strategy on the behavioural game tree as follows: In any player node that is included in  $\tilde{T}''$ , ask the question corresponding to the child that is also in  $\tilde{T}''$ . If we do that, then any player node that is not in  $\tilde{T}''$  will never be reached, so we do not need to define what we would do there.

This strategy is a reasonable strategy: Since every situation leaf in  $\tilde{T}'$  contains only one situation, this also applies to all estimate leaves in  $\tilde{T}''$ , and since each decision node in  $\tilde{T}'$  has two outgoing edges, so does each computer node in  $\tilde{T}''$ .  $\square$

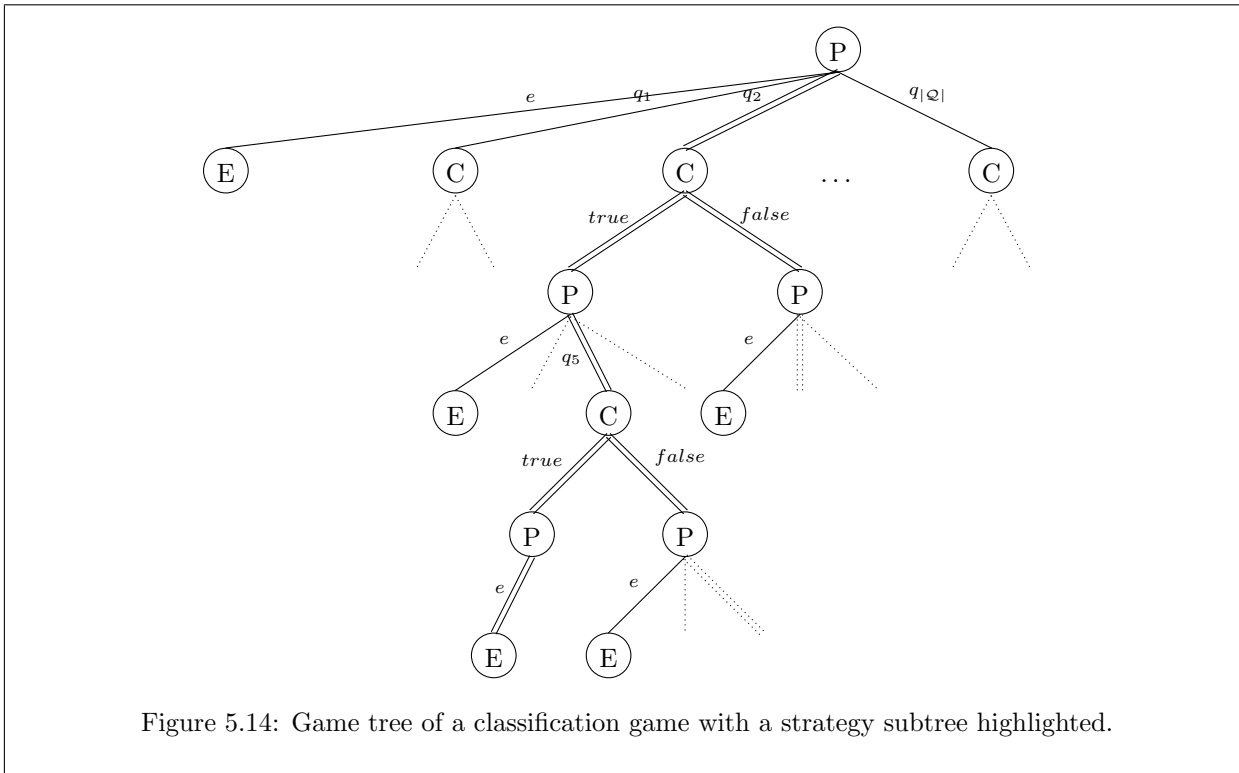


Figure 5.14: Game tree of a classification game with a strategy subtree highlighted.

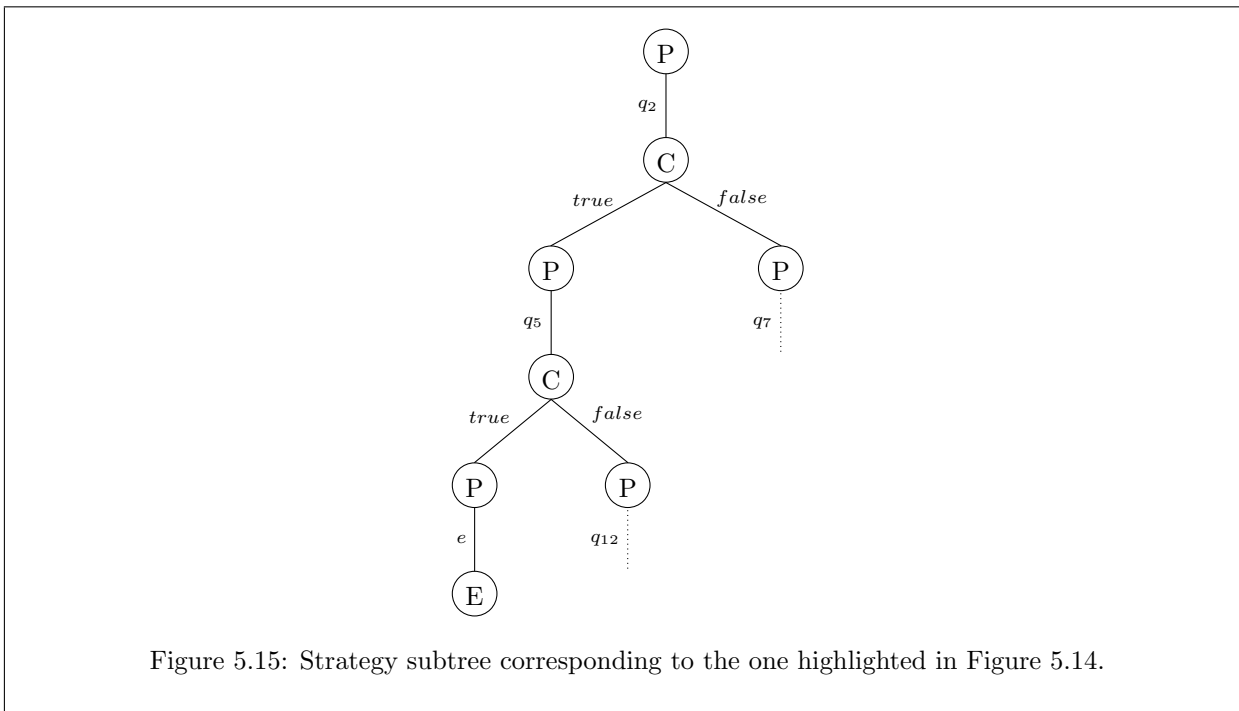


Figure 5.15: Strategy subtree corresponding to the one highlighted in Figure 5.14.

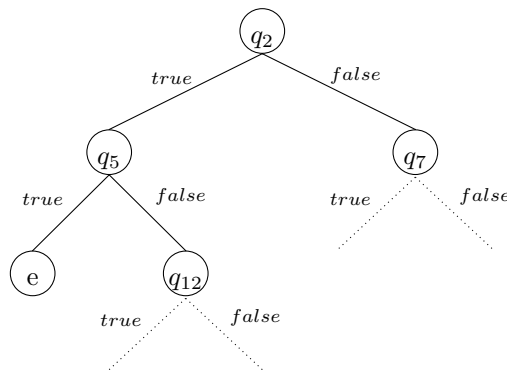


Figure 5.16: Decision tree corresponding to the strategy subtree in Figure 5.15.

Now that we have all necessary preparations, we will show how the problem of finding an optimal decision tree can be reduced to finding average-optimal strategies in a deduction game in polynomial time.

**Corollary 5.70.** *Let  $T$  be the strategy subtree for a reasonable strategy, let  $T'$  be the corresponding decision tree, and let  $T''$  be the strategy subtree derived from  $T'$ . Then  $T'' = T$ .*

*Proof.* This follows directly from the construction. □

**Theorem 5.71.** *Finding an average-optimal strategy is NP-hard.*

*Proof.* We now show that finding an average-optimal solution for the classification game solves the binary decision tree problem, which is NP-complete according to Theorem 5.63. To do so, we will show that the decision tree derived from the average-optimal solution is indeed the optimal decision tree.

First of all, we calculate the average cost in any given reasonable strategy.

Let  $s$  be the number of situations. To calculate the expected average cost, we can simply look at each situation, calculate what the costs are when the computer chooses this situation and we play our average-optimal strategy, and then average the costs of all situations. The average cost is therefore simply the sum of these costs divided by  $s$ . Since  $s$  is constant, the solution for which the average cost is minimal is therefore automatically also the solution for which the sum of these costs is minimal.

According to Lemma 5.68, for each situation  $s$  there exists exactly one estimate leaf in the strategy subtree in which that situation is the only remaining situation. Therefore, the cost in situation  $s$  is the cost of that estimate leaf. However, the cost in an estimate leaf is defined as the number of computer nodes on the path from the root to the node.

We immediately see from our construction of the corresponding decision tree that this is equivalent to the length of the path from the root of the decision tree to the situation leaf corresponding to this estimate leaf. The sum of these costs is therefore equal to the external path length in the decision tree.

Let  $T$  be the strategy subtree for the average-optimal solution, and let  $T'$  be the corresponding decision tree. Let us assume that  $T'$  was not optimal, i.e., there exists a decision tree  $\bar{T}'$  with shorter external path length. Then vice versa the corresponding strategy subtree  $\bar{T}$  would have a lower sum of the costs of the estimate leaves, and therefore also a lower average. This is a contradiction to  $T$  being the strategy subtree for the average-optimal solution.

Transforming the given binary decision tree problem into a classification game takes  $O(s+q)$  time. Transforming the average-optimal solution back into a decision tree takes  $O(\#[\text{nodes in the decision tree}])$  time. However, since the decision tree is a full binary tree with exactly  $s$  leaves, it has  $2s - 1$  nodes. The effort for transforming the result into a decision tree therefore is  $O(s)$ . □

Therefore, finding average-optimal solutions for deduction games is NP-hard. It might be “only” NP-complete, but the author of this thesis is not aware of any way to verify in polynomial time that a given solution is indeed optimal.

# Chapter 6

## Implementation tricks and details

In order to calculate the expected outcome in perfect play, we will implement Algorithm 4.54, which calculates expected costs of the nodes in the decision set graph. (As discussed in the previous chapter, this is equivalent to the four other presented solving algorithms, but contains the least nodes and therefore also requires the least calculation effort.)

However, there are still plenty of implementation tricks for implementing Algorithm 4.54 *efficiently*. We will discuss those in this chapter.

There will be plenty of pseudocode in this chapter, and some parts of it will intentionally be left a bit vague. Rest assured that before the end of the chapter, we will have a final, optimized version of the algorithm in which everything will be explicitly defined. A short summary of the chapter can be found at the end in Section 6.9.

In this entire chapter, let again  $s = |\mathcal{S}|$ ,  $q = |\mathcal{Q}|$ ,  $a_q = |\mathcal{A}_q|$  for each question  $q \in \mathcal{Q}$ , and  $\mathbf{a} = \max_{q \in \mathcal{Q}} a_q$ .

### 6.1 Input

**Definition 6.1** (input). *The input for our algorithm consists of the basic elements described in Definition 1.3, in a slightly modified form:*

- a finite set of possible situations  $\mathcal{S}$ ;
- a finite set of available questions  $\mathcal{Q}$ ;
- a finite set of possible answers  $\mathcal{A} \subset \mathbb{N}$ ;
- for each question  $q \in \mathcal{Q}$  a finite set of available answers  $\mathcal{A}_q \subseteq \mathcal{A}$ ;
- a finite set of possible estimates  $\mathcal{E}$ ;
- a weight function  $\mathbf{w}: \mathcal{S} \rightarrow \mathbb{N}$  describing the probability with which each situation is chosen by the computer;
- an evaluation function  $\mathbf{r}: \mathcal{S} \times \mathcal{Q} \rightarrow \mathcal{A}$  that calculates the answer to a question  $q \in \mathcal{Q}$  in a situation  $s \in \mathcal{S}$ ;
- an evaluation function  $\mathbf{c}: \mathcal{Q} \times \mathcal{A} \rightarrow \mathbb{N}_\infty$  that gives the corresponding cost; and
- an evaluation function  $\mathbf{p}: \mathcal{S} \times \mathcal{E} \rightarrow \mathbb{N}_\infty$  that calculates the penalty for an estimate  $e \in \mathcal{E}$  when the actual situation is  $s \in \mathcal{S}$ .
- *Optionally, we can give two additional evaluation functions:*
  - $\mathbf{p}': 2^{\mathcal{S}} \rightarrow \mathbb{N}_\infty$ , which takes a subset  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  and returns the expected average penalty when choosing the average-optimal estimate, multiplied with the sum of weights in  $\hat{\mathcal{S}}$ . (Equivalently, we can say that  $\mathbf{p}'$  chooses the estimate for which the sum of the penalties for the situations in  $\hat{\mathcal{S}}$  is minimal, and returns this sum.)
  - $\mathbf{p}'': 2^{\mathcal{S}} \rightarrow \mathbb{N}_\infty$ , which takes a subset  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  and returns the worst possible penalty when choosing the worst-case-optimal estimate.

The two main differences to the original definition are that we demand that  $\mathcal{A} \subset \mathbb{N}$ , and that we explicitly demand finite answer sets  $\mathcal{A}_q \subseteq \mathcal{A}$  for each question  $q$  in addition to the common set  $\mathcal{A}$ . In addition, we allow the user to manually provide [efficient implementations of] the functions  $\mathbf{p}'$  and  $\mathbf{p}''$ , even though we could calculate them from  $\mathbf{p}$ . (The multiplication with the sum of weights in the definition of  $\mathbf{p}'$  is necessary to get integers as values.)

**Lemma 6.2.** *The input described in Definition 6.1 is equivalent to the deduction game described in Definition 1.3.*

*Proof.* Let us assume we have a deduction game given as described in Definition 1.3. Since  $\mathcal{A}$  is finite, we can always enumerate  $\mathcal{A}$ ; that is, we assign a unique integer to each possible answer. Let  $\mathcal{A}' \subset \mathbb{N}$  be the set of these integers. Mathematically speaking, we get a bijective mapping  $f: \mathcal{A} \rightarrow \mathcal{A}'$  with an inverse function  $f^{-1}$ . We also see immediately that  $|\mathcal{A}| = |\mathcal{A}'|$ .

Accordingly, we can define  $\mathbf{r}': \mathcal{S} \times \mathcal{Q} \rightarrow \mathcal{A}'$  with  $\mathbf{r}'(s, q) = f(\mathbf{r}(s, q))$ , and  $\mathbf{c}': \mathcal{Q} \times \mathcal{A}' \rightarrow \mathbb{N}_\infty$  with  $\mathbf{c}'(q, a) = \mathbf{c}(q, f^{-1}(a))$ .

Let  $\mathcal{A}'_q = f(\mathcal{A}_q)$  for all questions  $q$ .

Then  $\mathcal{S}$ ,  $\mathcal{Q}$ ,  $\mathcal{A}'$ ,  $\{\mathcal{A}'_q\}_{q \in \mathcal{Q}}$ ,  $\mathcal{E}$ ,  $\mathbf{w}$ ,  $\mathbf{r}'$ ,  $\mathbf{c}'$ ,  $\mathbf{p}$  fulfil all properties required for the input in Definition 6.1.

In each calculation step, we can easily map the values of  $\mathcal{A}'$  back to values in  $\mathcal{A}$ . □

Basically, the above proof is just stating the obvious: even if the answers in the original game have some strange structure, like for example {“mammal”, “fish”, “bird”, “reptile”} in Example 5.48, we can always just distribute (distinct) numbers to them and refer to them by those numbers. To give two examples:

**Example 6.3.** *Let us assume we have a game in which the only two possible answers are “true” and “false”, that is,  $\mathcal{A} = \{\text{true}, \text{false}\}$ . Then we can map “true” to 1 and “false” to 0, so we get  $\mathcal{A}' = \{0, 1\}$  with  $f(\text{true}) = 1$  and  $f(\text{false}) = 0$ .*

**Example 6.4.** *In MasterMind, we have answers of the form “x black y white”. That is, we might have*

$$\mathcal{A} = \{ \begin{array}{l} \text{“0 black 0 white”}, \\ \text{“0 black 1 white”}, \\ \text{“0 black 2 white”}, \\ \text{“0 black 3 white”}, \\ \text{“0 black 4 white”}, \\ \text{“1 black 0 white”}, \\ \text{“1 black 1 white”}, \\ \text{“1 black 2 white”}, \\ \text{“1 black 3 white”}, \\ \text{“2 black 0 white”}, \\ \text{“2 black 1 white”}, \\ \text{“2 black 2 white”}, \\ \text{“3 black 0 white”}, \\ \text{“4 black 0 white”} \end{array} \}$$

*containing those 14 answers that can occur in a MasterMind game with 4 positions.*

One possible mapping would be:

$$\begin{aligned}
 f(\text{"0 black 0 white"}) &= 1 \\
 f(\text{"0 black 1 white"}) &= 2 \\
 f(\text{"0 black 2 white"}) &= 3 \\
 f(\text{"0 black 3 white"}) &= 4 \\
 f(\text{"0 black 4 white"}) &= 5 \\
 f(\text{"1 black 0 white"}) &= 6 \\
 f(\text{"1 black 1 white"}) &= 7 \\
 f(\text{"1 black 2 white"}) &= 8 \\
 f(\text{"1 black 3 white"}) &= 9 \\
 f(\text{"2 black 0 white"}) &= 10 \\
 f(\text{"2 black 1 white"}) &= 11 \\
 f(\text{"2 black 2 white"}) &= 12 \\
 f(\text{"3 black 0 white"}) &= 13 \\
 f(\text{"4 black 0 white"}) &= 14
 \end{aligned}$$

In that case, we would get  $\mathcal{A}' = \{1, 2, \dots, 14\}$ .

An alternative mapping would be  $\bar{f}(\text{"x black y white"}) = 10x + y$ , which leads to:

$$\begin{aligned}
 \bar{f}(\text{"0 black 0 white"}) &= 0 \\
 \bar{f}(\text{"0 black 1 white"}) &= 1 \\
 \bar{f}(\text{"0 black 2 white"}) &= 2 \\
 \bar{f}(\text{"0 black 3 white"}) &= 3 \\
 \bar{f}(\text{"0 black 4 white"}) &= 4 \\
 \bar{f}(\text{"1 black 0 white"}) &= 10 \\
 \bar{f}(\text{"1 black 1 white"}) &= 11 \\
 &\vdots \\
 \bar{f}(\text{"4 black 0 white"}) &= 40
 \end{aligned}$$

Here, we get  $\bar{\mathcal{A}}' = \{0, 1, 2, 3, 4, 10, 11, 12, 13, 20, 21, 22, 30, 40\}$ .

Which mapping we choose depends mostly on the implementation and calculation effort. In this case,  $\bar{f}$  might be the more convenient choice, even though it does not lead to a continuous range of numbers in  $\bar{\mathcal{A}}'$ .

**Lemma 6.5.** *The functions  $\mathbf{p}'$  and  $\mathbf{p}''$  can always be calculated from the other input.*

*Proof.* Given a list of situations, we can calculate the expected cost for the best estimate as described in Algorithm 4.34, and in the case of  $\mathbf{p}'$  multiply it with the sum of the situation weights that can be easily calculated as well.  $\square$

The reason why we might not want to use Algorithm 4.34 is that for most games, there exist much more efficient ways to calculate the best estimate. We therefore want to allow users to provide their own functions for doing this. Therefore, we will use  $\mathbf{p}'$  and  $\mathbf{p}''$  instead of  $\mathbf{p}$  in all future steps, and implement them ourselves the way described above if they are not supplied by the user.

## 6.2 The trivial implementation

Before looking into optimizations, let us describe the most straightforward way to implement the algorithm described in the previous chapter, that is, the algorithm that we get when we just follow the instructions there by the letter. This way to implement it has a lot of performance problems – both memory and runtime requirements are huge –, but it will teach us some things about ways to tweak performance.

Basically we have to do two things: first we need to construct the decision set graph, and then we need to run Algorithm 4.54 on that graph.

### 6.2.1 Constructing the decision set graph

Each node in the decision set graph is uniquely identified by the following information:

- the type of the node, which can be “player”, “computer” or “estimate”;
- a list of remaining situations; and
- for computer nodes, the question that is to be answered.

We will therefore store this information within each node.

In addition, each node has a list of outgoing edges, and each of these edges can be labeled. (For player nodes, the edges will be labeled with the questions they correspond to, for computer nodes they will be labeled with the answers they correspond to.)

Let us therefore assume that we have a data structure `Node` that provides the functions `getType()`, `getSituations()`, `getQuestion()` (for computer nodes), and `getChildren()`. Let us furthermore assume we have constants `PLAYER`, `COMPUTER` and `ESTIMATE`.

We will furthermore use a data structure `Situation` that we assume to provide a function `getWeight() = w(s)` (with  $s = \text{this}$ ), and a data structure `Question` that we assume to provide functions `getAnswer(Situation s) = r(s, q)`, `getAnswerCost(Situation s) = c(s, q)`, `getAnswerCost(integer answer) = c(q, a)`, and `getAvailableAnswers() =  $\mathcal{A}_q$`  (with  $q = \text{this}$ ).

Last but not least, let us assume we have a constant flag `SOLUTIONTYPE` which can be `AVERAGE` or `WORSTCASE`, denoting which type of solution we want to calculate.

By our construction, each node can be reached by following forward edges beginning from the root. (That is, the root is the only source of the graph.) For constructing the decision set graph, we can therefore start with the root, then add all of its children, then recursively add all of their children, and so on.

In somewhat more detail:

**Algorithm 6.6.** *Let  $X$  be a data structure that contains nodes that still need to be processed, i.e., for which we have not calculated their children yet.  $X$  is initially empty.*

*First, create the root as a player node with  $\mathcal{S}$  as the list of remaining situations, and add it to  $X$ .*

*For each node in  $X$ :*

- *If the node is a player node: Let  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  be the set of remaining situations in the node.*

*For each question  $q \in \mathcal{Q}$ , check whether there exist two situations  $s_1, s_2 \in \hat{\mathcal{S}}$  that produce different answers, i.e.,  $r(s_1, q) \neq r(s_2, q)$ .*

- *If yes, then there is an edge that leads to a computer node with  $\hat{\mathcal{S}}$  as set of remaining situations and  $q$  as the question to be asked. Create this computer node, add it to  $X$ , and add an edge (labeled with  $q$ ) from the current node to it.*
- *If no, then there is no outgoing edge for that question.*

*In addition, create an estimate node with  $\hat{\mathcal{S}}$  as its set of remaining situations, and add an edge to it. (Since estimate nodes have no children, we do not need to add it to  $X$ .)*

*Since both computer nodes and estimate leaves have a unique parent, we do not need to check whether it has been created before.*



- If the node is a computer node: Let  $\hat{S} \subseteq \mathcal{S}$  be the set of remaining situations in the node, and let  $q$  be the question to be answered.

For each possible answer  $a \in \mathcal{A}_q$ , check whether there exists a situation  $s \in \hat{S}$  that gives this answer, i.e., such that  $\mathbf{r}(s, q) = a$ .

- If yes, then find the subset  $\hat{S}_a \subseteq \hat{S}$  of situations that give this answer, i.e.,  $\hat{S}_a := \{s \in \hat{S} \mid \mathbf{r}(s, q) = a\}$ . There is an outgoing edge to a player node with  $\hat{S}_a$  as set of remaining situations.
  - \* If this player node exists already, add an edge to it.
  - \* Otherwise, create it, add it to  $X$ , and add an edge from the current node to it.
- If no, then there is no outgoing edge for that answer.

## 6.2.2 Calculate the expected outcome

Once we have the graph, we simply run Algorithm 4.54 on it. This requires two steps: First, we annotate some of the edges in the graph with costs, as described in that algorithm. (We could easily have done that during the creation of the graph already, so we will not present an algorithm for it.) Let us assume we have a function `getEdgeCost(Node A, Node B)` that returns the cost of the edge between nodes A and B. (We will again not go into detail how this is implemented, because it will not be needed any more in later revisions.)

Let us furthermore assume we have a function `getAnswerLabelOnEdge(Node A, Node B)` that returns the label of the edge from a computer node A to its child B, which tells us to which answer the edge corresponds.

Last but not least, let `getWeight(List[Situation] remainingSituations, Question q, Answer a)` be a function that calculates the weight from Algorithm 4.27 (or, equivalently,  $\hat{\mathbf{w}}(\hat{S}, q, a)$  from Algorithm 4.55).

Then, we define the following recursive functions (for calculating average-optimal solutions):

**Algorithm 6.7.** *First of all, we define one function that simply re-routes calculations to the correct function. This will come in handy for player nodes, where it allows us to treat all children the same.*

```

getExpectedFutureCosts(Node N) :=
  if N.getType() == PLAYER then
    return getExpectedFutureCostsPlayerNode(N);
  else if N.getType() == COMPUTER then
    if SOLUTIONTYPE == AVERAGE then
      return getExpectedFutureCostsComputerNodeAverage(N);
    else if SOLUTIONTYPE == WORSTCASE then
      return getExpectedFutureCostsComputerNodeWorstCase(N);
    end if
  else if N.getType() == ESTIMATE then
    return getExpectedFutureCostsEstimateLeaf(N);
  end if

```

*Calculation of player nodes:*

```

getExpectedFutureCostsPlayerNode(Node N) :=
  minimum = infinity;
  for all Node C : N.getChildren() do
    cost = getExpectedFutureCosts(C) + getEdgeCost(C, N);
    minimum = min(minimum, cost);
  end for
  return minimum;

```

*Here,  $\min(x, y)$  simply returns the smaller of the two numbers.*

*Calculation of computer nodes for average-optimal solutions:*

```

getExpectedFutureCostsComputerNodeAverage(Node N) :=
  sum = 0;
  sumweights = 0;
  for all Node C : N.getChildren() do
    cost = getExpectedFutureCosts(C) + getEdgeCost(C, N);
    weight = getWeight(N.getSituations(), N.getQuestion(), getAnswerLabelOnEdge(C, N));
    sum = sum + cost*weight;
    sumweights = sumweights + weight;
  end for
  return sum / sumweights;

```

*Calculation of computer nodes for worst-case-optimal solutions:*

```

getExpectedFutureCostsComputerNodeWorstCase(Node N) :=
  maximum = 0;
  for all Node C : N.getChildren() do
    cost = getExpectedFutureCosts(C) + getEdgeCost(C, N);
    maximum = max(cost, maximum);
  end for
  return maximum;

```

*Here,  $\max(x, y)$  simply returns the larger of the two numbers.*

*Calculation of estimate leafs:*

```

getExpectedFutureCostsEstimateLeaf(Node N) :=
  return 0;

```

*For finding the expected outcome in perfect play, we simply calculate `getExpectedFutureCosts(root)`.*

### 6.2.3 A small optimization

We said before that we would not optimize this algorithm too much since the whole approach is horribly ineffective, but we will pick one low-hanging fruit that we will also reuse in future implementations.

If the decision set graph were a tree, the above approach would calculate each node exactly once. However, the decision set graph is far from being a tree – as shown in Theorem 4.23, given any list of (question, answer) tuples, we can ask these questions (and demand the corresponding answers) in any order and will always end up at the same node in the decision set graph. Nodes in the decision set graph do not strictly correlate to numbers of questions asked, but if there is one way to reach a node by asking  $q$  questions, then there are at least  $q! - 1$  more such ways. If we used the algorithm above, the node would therefore be calculated at least  $q!$  times.

In fact, the above algorithm completely ignores all the nice properties that we introduced decision sets for in the first place, and behaves almost identical to Algorithm 4.36 that solved the game on the behavioural game tree. Thus, it calculates the same number of nodes as Algorithm 4.36 would, and as we have seen in Theorem 5.13, there are lots of them.

A long story cut short, this is an excellent moment to use some dynamic programming. After all, we introduced the entire concept of decision set graphs for that very purpose.

In practice, all we need to do is remember nodes we have already calculated. From the structure of the decision set graph we know that each computer node and estimate leaf has exactly one parent. It is therefore sufficient to store known results only for player nodes: if we guarantee that each player node is

calculated only once, then also their children are calculated only once (since those children do not have any other parents).

We therefore only need a simple modification to `getExpectedFutureCostsPlayerNode(node N)`. Changed parts of the code are highlighted in grey.

**Algorithm 6.8.** *Let `KnownResults` be a data structure that stores (node, value) tuples, where node can be any player node in the decision set graph, and value is the expected future cost in that node.*

```

getExpectedFutureCostsPlayerNode(Node N) :=
  if KnownResults.contains(N) then
    return KnownResults.get(N);
  end if
  minimum = infinity;
  for all Node C : N.getChildren() do
    cost = getExpectedFutureCosts(C) + getEdgeCost(C, N);
    minimum = min(minimum, cost);
  end for
  KnownResults.put(N, minimum);
  return minimum;

```

Since the decision set graph is acyclic, the recursive calls to `getExpectedFutureCosts(...)` that happen as part of the calculation of `getExpectedFutureCosts(node N)` will never have `N` as argument again. Therefore, for each node `N`, the calculation is done exactly once, and all future calls will return the stored result.

We will however see in Section 6.7 that in practice, we might not always have the storage space to actually store all known results.

### 6.3 Implicit decision set graph

We can see several things from the implementation above. First of all, we do not really use the entire graph structure we constructed in the first step; the only thing we use it for during the calculation is to find the children of a node.

However, looking at the construction step of the graph, we can see that the list of a node's children can always be constructed from the information contained in the node itself, that is, from the node type, the list of remaining situations and (in case of computer nodes) the question to be asked. Furthermore, when using Algorithm 6.8, we need the list of children of a node only a single time for each node. Therefore, we do not really need to create all these lists beforehand, but instead can just create them as we go.

Also, we already mentioned that storing the expected future costs for each node might require more memory than we have, yet so far suggested to construct the entire graph and keep it in memory.

In short, instead of actually constructing the entire graph, we want to generate it “on the fly”, as needed, and forget about it again as soon as we do not need it any more.

An alternative way to see it is this: Mathematically speaking, the decision set graph of course always “exists” as a mathematical construct – given a game in the above representation, the decision set graph is uniquely defined. However, instead of explicitly creating a computer representation of this graph by creating objects for nodes and edges, we want to just implicitly refer to the node of the graph we are currently in.

As stated before, the type of a node, list of remaining situations and (for computer nodes) question to be answered are enough to uniquely identify a node in the graph. Given some values for these properties, we therefore know exactly which node we are in, and fortunately we can also immediately get the list of children from this information.

**Example 6.9.** *Just to give an example, let us assume we have a game with 10 situations  $\mathcal{S} = \{1, 2, 3, \dots, 10\}$  and 3 questions  $\mathcal{Q} = \{a, b, c\}$ . Each question can give answers “true”, “false”, or “maybe”. Let us assume that*

- *Question a answers “true” for situations 1, 2, 3, 4 and 5, and “false” for 6, 7, 8, 9 and 10 (so question a could also be described “Is the number smaller than 6?”);*
- *Question b answers “true” for 1, 3, 5, 7 and 9, and “false” for 2, 4, 6, 8 and 10 (so question b could also be described “Is the number even?”); and*
- *Question c answers “true” for 1, 3, and 7, “maybe” for 2, 6, 8 and 9, and “false” for 4, 5 and 10 (for which there does not exist any easy description).*

*Then “player node in which 2, 4, 6, 8, 10 are still possible” uniquely identifies one player node (which we might, for example, reach by asking question b and getting answer “false”).*

*We know that this node has one computer node as child for each question that gives at least two different answers, plus one estimate node. Let us therefore check the questions one after the other and decide whether we have to create a child for each of them:*

- *Question a returns “true” for 2 and “false” for 10, so there are two answers that are still possible. Therefore we get one child denoted by “computer node in which 2, 4, 6, 8, 10 are still possible, and in which we have to answer a”.*
- *Question b returns “false” for all remaining situations. Therefore, there is no child corresponding to this question.*
- *Question c returns “maybe” for 2 and “false” for 4, therefore there are again two different answers that are still possible, and we get one child denoted by “computer node in which 2, 4, 6, 8, 10 are still possible, and in which we have to answer c”.*
- *We have one additional child denoted by “estimate leaf in which 2, 4, 6, 8, 10 are still possible”.*

*Let us now look at the “computer node in which 2, 4, 6, 8, 10 are still possible, and in which we have to answer c”. Here, we get one child for each answer that can still occur:*

- *The answer “true” does not occur, therefore we do not get a child for it.*
- *The answer “maybe” is given for 2, 6 and 8, therefore we get one child denoted by “player node in which 2, 6 and 8 are still possible”.*
- *The answer “false” is given for 4 and 10, therefore we get one child denoted by “player node in which 4 and 10 are still possible”.*

*Other things we might notice in this example include:*

- *In order for a computer node to be created, we need to see at least two different answers, but it is not necessary that all available answers can still occur.*
- *Not all situations must be distinguishable at all. For example, situations 1 and 3 both give “true” as answer to all three questions.*
- *Computer nodes and estimate nodes always have exactly the same set of remaining situations as their parents. (By our construction, computer nodes and estimate nodes also always have exactly one parent.)*
- *The children of a computer node have disjoint sets of remaining situations, and their union is the set of remaining situations in the computer node.*

Now we just need to reflect this in our algorithm:

**Algorithm 6.10.** *We introduce functions `calculatePlayerNodeChildren(Node N)` and `calculateComputerNodeChildren(Node N)` that calculate the list of children as described above. Along with the description of the child itself, they will also store for each child the label of the edge leading there, and the cost associated with that edge (which is the cost for the corresponding answer for edges from computer to player nodes, 0 for edges from player to computer nodes, and the expected estimate penalty for edges from player to estimate nodes). That is, they return a list of instances of some data structure `Child` which provides the following functions:*

- *`getChildNode()` which returns the [description of the] child node;*

- `getEdgeCost()` which returns the cost of the edge leading to that child; and
- `getAnswerLabel()` which (for computer nodes) returns the answer that the edge leading to this child corresponds to.

For simplicity, if `C` is an instance of `Child` we will sloppily use `C` and `C.getChildNode()` interchangeably, so that we do not have to explicitly write `C.getChildNode()` every time. (That is, we will, for example, simply write `C.getSituations()` instead of `C.getChildNode().getSituations()`, and write `getCosts(C)` instead of `getCosts(C.getChildNode())` for recursively calculating the costs of `C`.)

As we can see, we now do not need the functions `getEdgeCost(...)` and `getAnswerLabelOnEdge(...)` any more, and directly take these stored values instead.

The updated algorithm looks like this:

```

getExpectedFutureCostsPlayerNode(Node N) :=
  if KnownResults.contains(N) then
    return KnownResults.get(N);
  end if
  minimum = infinity;
  for all Child C : calculatePlayerNodeChildren(N) do
    cost = getExpectedFutureCosts(C) + C.getEdgeCost();
    minimum = min(minimum, cost);
  end for
  KnownResults.put(N, minimum);
  return minimum;

```

```

getExpectedFutureCostsComputerNodeAverage(Node N) :=
  sum = 0;
  sumweights = 0;
  for all Child C : calculateComputerNodeChildren(N) do
    cost = getExpectedFutureCosts(C) + C.getEdgeCost();
    weight = getWeight(N.getSituations(), N.getQuestion(), C.getAnswerLabel());
    sum = sum + cost*weight;
    sumweights = sumweights + weight;
  end for
  return sum / sumweights;

```

Note that some of these functions also need to access the list of questions  $\mathcal{Q}$ , the lists of available answers  $\mathcal{A}_q$ , and the evaluation functions  $\tau$ ,  $c$  and  $p'$ . Since those are all constant, they are not explicitly listed as parameters.

The equivalent modifications to the algorithm for a worst-case-optimal solution are quite straightforward:

```

getExpectedFutureCostsComputerNodeWorstCase(Node N) :=
  maximum = 0;
  for all Node C : calculateComputerNodeChildren(N) do
    cost = getExpectedFutureCosts(C) + C.getEdgeCost();
    maximum = max(cost, maximum);
  end for
  return maximum;

```

## 6.4 Simplified weights

The next thing to notice for refining this algorithm is that the weights have a special structure. We look back to Algorithm 4.27, which defined how weights are calculated, i.e., what is returned by `getWeight(List[Situation] remainingSituations, Question q, Answer a)`.

We make the following rather trivial observation:

**Lemma 6.11.** *Let  $C$  be a computer node, and let  $\hat{\mathcal{S}}$  be the set of remaining situations in  $C$ . Then the sum of the weights of its children, calculated with Algorithm 4.27, equals the sum of the weights of the situations in  $\hat{\mathcal{S}}$ .*

*Mathematically speaking: Let  $q$  be the question to be asked in  $C$ , let  $\mathfrak{w}$  be the given weight function for the situations, and let  $\hat{\mathfrak{w}}$  be the weight function defined in Algorithm 4.27. Then*

$$\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}_a = \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s)$$

*Proof.* Let  $\hat{\mathcal{S}}((q, a))$  be the subset of  $\hat{\mathcal{S}}$  for which  $a$  is the answer to  $q$ , that is,  $\hat{\mathcal{S}}((q, a)) = \{s \in \hat{\mathcal{S}} \mid \mathfrak{r}(s, q) = a\}$ . It is known that these sets are disjoint, and that their union is  $\mathcal{S}$ :

$$\begin{aligned} \hat{\mathcal{S}}((q, a_x)) \cap \hat{\mathcal{S}}((q, a_y)) &= \emptyset & (x \neq y) \\ \bigcup_{a \in \mathcal{A}_q} \hat{\mathcal{S}}((q, a)) &= \mathcal{S} \end{aligned}$$

Therefore, we can simply apply this to the sum that we get when substituting the definition of  $\hat{\mathfrak{w}}_a$ , and get the desired equality:

$$\begin{aligned} \sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}_a &= \sum_{a \in \mathcal{A}_q} \sum_{\substack{s \in \hat{\mathcal{S}} \\ \mathfrak{r}(s, q) = a}} \mathfrak{w}(s) \\ &= \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \end{aligned}$$

□

Let us have a closer look at the code for calculating the expected future costs of a computer node:

```
getExpectedFutureCostsComputerNodeAverage(Node N) :=
  sum = 0;
  sumweights = 0;
  for all Child C : calculateComputerNodeChildren(N) do
    cost = getExpectedFutureCosts(C) + C.getEdgeCost();
    weight = getWeight(N.getSituations(), N.getQuestion(), C.getAnswerLabel());
    sum = sum + cost*weight;
    sumweights = sumweights + weight;
  end for
  return sum / sumweights;
```

Instead of calculating `sumweights` as the sum of the weights of the children, we could therefore calculate this sum of weights directly from the set of remaining situations.

On the other hand, we see that after retrieving the expected future costs of a child, we multiply that cost with the weight of that child again. However, the weight of the child is exactly the sum of weights of the remaining situations in that child, and we probably just divided by that very value one step earlier.

In short, it seems that we spend a fair amount of time dividing by something and then multiplying again with the exact same value. Even if we can do this losslessly – which by the way is possible in this case by using fractions instead of floating point numbers –, it still is quite a waste of time.

Fortunately, since the list of remaining situations is known in each node, so is the sum of their weights. Calculating the expected future costs of a node is therefore equivalent to calculating the expected future costs *times the sum of the weights of the remaining situations* – we can easily (and losslessly) transform one value into the other.

Currently, we calculate expected future costs of a computer node as described in Algorithm 4.55. Based on that, let us derive how we could calculate the expected future costs times sum of weights efficiently:

**Lemma 6.12.** *Sticking with the notation used in Algorithm 4.55, let  $f(N)$  be the expected future cost of a node  $N$ , and let  $x(X, Y)$  denote the cost of the edge between  $X$  and  $Y$ .*

*Let furthermore  $\bar{\mathfrak{w}}(\hat{\mathcal{S}})$  denote the sum of the weights of the situations in  $\hat{\mathcal{S}}$ , that is,*

$$\bar{\mathfrak{w}}(\hat{\mathcal{S}}) := \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) .$$

*Let now  $g(N)$  denote the product of expected future cost and sum of the weights of the remaining situations of node  $N$ . That is, if  $\hat{\mathcal{S}}(N)$  denotes the set of remaining situations in  $N$ , then*

$$g(N) := f(N) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N))$$

*for each node  $N$ .*

*Then the following equalities hold:*

- *If  $N$  is an estimate leaf, then*

$$g(N) = 0 .$$

- *If  $N$  is a player node, then*

$$g(N) = \min_{C \in \text{children}(N)} \left( g(C) + \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \cdot x(N, C) \right) .$$

- *Let  $N$  be a computer node, and let  $q$  be the question to be answered. For each possible answer  $a \in \mathcal{A}_q$ , let  $C(a)$  be the child of  $N$  that corresponds to  $a$ .*

- *If we are looking for an average-optimal solution, then*

$$g(N) = \sum_{a \in \mathcal{A}_q} \left( g(C(a)) + \bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot x(N, C(a)) \right) .$$

- *If we are looking for a worst-case-optimal solution, then*

$$g(N) = \max_{a \in \mathcal{A}_q} \left( \frac{g(C(a))}{\bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a)))} + x(N, C(a)) \right) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) .$$

*Proof.* We will get all of this by simply substituting the definitions from Algorithm 4.55.

- *If  $N$  is an estimate leaf:*

$$\begin{aligned} g(N) &= f(N) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= 0 \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= 0 \end{aligned}$$

- *If  $N$  is a player node:*

$$\begin{aligned} g(N) &= f(N) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= \min_{C \in \text{children}(N)} (f(C) + x(N, C)) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= \min_{C \in \text{children}(N)} \left( f(C) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) + x(N, C) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \right) \\ &= \min_{C \in \text{children}(N)} \left( g(C) + \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \cdot x(N, C) \right) . \end{aligned}$$

- Let  $N$  be a computer node, and let  $q$  be the question to be answered. For each possible answer  $a \in \mathcal{A}_q$ , let  $C(a)$  be the child of  $N$  that corresponds to  $a$ .
  - If we are looking for an average-optimal solution, we will use that

$$\begin{aligned}\bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) &= \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s), \\ \sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}_a &= \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s)\end{aligned}$$

and

$$\hat{\mathfrak{w}}(\hat{\mathcal{S}}(N), q, a) = \bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))).$$

This gives us the desired equality:

$$\begin{aligned}g(N) &= f(N) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= \frac{\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}(N), q, a) \cdot (f(C(a)) + x(N, C(a)))}{\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}, q, a)} \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= \frac{\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}(N), q, a) \cdot (f(C(a)) + x(N, C(a)))}{\sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s)} \cdot \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \\ &= \sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}(N), q, a) \cdot (f(C(a)) + x(N, C(a))) \\ &= \sum_{a \in \mathcal{A}_q} \bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot (f(C(a)) + x(N, C(a))) \\ &= \sum_{a \in \mathcal{A}_q} (\bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot f(C(a)) + \bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot x(N, C(a))) \\ &= \sum_{a \in \mathcal{A}_q} (g(C(a)) + \bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot x(N, C(a)))\end{aligned}$$

- If we are looking for a worst-case-optimal solution, we simply get the following:

$$\begin{aligned}g(N) &= f(N) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= \max_{a \in \mathcal{A}_q} (f(C(a)) + x(N, C(a))) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\ &= \max_{a \in \mathcal{A}_q} \left( \frac{g(C(a))}{\bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a)))} + x(N, C(a)) \right) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N))\end{aligned}$$

□

Thus, we can calculate  $g(X)$  for all nodes  $X$  directly, without ever looking at  $f(X)$ . To get the expected future cost in the root  $R$ , we now calculate  $\frac{g(R)}{\bar{\mathfrak{w}}(\hat{\mathcal{S}}(R))}$ .

We can easily turn this into an algorithm:

**Algorithm 6.13.** Let `getWeight(List[Situation] situations)` be a function that calculates the sum of weights of the given situations.

Since function names like `getExpectedFutureCostsTimesWeightForPlayerNode(...)` would eventually become really impractical, let us just call these new functions `calculateCosts(...)`, `calculateCostsPlayerNode(...)`, `calculateCostsComputerNodeAvg(...)`, `calculateCostsComputerNodeWorst(...)`, and `calculateCostsEstimateLeaf(...)` from now on (which is still easily too long).



```

calculateCosts(Node N) :=
  if N.getType() == PLAYER then
    return calculateCostsPlayerNode(N);
  else if N.getType() == COMPUTER then
    if SOLUTIONTYPE == AVERAGE then
      return calculateCostsComputerNodeAvg(N);
    else if SOLUTIONTYPE == WORSTCASE then
      return calculateCostsComputerNodeWorst(N);
    end if
  else if N.getType() == ESTIMATE then
    return calculateCostsEstimateLeaf(N);
  end if

```

*Calculation of player nodes:*

```

calculateCostsPlayerNode(Node N) :=
  if KnownResults.contains(N) then
    return KnownResults.get(N);
  end if
  minimum = infinity;
  for all Child C : calculatePlayerNodeChildren(N) do
    cost = getCosts(C) + getWeight(N.getSituations()) * C.getEdgeCost();
    minimum = min(minimum, cost);
  end for
  KnownResults.put(N, minimum);
  return minimum;

```

*In the next function for calculating average-optimal computer nodes, we need to delete several parts. We therefore print it twice, once with these parts struck out, once with them simply removed:*

```

calculateCostsComputerNodeAvg(Node N) :=
  sum = 0;
  sumweights = 0;
  for all Child C : calculateComputerNodeChildren(N) do
    cost = getCosts(C) + getWeight(C.getSituations()) * C.getEdgeCost();
    weight = getWeight(N.getSituations(), N.getQuestion(), C.getAnswerLabel());
    sum = sum + cost*weight;
    sumweights = sumweights + weight;
  end for
  return sum /sumweights;

```

```

calculateCostsComputerNodeAvg(Node N) :=
  sum = 0;
  for all Child C : calculateComputerNodeChildren(N) do
    cost = getCosts(C) + getWeight(C.getSituations()) * C.getEdgeCost();
    sum = sum + cost;
  end for
  return sum;

```

Computer nodes for worst-case-optimal solutions get somewhat ugly now. We will assume that we have a data type `fraction` that stores fractions of integers, and that we can use this for all intermediate steps and returned results.

```

calculateCostsComputerNodeWorst(Node N) :=
  maximum = 0;
  for all Node C : calculateComputerNodeChildren(N) do
    cost = getCosts(C) / getWeight(C.getSituations()) + C.getEdgeCost();
    maximum = max(cost, maximum);
  end for
  return maximum * getWeight(N.getSituations());

```

Finally, estimate leafs still work the same way as before:

```

calculateCostsEstimateLeaf(Node N) :=
  return 0;

```

The expected future cost in the root `root` is now `getCosts(root) / getWeight(root.getSituations())`.

As we can see in Table 6.1, the equations for average-optimal solutions are a little prettier now than in the original.

	Algorithm 4.54 $f(N) :=$	Algorithm 6.13 $g(N) :=$
estimate	0	0
player	$\min_{C \in \text{children}(N)} (f(C) + x(N, C))$	$\min_{C \in \text{children}(N)} (g(C) + \bar{w}(\hat{S}(N)) \cdot x(N, C))$
computer (avg)	$\frac{\sum_{a \in \mathcal{A}_q} \hat{w}(\hat{S}, q, a) \cdot (f(C(a)) + x(N, C(a)))}{\sum_{a \in \mathcal{A}_q} \hat{w}(\hat{S}, q, a)}$	$\sum_{a \in \mathcal{A}_q} (g(C(a)) + \bar{w}(\hat{S}(C(a))) \cdot x(N, C(a)))$
computer (wc)	$\max_{a \in \mathcal{A}_q} (f(C(a)) + x(N, C(a)))$	$\max_{a \in \mathcal{A}_q} \left( \frac{g(C(a))}{\bar{w}(\hat{S}(C(a)))} + x(N, C(a)) \right) \cdot \bar{w}(\hat{S}(N))$

Table 6.1: Overview of Algorithms 4.54 and 6.13. (To keep the table small enough for the page, “computer (avg)” refers to the calculation for a computer node for an average-optimal solution, and “computer (wc)” refers to the calculation for a computer node for a worst-case-optimal solution.)

In particular, we now only have additions and multiplications, whereas in Algorithm 4.55 we also needed fractions. This means that we can now use integers for all values, which makes calculations somewhat easier.

We can further simplify the calculation of average-optimal computer nodes in the following way, using (among other things) that the cost of an outgoing edge is defined by the cost of the answer corresponding to it, and that the union of the remaining situations in the children nodes of a computer node is the set of remaining situations in their parent:

$$\begin{aligned}
g(N) &:= \sum_{a \in \mathcal{A}_q} \left( g(C(a)) + \bar{\mathfrak{w}} \left( \hat{\mathcal{S}}(C(a)) \right) \cdot x(N, C(a)) \right) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{a \in \mathcal{A}_q} \bar{\mathfrak{w}} \left( \hat{\mathcal{S}}(C(a)) \right) \cdot x(N, C(a)) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{a \in \mathcal{A}_q} \left( \sum_{s \in \hat{\mathcal{S}}(C(a))} \mathfrak{w}(s) \right) \cdot x(N, C(a)) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{a \in \mathcal{A}_q} \left( \sum_{s \in \hat{\mathcal{S}}(C(a))} \mathfrak{w}(s) \cdot x(N, C(a)) \right) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{a \in \mathcal{A}_q} \left( \sum_{s \in \hat{\mathcal{S}}(C(a))} \mathfrak{w}(s) \cdot \mathfrak{c}(q, a) \right) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{a \in \mathcal{A}_q} \left( \sum_{\substack{s \in \hat{\mathcal{S}}(N) \\ \mathfrak{r}(s, q) = a}} \mathfrak{w}(s) \cdot \mathfrak{c}(q, a) \right) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{a \in \mathcal{A}_q} \left( \sum_{\substack{s \in \hat{\mathcal{S}}(N) \\ \mathfrak{r}(s, q) = a}} \mathfrak{w}(s) \cdot \mathfrak{c}(q, \mathfrak{r}(s, q)) \right) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{a \in \mathcal{A}_q} \sum_{\substack{s \in \hat{\mathcal{S}}(N) \\ \mathfrak{r}(s, q) = a}} (\mathfrak{w}(s) \cdot \mathfrak{c}(q, \mathfrak{r}(s, q))) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(q, \mathfrak{r}(s, q)) \\
&= \sum_{a \in \mathcal{A}_q} g(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q)
\end{aligned}$$

For simplifying the player node calculation, we have to notice that all outgoing edges of a player node have cost 0, except for the one edge that leads to the estimate leaf. We therefore “move” the cost of the expected penalty from the edge between player node and estimate leaf down to the estimate leaf itself. That is, we redefine the following two things:

- For each estimate leaf  $E$ , we label the edge between  $E$  and its parent with cost 0.
- For each estimate leaf  $E$ , we define the future costs of  $E$  as the cost that was originally assigned to the edge above it, that is, the cost defined by Algorithm 4.34.

This results in the modified future cost times weight function  $g'(N)$ :

**Algorithm 6.14.** • Let  $N$  be an estimate leaf, and let  $\hat{\mathcal{S}}$  be the set of remaining situations in  $E$ .

If looking for an average-optimal solution, then choose an estimate  $\bar{e} \in \mathcal{E}$  such that

$$\hat{\mathbf{p}}(e) = \frac{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot \mathbf{p}(s, e)}{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)}$$

is minimal, and set

$$g'(N) := \hat{\mathbf{p}}(\bar{e}) \cdot \bar{\mathbf{w}}(\hat{\mathcal{S}}(N)) .$$

If looking for a worst-case-optimal solution, choose an estimate  $\bar{e}' \in \mathcal{E}$  such that

$$\hat{\mathbf{p}}'(e) = \max(\mathbf{p}(s, e))$$

is minimal, and set

$$g'(N) := \hat{\mathbf{p}}'(\bar{e}') \cdot \bar{\mathbf{w}}(\hat{\mathcal{S}}(N)) .$$

• If  $N$  is a player node, then

$$g'(N) := \min_{C \in \text{children}(N)} (g(C)) .$$

• Let  $N$  be a computer node, then

$$g'(N) := g(N) .$$

**Lemma 6.15.** Algorithm 6.14 is equivalent to Algorithm 6.13. In particular, for all nodes except for estimate leaves, it holds that  $g(N) = g'(N)$ .

*Proof.* This follows directly from the discussion above. □

Before summarizing this new version, let us simplify the calculation of the estimate leaves a little more. For an average-optimal solution, what we actually have is this:

$$\begin{aligned} g'(N) &:= \hat{\mathbf{p}}(\bar{e}) \cdot \bar{\mathbf{w}}(\hat{\mathcal{S}}(N)) \\ &= \min_{e \in \mathcal{E}} \left( \frac{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot \mathbf{p}(s, e)}{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)} \right) \cdot \bar{\mathbf{w}}(\hat{\mathcal{S}}(N)) \\ &= \min_{e \in \mathcal{E}} \left( \frac{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot \mathbf{p}(s, e)}{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)} \right) \cdot \sum_{s \in \hat{\mathcal{S}}(N)} \mathbf{w}(s) \\ &= \min_{e \in \mathcal{E}} \left( \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot \mathbf{p}(s, e) \right) \end{aligned}$$

Alternatively, if a nice function  $\mathbf{p}'$  was supplied by the user, we can simply use

$$g'(N) := \mathbf{p}'(\hat{\mathcal{S}}(N)) .$$

Likewise, if we want a worst-case-optimal solution and have a function  $\mathbf{p}''$  given by the user, we can simply use

$$g'(N) := \mathbf{p}''(\hat{\mathcal{S}}(N)) \cdot \bar{\mathbf{w}}(\hat{\mathcal{S}}(N)) .$$

There is no good optimization available in case  $\mathbf{p}''$  was not supplied by the user, so in that case we can as well create  $\mathbf{p}''$  ourselves as described earlier.

Together with the previous optimization, we therefore get the rules described in Table 6.2.

	Algorithm 4.54 $f(N) :=$	Algorithm 6.14 $g'(N) :=$
estimate (avg)	0	$\min_{e \in \mathcal{E}} (\sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot \mathfrak{p}(s, e))$ or $\mathfrak{p}'(\hat{\mathcal{S}}(N))$
estimate (wc)	0	$\min_{e \in \mathcal{E}} (\max(\mathfrak{p}(s, e))) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N))$ or $\mathfrak{p}''(\hat{\mathcal{S}}(N)) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N))$
player	$\min_{C \in \text{children}(N)} (f(C) + x(N, C))$	$\min_{C \in \text{children}(N)} (g'(C))$
computer (avg)	$\frac{\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}, q, a) \cdot (f(C(a)) + x(N, C(a)))}{\sum_{a \in \mathcal{A}_q} \hat{\mathfrak{w}}(\hat{\mathcal{S}}, q, a)}$	$\sum_{a \in \mathcal{A}_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q)$
computer (wc)	$\max_{a \in \mathcal{A}_q} (f(C(a)) + x(N, C(a)))$	$\max_{a \in \mathcal{A}_q} \left( \frac{g'(C(a))}{\bar{\mathfrak{w}}(\hat{\mathcal{S}}(C(a)))} + \mathfrak{c}(q, a) \right) \cdot \bar{\mathfrak{w}}(\hat{\mathcal{S}}(N))$

Table 6.2: Overview of Algorithms 4.54 and 6.14. (To keep the table small enough for the page, “computer (avg)” and “estimate (avg)” refer to the calculations for an average-optimal solution, and “computer (wc)” and “estimate (wc)” refer to the calculation for a worst-case-optimal solution.)

How is this better than the original algorithm?

First of all, we do not have any references to  $x(X, Y)$  any more, which were simply hiding the edge labeling part of the algorithm. Instead, we can now explicitly write  $\mathfrak{p}'(\hat{\mathcal{S}}(N))$  and  $\mathfrak{c}(s, q)$  for the edge costs of edges leading to estimate and player nodes, respectively. In the old algorithm we also multiplied with  $x(X, Y)$  in places where we actually knew it to be 0. Mathematically that does not make a difference, but once it comes to implementing, even those multiplications with 0 might have a performance cost (unless we have a really smart compiler).

Depending on how we would have implemented  $x(X, Y)$  (`getEdgeCost(...)`), calculating the cost of an edge would probably also have required some lookup, or at the very least some if-statements. Is the parent is a player node? Is the child is a computer node? And so on.

Second, we now only have additions and multiplications. In addition, for average-optimal solutions we got rid of all calculations of  $\hat{\mathfrak{w}}_a$  and  $\bar{\mathfrak{w}}$  in the computer node. In fact, we can process child costs and edge costs completely separately. The child costs simply are the sum of the costs of all children, without any further multiplication, division or anything of that sort. The edge costs are just as simply the sum of the edge costs of all remaining situations. We do not even need to count any more how many situations for each answer we get; we simply add the edge cost for each situation exactly once (and we can calculate both weight and edge cost directly from the situation we are currently looking at).

We also finally integrated the functions  $\mathfrak{p}'$  and  $\mathfrak{p}''$  that the user might provide, and we found an optimization in case he does not.

Last but not least, we got entirely rid of the edge costs in the player node calculation. What is left is a simple calculation of a minimum.

For worst-case-optimal solutions, on the other hand, the formerly used Algorithm 6.10 is still more efficient, since – other than for the average case – we did not need fractions there anyway, whereas in the new algorithm using  $g'(\cdot)$  we now actually need them. This is a result of the fact that the worst-case-optimal algorithm completely ignores the existence of weights, whereas the function  $g'(X)$  needs to include information about them.

Last but not least, let us transform this into an algorithm:

**Algorithm 6.16.** *Calculating player nodes:*

```

calculateCostsPlayerNode(Node N) :=
  if KnownResults.contains(N) then
    return KnownResults.get(N);
  end if
  minimum = infinity;
  for all Child C : calculatePlayerNodeChildren(N) do
    cost = getCosts(C) + getWeight(N.getSituations()) * C.getEdgeCost();
    minimum = min(minimum, cost);
  end for
  KnownResults.put(N, minimum);
  return minimum;

```

*Calculating player nodes without the deleted part:*

```

calculateCostsPlayerNode(Node N) :=
  if KnownResults.contains(N) then
    return KnownResults.get(N);
  end if
  minimum = infinity;
  for all Child C : calculatePlayerNodeChildren(N) do
    cost = getCosts(C);
    minimum = min(minimum, cost);
  end for
  KnownResults.put(N, minimum);
  return minimum;

```

*Calculating average-optimal computer nodes:*

```

calculateCostsComputerNodeAvg(Node N) :=
  sum = 0;
  for all Situation S : N.getSituations() do
    sum = sum + S.getWeight() * N.getQuestion().getAnswerCost(S);
  end for
  for all Child C : calculateComputerNodeChildren(N) do
    sum = sum + getCosts(C);
  end for
  return sum;

```

*Calculating worst-case-optimal computer nodes did not change, so we do not print it again.*

*Calculating estimate leafs:*

```

calculateCostsEstimateLeaf(Node N) :=
  return getBestEstimateCosts(N.getSituations());

```

We assume `getBestEstimateCosts(List[Situation])` to be implemented according to one of the functions from Table 6.2, that is, either using `p'` if it was provided, or just using the normal calculation.

The expected future cost in the root `root` is again

`getCosts(root) / getWeight(root.getSituations())`.

## 6.5 Branch and bound

As in many other optimization problems, when looking for the best solution we sometimes already know that a better solution exists in some other branch. Consequently, we want to abort calculating branches as soon as we see that they cannot possibly be optimal any more.

To do so, we simply pass an upper limit to `getCosts(...)` and allow the function to return  $\infty$  instead of  $g'(N)$  if it is certain that  $g'(N)$  is larger than the limit.

In other words, we define a new function as follows:

$$g''(N, L) := \begin{cases} g'(N) & g'(N) < L \\ \infty & g'(N) \geq L \end{cases}$$

Here, we actually even demand the function to always return  $\infty$  if  $g'(N) \geq L$ . However, this is easy to implement by simply calculating  $g'(N)$  as before and comparing it to  $L$  before returning a value.

What we actually want to do however is finding ways to tell that  $g'(N)$  will be larger than  $L$  without calculating all of it.

Before we start looking into this, one small remark that follows directly from the definition:

**Remark 6.17.**

$$g''(N, \infty) = g'(N)$$

### 6.5.1 Limits in player nodes

For a player node, we need to find the minimum of its children. If at least one child has expected costs lower than the given limit, then we will return the expected future costs of the cheapest child. If all children are above the limit, then so will be their minimum, therefore we can return  $\infty$ . This is described in the following lemma:

**Lemma 6.18.** *Let  $N$  be a player node and  $L$  a given limit. Let  $C$  be a child of  $N$  with  $g'(C) \geq L$ . Then the exact value of  $g'(C)$  is irrelevant for the calculation of  $g''(N, L)$ . (That is, even if  $g'(C)$  had some other value  $x \geq L$ , the result of  $g''(N, L)$  would be the same.)*

*Proof.* We have two cases here. Either there exists some other child  $C'$  of  $N$  for which  $g'(C') < L$ . Then  $g''(N, L) = g'(N) = \min_{\bar{C} \in \text{children}(N)} g'(\bar{C}) \leq g'(C') < L \leq g'(C)$ , that is, the value of  $g'(C)$  does not influence the minimum.

Or for all children  $C''$  of  $N$  it holds that  $g'(C'') \geq L$ . Then  $g'(N) = \min_{\bar{C} \in \text{children}(N)} g'(\bar{C}) \geq L$  and thus  $g''(N, L) = \infty$ . Again, the exact value of  $g'(C)$  is irrelevant.  $\square$

This means that only the exact values of children  $C$  with  $g'(C) < L$  are interesting for the calculation of  $g'(N)$  anyway. We can therefore recursively calculate the children of  $N$  using  $g''(C, L)$  instead of  $g'(C)$  with the same upper limit  $L$ :

**Corollary 6.19.** *When calculating the expected future costs of a child  $C$  of  $N$ , we can calculate  $g''(C, L)$  instead of  $g'(C)$ .*

*Proof.* This follows from the previous lemma and the definition of  $g''$ . If  $g'(C) < L$ , then  $g''(C, L) = g'(C)$ . If  $g'(C) \geq L$ , then  $g''(C, L) = \infty \geq L$ . As we saw in the previous lemma, the exact value of  $g'(C)$  does not influence the result in this case.  $\square$

Besides that, since we are looking for the minimum of the children, each child is an upper limit for all its siblings as well:

**Lemma 6.20.** *Let  $N$  be a player node, and let  $C$  be one child of  $N$  with  $g'(C) = x$ . Let  $C'$  be another child of  $N$  with  $g'(C') \geq g'(C) = x$ . Then the exact value of  $g'(C')$  is irrelevant for the calculation of  $g''(N, L)$ .*

*Proof.* We will show that this is already irrelevant for the calculation of  $g'(N)$ , and consequently also for that of  $g''(N, L)$ . The value  $g'(N)$  is the minimum of the children, and by our assumption  $C'$  is not the child for which the costs are smallest. Therefore, the exact value does not influence the minimum.  $\square$

Combining these two we get the following:

**Theorem 6.21.** *Let  $N$  be a player node and  $L$  a given limit. Let  $\langle C \rangle \subseteq \text{children}(N)$  be a subset of the children whose values we have already calculated, and let  $C' \in \text{children}(N) \setminus \langle C \rangle$  be another child of  $N$ .*

*Let  $m = \min_{C \in \langle C \rangle} (g'(C))$  be the minimum among the already calculated children.*

*If  $g'(C') \geq \min(L, m)$ , then the exact value of  $g'(C')$  is irrelevant for the result of  $g''(N, L)$ .*

*Proof.* Most of this already follows from the previous lemmata. If there is one child  $C$  in  $\langle C \rangle$  for which  $g'(C) \leq g'(C')$ , then by Lemma 6.20 the exact value  $g'(C')$  does not matter. If  $g'(C') \geq L$ , then the exact value does not matter by Lemma 6.18.

Consequently,  $g'(C')$  has to be smaller than the limit *and* smaller than all children we already know in order to be possibly relevant.  $\square$

For any child that is irrelevant, returning  $\infty$  instead of its value does not change the value of the minimum. Therefore we can pass upper limits to the calculation of the children such that the calculation of the child can be aborted if its result will be higher than a child we have found already, or higher than  $L$ , the limit given for the calculation of the parent itself. This leads to the following way to calculate the expected future costs of a child node:

**Algorithm 6.22.** *Let  $\{C_1, C_2, \dots, C_m\} = \text{children}(N)$  be the list of children of a player node  $N$  in any fixed order. We can calculate  $g'(C')$  as follows, where  $g_i$  denotes the intermediate value after having looked at  $i$  children:*

$$\begin{aligned} g_0 &= \infty \\ g_1 &= \min(g_0, g''(C_1, \min(L, g_0))) \\ g_2 &= \min(g_1, g''(C_2, \min(L, g_1))) \\ &\vdots \\ g_i &= \min(g_{i-1}, g''(C_i, \min(L, g_{i-1}))) \end{aligned}$$

*Then we set  $g''(N, L) = g_m$ .*

Even though this is fairly obvious and mostly follows already from the argumentation above, getting it into a clear mathematical form will be a little laborious.

**Theorem 6.23.** *Algorithm 6.22 correctly calculates  $g''(N, L)$ .*

*Proof.* In each step,  $g_i$  contains the minimum of all potentially relevant children we have seen already.

By Theorem 6.21, the next child we look at can only be relevant if its value is smaller than  $L$  and smaller than the minimum of *all* children seen so far. Here, we only consider it irrelevant if it is smaller than  $L$  and smaller than all *potentially relevant* children seen so far, which (at the first glance) is a weaker condition. Consequently,  $g''(C_i, \min(L, g_{i-1}))$  can only be  $\infty$  if the child is certainly irrelevant.

Therefore, all potentially relevant children are accounted for when we reach  $g_m$ , and the value  $g''(N, L)$  is the same as if we had not given any limits to the calculations for the children.

Looking at the algorithm in some more detail, we even see that the condition is not weaker but equivalent.



We can also prove this theorem directly:

Let  $x = \min_{C_i \in \text{children}(N)} g'(C_i)$ . By definition,  $g''(N, L) = \begin{cases} x & x < L \\ \infty & x \geq L \end{cases}$ .

Let us define  $x_k = \min_{i \in \{1, 2, \dots, k\}} g'(C_i)$ , the minimum of the children we have looked at already, and let  $h_k = \begin{cases} x_k & x_k < L \\ \infty & x_k \geq L \end{cases}$ . We see that  $x = x_m$  and  $g''(N, L) = h_m$ .

We will show that  $g_k = h_k$  for all  $k > 0$  using complete induction. For 1, we use  $x_1 = \min_{i \in \{1\}} g'(C_i) = g'(C_1)$  and get

$$\begin{aligned} g_1 &= \min(g_0, g''(C_1, \min(L, g_0))) \\ &= \min(\infty, g''(C_1, \min(L, \infty))) \\ &= g''(C_1, L) \\ &= \begin{cases} g'(C_1) & g'(C_1) < L \\ \infty & g'(C_1) \geq L \end{cases} \\ &= \begin{cases} x_1 & x_1 < L \\ \infty & x_1 \geq L \end{cases} \\ &= h_1. \end{aligned}$$

For the induction step, let us assume that  $g_{k-1} = h_{k-1}$  holds. Then we get

$$\begin{aligned} g_k &= \min(g_{k-1}, g''(C_k, \min(L, g_{k-1}))) \\ &= \min(h_{k-1}, g''(C_k, \min(L, g_{k-1}))) \\ &= \min \left( \left( \begin{cases} x_{k-1} & x_{k-1} < L \\ \infty & x_{k-1} \geq L \end{cases} \right), \left( \begin{cases} g'(C_k) & g'(C_k) < \min(L, g_{k-1}) \\ \infty & g'(C_k) \geq \min(L, g_{k-1}) \end{cases} \right) \right). \end{aligned}$$

We have two cases for the first and two for the second parameter, so we differentiate between four cases.

- Case 1:  $x_{k-1} < L$  and  $g'(C_k) < \min(L, g_{k-1})$  (That is, we have seen at least one potentially relevant child before, and the current child is relevant.)

$$\begin{aligned} g_k &= \min \left( \left( \begin{cases} x_{k-1} & x_{k-1} < L \\ \infty & x_{k-1} \geq L \end{cases} \right), \left( \begin{cases} g'(C_k) & g'(C_k) < \min(L, g_{k-1}) \\ \infty & g'(C_k) \geq \min(L, g_{k-1}) \end{cases} \right) \right) \\ &= \min(x_{k-1}, g'(C_k)) \\ &= \min \left( \min_{i \in \{1, 2, \dots, k-1\}} g'(C_i), g'(C_k) \right) \\ &= \min_{i \in \{1, 2, \dots, k\}} g'(C_i) \\ &= x_k \end{aligned}$$

We also see immediately that  $x_k = g_k < L$ , therefore  $h_k = x_k = g_k$ .

- Case 2:  $x_{k-1} \geq L$  and  $g'(C_k) < \min(L, g_{k-1})$  (That is, we have not seen any relevant children before, and the current child is relevant.)

From  $x_{k-1} = \min_{i \in \{1, 2, \dots, k-1\}} g'(C_i) \geq L$  it follows that  $g'(C_i) \geq L$  for all  $1 \leq i \leq k-1$ . Because  $g'(C_k) < \min(L, g_{k-1}) \leq L$  we get  $x_k = \min_{i \in \{1, 2, \dots, k\}} g'(C_i) = g'(C_k) < L$ , and consequently also  $h_k = x_k$ .

Therefore:

$$\begin{aligned} g_k &= \min \left( \left( \left\{ \begin{array}{ll} x_{k-1} & x_{k-1} < L \\ \infty & x_{k-1} \geq L \end{array} \right\} \right), \left( \left\{ \begin{array}{ll} g'(C_k) & g'(C_k) < \min(L, g_{k-1}) \\ \infty & g'(C_k) \geq \min(L, g_{k-1}) \end{array} \right\} \right) \right) \\ &= \min(\infty, g'(C_k)) \\ &= g'(C_k) \\ &= x_k \\ &= h_k \end{aligned}$$

- Case 3:  $x_{k-1} < L$  and  $g'(C_k) \geq \min(L, g_{k-1})$  (That is, we have seen at least one potentially relevant child before, and the current child is not relevant.)

Since  $x_{k-1} < L$  we have  $x_{k-1} = h_{k-1} = g_{k-1}$ , and thus also  $g_{k-1} < L$ . Consequently,  $\min(L, g_{k-1}) = g_{k-1}$ , and therefore  $g'(C_k) \geq \min(L, g_{k-1}) = g_{k-1} = x_{k-1}$ . Therefore,  $x_{k-1} = \min(x_{k-1}, g'(C_k))$ , and we get

$$\begin{aligned} g_k &= \min \left( \left( \left\{ \begin{array}{ll} x_{k-1} & x_{k-1} < L \\ \infty & x_{k-1} \geq L \end{array} \right\} \right), \left( \left\{ \begin{array}{ll} g'(C_k) & g'(C_k) < \min(L, g_{k-1}) \\ \infty & g'(C_k) \geq \min(L, g_{k-1}) \end{array} \right\} \right) \right) \\ &= \min(x_{k-1}, \infty) \\ &= x_{k-1} \\ &= \min(x_{k-1}, g'(C_k)) \\ &= \min\left(\min_{i \in \{1, 2, \dots, k-1\}} g'(C_i), g'(C_k)\right) \\ &= \min_{i \in \{1, 2, \dots, k-1\}} g'(C_i) \\ &= x_k \\ &= h_k, \end{aligned}$$

using that  $x_k = x_{k-1} < L$  in the last step.

- Case 4:  $x_{k-1} \geq L$  and  $g'(C_k) \geq \min(L, g_{k-1})$  (That is, we have not seen any relevant children before, and the current child is not relevant.)

From  $x_{k-1} = \min_{i \in \{1, 2, \dots, k-1\}} g'(C_i) \geq L$  it follows that  $g'(C_i) \geq L$  for all  $1 \leq i \leq k-1$ . Since  $g_{k-1} = h_{k-1} = \infty$ , we have  $\min(L, g_{k-1}) = \min(L, \infty) = L$  and therefore also  $g'(C_k) \geq \min(L, g_{k-1}) = L$ .

Therefore we get

$$\begin{aligned} g_k &= \min \left( \left( \left\{ \begin{array}{ll} x_{k-1} & x_{k-1} < L \\ \infty & x_{k-1} \geq L \end{array} \right\} \right), \left( \left\{ \begin{array}{ll} g'(C_k) & g'(C_k) < \min(L, g_{k-1}) \\ \infty & g'(C_k) \geq \min(L, g_{k-1}) \end{array} \right\} \right) \right) \\ &= \min(\infty, \infty) \\ &= \infty \\ x_k &= \min_{i \in \{1, 2, \dots, k\}} g'(C_i) \\ &\geq L \\ h_k &= \infty \end{aligned}$$

and thus  $g_k = h_k$ .

□

We have one last small problem: If at least one child is smaller than the limit, everything is fine and we get the same result as  $g'(N)$ . If however all children are above the limit, then we never find out what the value of  $g'(N)$  is, so we also cannot store it for future reuse. However, if all children are higher than the given limit, then we at least know that they are also all higher than any lower limit we could have used. (For example, if we tried to calculate with limit 5 and found no child with costs below 5, then we certainly also will not find a child with costs below 3.)

The following observation follows directly from the definition of  $g''(N, L)$ :

**Lemma 6.24.** *For every  $\epsilon > 0$  it holds that*

$$g''(N, L) \leq g''(N, L - \epsilon) .$$

*Proof.* This follows immediately from

$$\begin{aligned} g''(N, L) &= \begin{cases} g'(N) & g'(N) < L \\ \infty & g'(N) \geq L \end{cases} \\ &\leq \begin{cases} g'(N) & g'(N) < L - \epsilon \\ \infty & g'(N) \geq L - \epsilon \end{cases} \\ &= g''(N, L - \epsilon) . \end{aligned}$$

□

Consequently, if  $g''(N, L) = \infty$  for some  $L$ , then also  $g''(N, L - \epsilon) = \infty$  for every  $\epsilon > 0$ . If we get  $\infty$  as result, we therefore store this in a separate data structure together with the limit.

This leads to the following modified algorithm for calculating `getCosts(...)` for a player node:

**Algorithm 6.25.** *Let `KnownLimits` be the new additional data structure that stores (node, limit) tuples, with node as key.*

```

calculateCostsPlayerNode(Node N, integer L) :=
  if KnownResults.contains(N) then
    if KnownResults.get(N) < L then
      return KnownResults.get(N);
    else
      return infinity;
    end if
  end if
  if KnownLimits.contains(N) then
    if KnownLimits.get(N) <= L then
      return infinity;
    end if
  end if

  minimum = infinity;
  for all Child C : calculatePlayerNodeChildren(N) do
    cost = getCosts(C, min(L, minimum));
    minimum = min(minimum, cost);
  end for
  if minimum < infinity then
    KnownResults.put(N, minimum);
  else
    KnownLimits.put(N, L);
  end if
  return minimum;

```

### 6.5.2 Limits in computer nodes for average-optimal solutions

Computer nodes calculate a weighted average of their children. If already every single edge leading to one of these children is too expensive, then also their average will be too high.

**Lemma 6.26.** *Let  $N$  be a computer node, and let  $x(X, Y)$  denote the cost of the edge between  $X$  and  $Y$ .*

*If  $x(N, C) \geq \frac{L}{\overline{\mathfrak{w}}(\hat{\mathcal{S}}(N))}$  for all children  $C$  of  $N$ , then  $g'(N) \geq L$ .*

*Proof.* This is fairly obvious, since for computer nodes

$$\begin{aligned}
 g'(N) &= g(N) \\
 &= \sum_{a \in \mathcal{A}_q} \left( g(C(a)) + \overline{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot x(N, C(a)) \right) \\
 &\geq \sum_{a \in \mathcal{A}_q} \left( \overline{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot x(N, C(a)) \right) \\
 &\geq \sum_{a \in \mathcal{A}_q} \left( \overline{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \cdot \frac{L}{\overline{\mathfrak{w}}(\hat{\mathcal{S}}(N))} \right) \\
 &= \frac{L}{\overline{\mathfrak{w}}(\hat{\mathcal{S}}(N))} \cdot \sum_{a \in \mathcal{A}_q} \left( \overline{\mathfrak{w}}(\hat{\mathcal{S}}(C(a))) \right) \\
 &= \frac{L}{\overline{\mathfrak{w}}(\hat{\mathcal{S}}(N))} \cdot \overline{\mathfrak{w}}(\hat{\mathcal{S}}(N)) \\
 &= L .
 \end{aligned}$$

□

This condition can be easily checked in  $O(\mathfrak{a})$  time if we know  $\overline{\mathfrak{w}}(\hat{\mathcal{S}}(N))$ , since we only need to look up the costs of the available answers to the question that has to be answered, of which there are at most  $\mathfrak{a}$ .

Alternatively, we can look at the two parts of calculating the costs of a computer node. As mentioned before, the sum of the edge costs and the sum of the child costs can be calculated separately. Calculating child costs requires recursive calculations, whereas calculating the edge costs can be done immediately. We therefore start with the edge costs, and if these are too high already, we are done as well.

**Lemma 6.27.** *Let  $N$  be a computer node.*

*If  $\sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \geq L$ , then also  $g'(N) \geq L$ .*

*Proof.* This follows directly from

$$\begin{aligned}
 g'(N) &= \sum_{a \in \mathcal{A}_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \\
 &\geq \sum_{a \in \mathcal{A}_q} g'(C(a)) + L \\
 &\geq L .
 \end{aligned}$$

□

If this is still not too high, we can hope that we can at least abort after the first few children:

**Lemma 6.28.** *Let  $N$  be a computer node, and let  $\hat{\mathcal{A}}_q \subseteq \mathcal{A}_q$  be the set of answers for which we already calculated the children.*

*If  $\sum_{a \in \hat{\mathcal{A}}_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \geq L$ , then also  $g'(N) \geq L$ .*

*Proof.* This again follows directly from

$$\begin{aligned} g'(N) &= \sum_{a \in \mathcal{A}_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \\ &\geq \sum_{a \in \hat{\mathcal{A}}_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \\ &\geq L . \end{aligned}$$

□

We can therefore check after each child whether we already are too high.

We can even go one step further and impose a limit on the calculation of the next child: If our limit is  $L$ , and the sum of the edges and the children we have already calculated is  $X$ , then we are only  $L - X$  away from that limit any more. If the next child was more expensive than  $L - X$ , we would be over the limit, so we do not really care any more *how much* the costs of that child are above  $L - X$ . Therefore, we can use  $L - X$  as an upper limit for the calculation of that child:

**Lemma 6.29.** *Let  $N$  be a computer node and let  $\hat{\mathcal{A}}_q \subseteq \mathcal{A}_q$  be a set of answers (for which we already calculated the cost of their children). Let  $a' \in \mathcal{A}_q \setminus \hat{\mathcal{A}}_q$  be an answer (for which we have not calculated the costs yet), and let  $C(a')$  be the child corresponding to that answer.*

*If*

$$g'(C(a')) \geq L - \left( \sum_{a \in \hat{\mathcal{A}}_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \right) ,$$

*then  $g'(N) \geq L$ .*

*Proof.* Let us assume that  $g'(C(a'))$  is larger than the limit given above. Let  $\hat{\mathcal{A}}'_q = \hat{\mathcal{A}}_q \cup \{a'\} \subseteq \mathcal{A}_q$ .

Then

$$\begin{aligned} g'(N) &= \sum_{a \in \hat{\mathcal{A}}'_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \\ &= \sum_{a \in \hat{\mathcal{A}}_q} g'(C(a)) + g'(C(a')) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \\ &\geq \sum_{a \in \hat{\mathcal{A}}_q} g'(C(a)) + L - \left( \sum_{a \in \hat{\mathcal{A}}_q} g'(C(a)) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \right) + \sum_{s \in \hat{\mathcal{S}}(N)} \mathfrak{w}(s) \cdot \mathfrak{c}(s, q) \\ &= L . \end{aligned}$$

□

This leads to the following modified algorithm for calculating `getCosts(...)` for a computer node:

**Algorithm 6.30.**

```

calculateCostsComputerNodeAvg(Node N, integer L) :=
  // check whether all edges are too expensive
  comparisonLimit = L / getWeight(N.getSituations());
  allTooExpensive = true;
  for all integer A : Q.getAvailableAnswers() do
    if Q.getAnswerCost(A) < L then
      allTooExpensive = false;
    end if
  end for
  if allTooExpensive then
    return infinity;
  end if

  // calculate edge costs
  sum = 0;
  for all Situation S : N.getSituations() do
    sum = sum + S.getWeight() * N.getQuestion().getAnswerCost(S);
  end for
  if sum >= L then
    return infinity;
  end if

  // calculate child costs
  for all Child C : calculateComputerNodeChildren(N) do
    sum = sum + getCosts(C, L - sum);
    if sum >= L then
      return infinity;
    end if
  end for

  return sum;

```

### 6.5.3 Limits in computer nodes for worst-case-optimal solutions

When looking for worst-case-optimal solutions, then limits for computer nodes are a lot simpler: as soon as we find one child of the computer node that is more expensive than the limit, then the computer node itself is too expensive.

**Lemma 6.31.** *Let  $N$  be a computer node, and let  $C$  be a child of  $N$  with*

$$f(C) + x(N, C) \geq \frac{L}{\overline{\mathfrak{w}(\mathcal{S}(N))}},$$

*then  $g'(N) \geq L$  (where  $x(X, Y)$  denotes the cost of the edge from  $X$  to  $Y$ ).*

*Proof.* Note that we used the original  $f(C)$  again to avoid the less readable equation that we would get for  $g'(C)$ . Since  $f(C) = \frac{g'(C)}{\overline{\mathfrak{w}(\mathcal{S}(C))}}$  we can always calculate  $f(C)$  from  $g'(C)$  again.

We now get

$$\begin{aligned}
f(N) &= \max_{a \in \mathcal{A}_q} (f(C(a)) + x(N, C(a))) \\
&\geq \frac{L}{\overline{w}(\hat{\mathcal{S}}(N))} \\
g'(N) &= \overline{w}(\hat{\mathcal{S}}(N)) \cdot f(N) \\
&\geq \overline{w}(\hat{\mathcal{S}}(N)) \cdot \frac{L}{\overline{w}(\hat{\mathcal{S}}(N))} \\
&= L,
\end{aligned}$$

the desired inequality. □

Transforming back to  $g'(N)$ , we get the following equivalent condition:

$$\begin{aligned}
f(C) + x(N, C) &\geq \frac{L}{\overline{w}(\hat{\mathcal{S}}(N))} \iff \\
f(C) &\geq \frac{L}{\overline{w}(\hat{\mathcal{S}}(N))} - x(N, C) \iff \\
\frac{g'(C)}{\overline{w}(\hat{\mathcal{S}}(C))} &\geq \frac{L}{\overline{w}(\hat{\mathcal{S}}(N))} - x(N, C) \iff \\
g'(C) &\geq \left( \frac{L}{\overline{w}(\hat{\mathcal{S}}(N))} - x(N, C) \right) \cdot \overline{w}(\hat{\mathcal{S}}(C))
\end{aligned}$$

Therefore, as soon as we find one child that is above this limit, it does not matter any more *how much* it is above it. We can therefore pass this as upper limit to the calculation of the children and get the following algorithm (in which we will again assume that we have a data type `fraction` that allows us to use division without loss of precision):

**Algorithm 6.32.** `calculateCostsComputerNodeWorst(Node N) :=`

```

maximum = 0;
for all Node C : calculateComputerNodeChildren(N) do
  limit = (L / getWeight(N.getSituations())) - C.getEdgeCost() *
  getWeight(C.getSituations());
  cost = getCosts(C, limit) / getWeight(C.getSituations()) + C.getEdgeCost();
  maximum = max(cost, maximum);
  if maximum * getWeight(N.getSituations()) >= L then
    return infinity;
  end if
end for
return maximum * getWeight(N.getSituations());

```

The two advantages compared to the original algorithm are on the one hand that we abort after the first child that is too large, and on the other hand that we pass the given limit on to the recursive calculations of the children, thereby allowing them to abort without finishing the full calculation as well.

### 6.5.4 Randomization

Finding good limits quickly can vastly improve performance, since it allows us to abort many calculations. For example, let us assume we are looking for the strategy with fewest moves. If we want to calculate the best next move for a player node and found one move already that will result in average costs of 8.6, then we can tell all other children of that node that if they need more than 8.6 moves on average, they are not interesting to us anyway, so they do not even need to bother calculating their exact results.

In particular, in BattleShips and in Minesweeper with free empty fields we used questions that had infinite cost if they were asked in situations where it was not legal to ask them. As soon as we find one legal strategy, we can give the corresponding finite limit to computer nodes, so looking at Algorithm 6.30 we see that any computer nodes that finds one outgoing edge with infinite costs will abort the calculation before ever looking at one of its children. Consequently, finding at least one legal strategy soon is very important here.

One simple trick to find a good initial limit could be to choose any random strategy and evaluate the expected outcome for that strategy. That is, for each player node, we do not calculate the minimum child (and potentially have to evaluate all children to do so), but simply pick one child at random.

However, for computer nodes we still have to evaluate all children. The total number of nodes we need to look at for one random strategy can therefore go up to  $a^q$  – we can potentially ask up to  $q$  questions, and have to evaluate up to  $a$  subtrees for each of them. Given that the entire decision set graph contains at most  $(a + 1)^q$  nodes, this is a significant amount of the tree. Worse than that, nodes evaluated this way do not generate any reusable results for later.

Therefore, the calculation effort that this would require probably outweighs the benefits.

However, one problem that we see in practice in many games is that the questions are ordered in some way. For example, for MasterMind the first questions might be AAAA, AAAB, AAAC, AAAD, AABA, AABB, AABC, ... (using A, B, C and D to denote colours).

Actually asking the questions in this order might well be the worst possible strategy. However, this is exactly the order in which we currently evaluate children, assuming that “for (Question  $q$  : questions)” always returns questions in the same order.

While not asymptotically better in any way, in practice, simply shuffling this list before starting to evaluate the children of a player node can improve performance a fair bit.

## 6.6 Fingerprinting

We will now look at one trick that will solve four problems in one go.

First of all, evaluating what the answer for a question is in a given situation can take a significant amount of time. In the number guessing game it might only be a simple comparison of two numbers, but in BlackBox we have to simulate the entire path of the light ray to find out where it lands. Consequently, we want to do this only once for each combination of question and situation.

Second, so far we said that each node in the decision set graph contains a list of remaining situations. Actually storing, modifying, comparing and passing around such lists creates quite some overhead, though. We would therefore like to have an alternative way to describe which situations are still possible in a situation. We have only one requirement for such a description format: All sets of situations that can occur as remaining situations in a node must be describable in that format. (Note that we do not demand *all* subsets of  $S$  to be describable, but only those that actually occur as sets of remaining situations.)

Third, we want an efficient way to calculate the children of a player node, as well as an efficient way to calculate the children of a computer node.

Fourth, we want an efficient way to store and retrieve the results of nodes we have already calculated.

Fingerprints will solve all of these problems nicely.

### 6.6.1 Creating situation fingerprints

From the analyzation point of view, the only thing that is interesting about a situation are its answers to the given questions. Also, we will most probably have to evaluate each question for each situation at least once anyway. The idea therefore is to evaluate all questions for all situations in the beginning, and store the results. Once that is done, we never need to evaluate the questions again, and just access the stored results instead.

**Definition 6.33** (situation fingerprint). *A SITUATION FINGERPRINT is a data structure that stores the answers to all questions for one situation.*

In terms of performance, we are interested in the following properties:

- For each question  $q \in \mathcal{Q}$ , the answer  $\tau(s, q)$  should be retrieved efficiently.
- Given a list of (question, answer) pairs, it should be easy to check whether they are all satisfied.



- We want to use as little storage space as possible.
- Ideally, it should also be easy to export and read in again.

The performance for *creating* the fingerprints is not that important, since it will be only done once at the beginning, and has a runtime of “only”  $O(sq)$ .

We will again look at one primitive implementation first, and then at two somewhat more efficient ones.

**Algorithm 6.34.** *The most straightforward way to implement this is to simply store (question, answer) pairs.*

*To get some amount of efficiency, we might want to first create a (bijective) mapping of  $\mathcal{Q}$  to the integers  $\{0, 1, 2, 3, \dots, |\mathcal{Q}| - 1\}$ , and a function `getQuestionNumber(Question q)`. With some optimization, looking up the number of a question with that function can run in  $O(1)$  time. (We can, for example, just take a sufficiently large hash map and hope not to get any collisions. Alternatively, Dynamic Perfect Hashing is a very elaborate version of this that guarantees  $O(1)$  lookup time. Of course, we can also simply implement `Question` in such a way that we can store this number within that data structure itself.)*

*Once we have done that, we simply store the answers in an array.*

*Calculating the fingerprint for a situation  $s$  is fairly straightforward:*

```
int[] answers = new int[questions.length];
for (Question q : questions) {
    answer[getQuestionNumber(q)] = q.getAnswer(s);
}
```

*Retrieving the answer to some question from this array is equally simple:*

```
int getAnswer(Question q) {
    return answers[getQuestionNumber(q)];
}
```

The performance for looking up the answer to one question is fairly good, and even the creation is pretty efficient. Also exporting and importing is easy enough.

However, checking whether such a situation fingerprint fulfils a list of (question, answer) pairs can be a bit of a hassle, with a runtime of  $O(k)$  for  $k$  pairs.

And as far as storage space is concerned, the array creates a fair amount of overhead. This is not so much because of the array structure itself, which is fairly efficient in most programming languages, but simply because we reserve space for an integer for each answer. If we reserve a 32 bit integer and have only 300 possible answers per question, we effectively use only 9 of those 32 bit – an overhead of more than 250%. We might try to reserve smaller integers instead, but even with a 16 bit integer we still have 78% overhead. Looking at some of our games, we have for example 34 answers per question in BlackBox, thus using 6 out of 8 bit (for an overhead of 33%), and 14 answers per question in (4-peg) MasterMind, thus using 4 out of 8 bit (for an overhead of 100%).

Furthermore, it might well happen that one question has 3000 different answers, but all the other questions only have 2. Unfortunately, an array does not allow us to choose different sizes for elements. Thus, since we need 16 bit to fit the 3000 answers, we also need to reserve 16 bits for all other questions, even though we would only need 1 bit for each of those.

One additional problem might arise in the above implementation if the possible answers  $\mathcal{A}_q$  are not continuous. For example, if we have only two possible answers for a question  $q$ , but these two answers are 138 and 712843606, then we still need 32 bit integers. However, this at least could be fixed by enumerating the answers in the same way as the questions.

A long story cut short, the storage overhead is quite bad, and we care a lot about it, since this will be the most frequently used data structure in the entire program.

Now that we have seen the not so great method, let us look at a better one.

The idea is simple: Instead of writing each answer into its own array field, we simply write the answers one after the other to form a single long bit sequence.

**Example 6.35.** For example, if we play number guessing with 10 situations, then for the situation 7, question “Is it larger than 1?” answers “true”, “Is it larger than 2?” answers “true”, “Is it larger than 3?” answers “true”, and so on, until “Is it larger than 7?” which answers “false” for the first time, followed by “Is it larger than 8?” and “Is it larger than 9?” which also answer “false”. We can denote “true” with 1 and “false” with 0. Writing all answers down one after the other this way, we get 111111000.

**Example 6.36.** If we have questions with more than two answers, then we will need more than one bit to represent each answer. As another example, let us assume we have numbers between 1 and 40, but only two questions: “What is the remainder modulo 4?” and “What is the remainder modulo 7?”. We need 2 bits to represent all possible answers for the first question, and 3 bits to represent the answers to the second. We can in each case easily map each answer to its bit representation; that is, answer 5 would, for example, be represented by 101. (Note that although we could represent 8 different answers with the 3 bits for the second question, we use only 7 of them; the answer 111 does not appear.) For situation 33, for example, we get 1 = 01 as answer to the first question, and 5 = 101 as answer to the second question. Writing them one after the other, we get 01101. We also know that the first two bits correspond to the first, the next three bits to the second question.

This is already the entire idea behind fingerprints. We will now define it in a more formal way, and will afterwards immediately look at another example (Example 6.38) for better understanding.

**Algorithm 6.37.** Let  $\mathcal{Q} = \{q_1, q_2, \dots, q_q\}$  be the list of questions in some predefined order.

To express all  $\mathbf{a}_i = |\mathcal{A}_{q_i}|$  possible answers for any question  $q_i$ , it is sufficient to use  $\lceil \text{ld}(\mathbf{a}_i) \rceil$  bits, where  $\text{ld}$  denotes the logarithm of base 2.

Consequently,  $l := \sum_{i=1}^q \lceil \text{ld}(\mathbf{a}_i) \rceil$  bits are sufficient to represent the answers to all questions. We therefore store the situation fingerprint as an  $l$ -bit integer (using a data structure that supports large integers). The first  $\lceil \text{ld}(\mathbf{a}_1) \rceil$  bits will contain the answer to question 1, the next  $\lceil \text{ld}(\mathbf{a}_2) \rceil$  bits the answer to question 2, and so on.

In order to remember which bits belong to which question, we additionally store this information for each question  $q_i$ . For simplicity, we store this information as a bitmask, that is, for question  $q_i$  we store an  $l$ -bit integer in which exactly the bits corresponding to  $q_i$  are 1, and all others are 0. Let  $M(q_i)$  denote this bitmask.

Then

$$\begin{aligned} M(q_i) \ \& \ M(q_j) &= 0 & \quad i \neq j \\ M(q_1) \ | \ M(q_2) \ | \ \dots \ | \ M(q_q) &= 2^l - 1 \end{aligned}$$

where  $\&$  denotes bitwise AND and  $|$  denotes bitwise OR. That is, no bit is set in two of the bitmasks, and each bit is set in at least one bitmask.

We also store the mapping between bit values and original answers, in such a way that we can access it in both directions. (That is, we, for example, store that answer 5 is represented by the bits 101, and vice versa.)

Let us now have a look at the actual algorithms. This is a simplified version of a Java algorithm that calculates this fingerprint. We will use `BigInteger` to represent large integers<sup>1</sup>.

First, we need to prepare our questions by calculating for each of them:

- `answerEncoding`, a mapping that maps answers from the original question to parts of the fingerprint (i.e., that would, for example, map 5 to 101);
- `answerDecoding`, the reverse mapping that maps parts of the fingerprint back to answers (i.e., that would map 101 back to 5);
- `mask`, the bitmask for the question representing which bits in the fingerprint belong to it (in Example 6.36, the mask for the first question would be 11000 and the mask for the second question would be 00111); and

<sup>1</sup>Note that we choose `BigInteger` mostly for readability and ease of use. Other data structures (like for example a simple array of integers) might be more efficient in practice.

- `bitshift`, the position of the rightmost bit belonging to this question, i.e., the position of the rightmost 1 in the `mask`, so that we know how far we need to shift for several operations (in Example 6.36, we would, for example, store for the first question with `mask=11000` that the 4th bit from the right is the lowest bit that belongs to it, so we set `bitshift` to 3 (since in Java and C++ it is customary to start counting from 0); from this we can later see that if a situation answers `1 = 01` to this question, we have to shift this 3 bits to the left to get `01000`, that is, to move it to the position in the fingerprint where it appears).

We assume that `Question` has public fields<sup>2</sup> for these four values, and prepare the questions by properly initializing them:

```
int bitpos = 0; // number of bits used by previous questions

for (Question q : questions) {
    int numAnswers = 0; // number of different answers found for this question

    // map for enumerating possible answers (which may be arbitrary integers)
    q.answerEncoding = new HashMap<BigInteger, BigInteger>();
    q.answerDecoding = new HashMap<BigInteger, BigInteger>();

    // enumerate possible answers
    for (BigInteger answer : q.getAvailableAnswers()) {
        BigInteger encodedAnswer = BigInteger.valueOf(numAnswers);
        answerEncoding.put(answer, encodedAnswer);
        answerDecoding.put(encodedAnswer, answer);
        numAnswers++;
    }

    // number of bits needed for this question is ceil(ld(numbits))
    int numbits = BigInteger.valueOf(numAnswers-1).bitLength();
    // create mask as described above
    q.mask = BigInteger.ONE
        .shiftLeft(numbits)
        .subtract(BigInteger.ONE)
        .shiftLeft(bitpos);
    // remember how many bits before the first bit
    q.bitshift = bitpos;
    // increase number of bits already used
    bitpos += numbits;
}
```

As we can see, the first question we process now gets assigned to the lowest bits. This is because we do not know in advance how many bits we will need altogether. Thus, starting with the lowest bits and adding additional digits as needed is a lot more convenient than first calculating the total that we will need. This moves the first processed question to the “last” place in the fingerprint, which is irrelevant in practice since the order in which we process the questions is arbitrary anyway. (There does not exist any designated ordering of the questions in  $\mathcal{Q}$ .)

<sup>2</sup>Public non-static fields are immensely hazardous in practice, but useful for keeping pseudocode readable. Do not try this at home.

Now that we have prepared the questions, we can calculate the fingerprint of a situation as follows:

```

BigInteger fingerprint = BigInteger.ZERO;
for (Question q : questions) {
    BigInteger answer = q.getAnswer(s);
    BigInteger encodedAnswer = q.answerEncoding.get(answer);
    BigInteger encodedAndShiftedAnswer = encodedAnswer.shiftLeft(q.bitshift);
    fingerprint = fingerprint.or(encodedAndShiftedAnswer);
}

```

For looking up the answer for a situation  $s$  to a question  $q_i$ , we are only interested in the bits corresponding to that question. We therefore:

1. set all bits of the fingerprint to 0 except for those that correspond to the question;
2. shift the remaining part to the right (depending on the position of the question in the fingerprint, i.e.,  $q.bitshift$ ) so that we get an integer in the range from 0 to  $a_i - 1$ ; and
3. look up which original answer this answer number refers to.

In code it looks like this:

```

BigInteger onlyRelevantBits = s.fingerprint.and(q.mask);
BigInteger shiftedToTheRight = onlyRelevantBits.shiftRight(q.bitshift);
BigInteger originalAnswer = q.answerDecoding.get(shiftedToTheRight);
return originalAnswer;

```

Or, a one line version that does the same at the cost of some readability:

```

return q.answerDecoding.get(s.fingerprint.and(q.mask).shiftRight(q.bitshift));

```

For checking whether a single (question, answer) tuple is satisfied, we could simply retrieve the answer for the question in the way described above and compare. However, we can also see that a fingerprint fulfils this if and only if the bits that correspond to the question we have to check have the value that corresponds to the demanded answer. We can therefore go the other way round: If the question has the demanded answer, we know what the values of the fingerprint for the bits corresponding to the questions must be. (For example, if in Example 6.36 a situation should have answer 3 to the first question, then the two leftmost bits of its fingerprint must be 11.) We calculate which values the fingerprint would have to have at the bit positions corresponding to this question, and verify that it indeed has those values there:

```

BigInteger expectedValues = q.answerEncoding.get(a).shiftLeft(q.bitshift);
BigInteger restrictedFingerprint = s.fingerprint.and(q.mask);
return restrictedFingerprint.equals(expectedValues);

```

The first statement calculates the values that the fingerprint should contain at the positions for the question if it has the demanded answer. The second statement sets all bits of the fingerprint that we do not need to check to 0. After that, we just compare. (We could alternatively XOR the two values and check whether the result is 0.)

For checking whether a whole list of (question, answer) tuples is satisfied, we again could either simply retrieve the answers for all questions in the way described above and compare, or take the approach used for verifying one tuple: Each (question, answer) tuple demands for a different part of the fingerprint what the values for that part have to be.

We therefore first go through the list of (question, answer) tuples and find out which parts of the fingerprints must be verified, and what values they must have. We represent this by creating a query bitmask and a query content in the following way:

- Each bit in the query bitmask is 1 if and only if the corresponding bit in the fingerprint has to have a certain value.
- Each bit in the query content contains the value that the corresponding bit in the fingerprint has to have, if we have to verify this bit, or 0 if we do not have to check this bit of the fingerprint.

Accordingly, the fingerprint fulfils all (question, answer) tuples if and only if for each bit for which the query bitmask is 1, the query content equals the fingerprint.

In pseudocode that looks as follows (using a data structure `Tuple` to represent (question, answer) tuples):

```
// create query bitmask and query content
BigInteger queryMask = BigInteger.ZERO;
BigInteger queryContent = BigInteger.ZERO;
for (Tuple tuple : tuples) {
    queryMask = queryMask.or(tuple.question.mask);
    BigInteger encodedAnswer = tuple.question.answerEncoding.get(tuple.answer);
    BigInteger encodedAndShiftedAnswer =
        encodedAnswer.shiftLeft(tuple.question.bitshift);
    queryContent = queryContent.or(encodedAndShiftedAnswer);
}

// check whether conditions is satisfied
BigInteger restrictedFingerprint = s.fingerprint.and(queryMask);
return restrictedFingerprint.equals(queryContent);
```

This algorithm requires an example for understanding.

**Example 6.38.** We will reuse Example 6.9. Let us assume that  $a, b, c$  is the ordering of the questions.

For  $a$  and  $b$ , we need 1 bit each, whereas for  $c$  we need 2 bits. Altogether we will therefore have 4 bit long fingerprints.

- We process question  $a$  first, therefore  $a$  gets the lowest bit. The mask for  $a$  therefore is 0001. Let us assume that we map “true” to 1 and “false” to 0.
- The mask for  $b$  is 0010. Let us assume that we again map “true” to 1 and “false” to 0.
- The mask for  $c$  is 1100. Let us assume that we map “false” to 00, “true” to 01, and “maybe” to 10.

For situation 2, the answers are “true” for question  $a$ , “false” for question  $b$  and “maybe” for question  $c$ . The fingerprint for situation 2 therefore is 1001. (With some spaces for readability: 10 0 1. The highest two bits contain the answer to  $c$ , the next one the answer to  $b$ , and the last one the answer to  $a$ .)

The complete list of fingerprints:

Situation	Fingerprint
1	0111
2	1001
3	0111
4	0001
5	0011
6	1000
7	0110
8	1000
9	1010
10	0000

Let us assume we want to retrieve the answer to question  $b$  for situation 7. Then we would first look at the fingerprint of situation 7, which is 0110. We know that the third bit from the left corresponds to question  $b$ . Since the other bits are not interesting for us at the moment, we can grey them out a little: 0110.

The interesting bit is 1 in our case, and we know that 1 for question  $b$  corresponds to “true”.

What the algorithm described above does is just the more mathematical version of this:

1. First, we perform a bitwise AND with the bitmask 0010 of  $b$ , which removes all irrelevant bits of the fingerprint 0110 and results in 0010.
2. We know that  $b$  starts at the second-lowest bit, therefore we shift the previous result 1 to the right and get 001, or, without leading zeroes, simply 1.
3. In our mapping, we have stored that 1 corresponds to “true” and 0 corresponds to “false”. Therefore we know that the answer of 7 for  $b$  is “true”.

To give another example: If we wanted to retrieve the answer for  $c$  in situation 1, we would again look at the fingerprint 0111, then first apply the bitmask 1100 and get 0100, then shift two bits to the right to get 01, and finally look up that 01 is mapped to “true” for question  $c$ .

Let us now assume that we want to vice versa find out whether situation 3 has answer “maybe” for question  $c$ . This can only be the case if the two leftmost bits of the fingerprint are 10. To check this, we do the same as before in the opposite direction:

1. “maybe” is mapped to 10 for question  $c$ .
2. We shift this 2 bits to the right to get the expected values 1000. Again, bits that can have any value are already greyed out.
3. We remember that we want to compare the highest 2 bits, that is, those corresponding to the bitmask 1100.
4. We calculate the bitwise AND of the bitmask 1100 with the fingerprint of 3, which is 0111. The result is 0100. Now all irrelevant bits have been set to 0, and all remaining bits should match the expectations.
5. We compare this to the expected values 1000 and see that they do not match. Therefore, 3 does not have “maybe” as answer to question  $c$ .

Finally, let us assume that we want to check whether situation 1 has answer “true” to question  $a$  and also answer “true” to question  $c$ . Again, we could check those separately. Following the algorithm from above, we however define a query mask and content instead. If the question to  $a$  is “true”, then the rightmost bit must be 1. If the question to  $c$  is true, then the leftmost two bits must be 01. Altogether, this gives the expected values 0101, where again grey values are irrelevant. In case the grey is not distinguishable from black on some printers, or in case we want it more mathematically, the bitmask is 1101, that is, all bits except for the second bit from the right must be checked.

What happens in the code above is just that:

1. Start with empty expected values and empty bitmask. That is, 0000 in grey and black representation – all values are still grey.
2. For question  $a$ , set the expected value of the rightmost bit to 1, and set the bitmask for that bit to 1 to indicate that it has to be verified. This leads to bitmask 0001 and expected values 0001 (or 0001 in grey and black representation).
3. For question  $c$ , set the expected value of the leftmost two bits to 01, and set the bitmask for them to 1 to indicate that they have to be verified. This leads to bitmask 1101 and expected values 0101 (or 0101 in grey and black representation).
4. We are done processing all demanded (question, answer) tuples now. This means that a situation satisfies them all if for each bit that is 1 in the bitmask 1101, the situation’s fingerprint has the same values as the expected values 0101. For the remaining bit that is 0 in the bitmask, the situation is allowed to have any value.
5. The fingerprint of situation 1 is 0111. Greying out irrelevant bits, this is 0111.
6. We can see that all relevant bits – that is, all black ones – match. To do this mathematically, we set all irrelevant bits of the fingerprint to 0 by calculating the bitwise AND with the bitmask. This leads to 0101.

7. In the expected values, we already know that all irrelevant bits are set to 0 by construction.
8. All we still have to do is compare the expected values 0101 with the masked parts of the fingerprint from step 6, which is 0101 as well. Therefore, situation 1 adheres to both requested (question, answer) tuples.

**Example 6.39.** In Example 6.36, we can make use of the fact that the ordering in  $\mathcal{Q}$  is irrelevant. We therefore process the “second question” first and the “first question” afterwards, so that they are assigned to the parts of the fingerprint as described in that example.

We can always arbitrarily choose the order in which questions are processed, and will for better readability make use of that in some of the following examples.

Let us have a quick look at performance, even though this part of the calculation is fairly manageable even on a slow computer.

**Lemma 6.40.**

- Fingerprints created with Algorithm 6.37 have a length  $f$  of at most  $q \cdot \lceil \text{ld}(a) \rceil$ . All occurring operations that access fingerprints therefore take  $O(q \cdot \text{ld}(a)) = O(f)$  time.
- Preparing the questions takes  $O(qa + qf)$  time.
- Generating the fingerprint for one situation takes  $O(qa + qf)$  time, and consequently for creating all of them we need  $O(s \cdot (qa + qf))$  time.
- Looking up the answer for one question takes  $O(f + a)$  time.
- Verifying one (question, answer) tuple takes  $O(f + a)$  time.
- Verifying  $k$  (question, answer) tuples takes  $O(k(f + a))$  time.

*Proof.*

- The maximum length of the fingerprint follows directly from its construction: since for each question we need at most  $\lceil \text{ld}(a) \rceil$  bits, and we have  $q$  questions, we get at most  $q \cdot \lceil \text{ld}(a) \rceil < q \cdot (\text{ld}(a) + 1) = q \cdot \text{ld}(a) + q = O(q \cdot \text{ld}(a))$  bits.

The only operations we use on fingerprints are bitwise AND, OR and XOR, shift left, shift right, and comparison. These all work in  $O(\text{length of the number}) = O(f)$  time.

- For preparing the questions, we have to do the following steps for each question:
  - For each of up to  $a$  answers, encode the answer, add it to some hash maps and increase the number. All of these run in  $O(1)$  time. This leads to a total time of  $O(a)$ .
  - Create a bit mask, which takes  $O(f)$  time.
  - Create new variables and perform some other calculations that run in  $O(1)$  time.

Altogether, that is an effort of  $O(q \cdot (a + f)) = O(qa + qf)$  time.

- For generating a fingerprint for one situation, we have to look at all questions, and for each question
  - calculate the answer in  $O(1)$  time (if we assume the runtime of  $\tau(s, q)$  to be constant for the game);
  - retrieve the encoded value of that answer in  $O(a)$  time (though the average runtime for retrieving values from a hash map is only  $O(1)$ );
  - shift the answer in  $O(f)$  time; and
  - calculate a bitwise OR with the calculated part of the fingerprint in  $O(f)$  time.

In sum, that is again  $O(q \cdot (a + f)) = O(qa + qf)$  time.

- Looking up an answer only requires operations on the fingerprint and a lookup in a hash map, thus  $O(f + a)$  time.
- Verifying an answer takes the same, thus also  $O(f + a)$  time.
- Verifying  $k$  answers finally takes  $k$  operations on the fingerprint and  $k$  lookups in the hash map, followed by 2 fingerprint operations. Altogether we therefore get a runtime of  $O(kf + ka + f) = O(k(f + a))$ .

□

We talked about the difference between  $\mathcal{A}_q$  and  $\overline{\mathcal{A}}_q$  earlier: in  $\overline{\mathcal{A}}_q$ , we only have the answers that actually occur, whereas  $\mathcal{A}_q$  can be a much bigger set. Our current algorithm however calculates the fingerprints from  $\mathcal{A}_q$ , which means that we might waste valuable space for answers that never occur. We therefore modify the preparation of the questions to actually calculate  $\overline{\mathcal{A}}_q$ . This does take some time, but is worth it on the long run (unless the  $\mathcal{A}_q$  already are very well defined, of course).

**Algorithm 6.41.**

```

int bitpos = 0; // number of bits used by previous questions

for (Question q : questions) {
    int numAnswers = 0; // number of different answers found for this question

    // map for enumerating possible answers (which may be arbitrary integers)
    q.answerEncoding = new HashMap<BigInteger, BigInteger>();
    q.answerDecoding = new HashMap<BigInteger, BigInteger>();

    // enumerate possible answers
    for (Situation s : situations) {
        BigInteger answer = q.getAnswer(s);
        if (!q.answerEncoding.containsKey(answer)) {
            BigInteger encodedAnswer = BigInteger.valueOf(numAnswers);
            answerEncoding.put(answer, encodedAnswer);
            answerDecoding.put(encodedAnswer, answer);
            numAnswers++;
        }
    }

    // number of bits needed for this question is (ceil(ld(numbits)))
    int numbits = BigInteger.valueOf(numAnswers-1).bitLength();
    // create mask as described above
    q.mask = BigInteger.ONE
        .shiftLeft(numbits)
        .subtract(BigInteger.ONE)
        .shiftLeft(bitpos);

    // remember how many bits before the first bit
    q.bitshift = bitpos;
    // increase number of bits already used
    bitpos += numbits;
}

```

*All other parts of the algorithm can stay the same as in Algorithm 6.37*

**Lemma 6.42.** *In Algorithm 6.41, preparing the questions takes  $O(qsa + qf)$  time.*

*Proof.* We have to do the following steps for each question:

- For each of the  $s$  situations:
  - Calculate the answer (in  $O(1)$  time),
  - check whether this answer has already been seen (in  $O(a)$  time), and potentially
  - add the new answer to the hash maps (in  $O(1)$  time).

Altogether, this results in a runtime of  $O(sa)$ .

- Create a bit mask, which takes  $O(f)$  time.
- Create new variables and some other calculations that run in in  $O(1)$  time.

Altogether, that is a runtime of  $O(q \cdot (sa + f)) = O(qsa + qf)$ . □



As we can see, this takes a lot longer than Algorithm 6.37 which had a runtime of  $O(\text{qa} + \text{qf})$ . Also, we cannot guarantee any benefit of it; if we already received perfect input sets with  $\mathcal{A}_q = \overline{\mathcal{A}}_q$ , we get exactly the same fingerprint length. Thus, this much more expensive algorithm does not guarantee us even a single bit of saved fingerprint space, let alone an order of magnitude that would be visible in worst-case estimates for the rest of the algorithm.

However, even this more expensive algorithm runs in acceptable time on a moderately fast computer<sup>3</sup>. The fingerprints on the other hand will be used many many times in the remaining algorithm, so every single bit of length reduction that we can squeeze out of this is usually worth it on the long run. Ultimately, deciding between Algorithm 6.37 and 6.41 will depend on the problem and on the quality of the  $\mathcal{A}_q$  that we can provide manually.

There is one last optimization that we should make for Algorithm 6.41: Both in the preparation of the questions and in the calculation of all situation fingerprints, we now access all situations and even calculate their answers for all questions. Thus, we basically do the same work twice. Merging these two steps does not change the asymptotic runtime, but it *is* a factor 2 in terms of calculation effort. We will skip the details of this merging here, since it does not provide any new insights. It is however implemented that way in the source code that accompanies this thesis.

Note that once the setup step of the questions is completed, fingerprints generated by Algorithm 6.37 and 6.41 work exactly the same way.

## 6.6.2 Expressing remaining situation lists in fingerprints

We promised that fingerprints would solve four problems. So far we saw in a lot of detail the solution to the first of these – avoiding to evaluate questions more than once.

The basic idea for solving the next one is to express the list of remaining situations in terms of these fingerprints. For example, we might reach one node  $N$  (following edges from the root) by first asking question  $q_{13}$  and getting answer 28, then asking  $q_8$  and getting answer 12, and finally asking  $q_{16}$  and getting answer 23. What we know is that all remaining situations in this node  $N$  give these answers to these three questions. That, however, also means that the part of the fingerprint that corresponds to these questions must be the same for all remaining situations in  $N$ . And vice versa, all situations for which these parts of the fingerprint have exactly these values are included in  $N$ .

Some definitions for convenience before we start:

**Definition 6.43.** Let  $F: \mathcal{S} \rightarrow \mathbb{N}$  denote the fingerprint as described in Algorithm 6.37 (or Algorithm 6.41), that is, for any situation  $s$ , let  $F(s)$  be the fingerprint of the situation.

Similarly, let  $M: \mathcal{Q} \rightarrow \mathbb{N}$  denote the bitmask of a question, that is, for any situation  $q$  let  $M(q)$  be the bitmask in which exactly the bits corresponding to  $q$  are set to 1.

Let  $m: \mathcal{Q} \times \mathcal{A} \rightarrow \mathbb{N}$  denote the mapping of the original questions to fingerprint values, i.e., for any question  $q$  and answer  $a$ , let  $m(q, a)$  be the integer that we get when looking up the bit value that  $a$  is mapped to and shifting it to the left according to the position of the question within the fingerprint.

Thus, in Example 6.36, the bitmask for the first question would be  $M(q_1) = 11000$ , and the possible values the first question could have for  $m$  are

$$\begin{aligned} m(q_1, 0) &= 00000 \\ m(q_1, 1) &= 01000 \\ m(q_1, 2) &= 10000 \\ m(q_1, 3) &= 11000 \end{aligned}$$

<sup>3</sup>For BlackBox with 5 dots on an  $8 \times 8$  grid, for example, which is one of the biggest games that we will ever look at in practice, calculating these optimized fingerprints takes around 11 minutes.

Now we summarize what we discussed in the following simple lemma:

**Lemma 6.44.** *Let  $((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$  be a list of (question, answer) tuples, and let  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$  be the set of situations that adhere to these.*

*We define*

$$M := M(q_1) \mid M(q_2) \mid \dots \mid M(q_k) ,$$

*which simply is a bitmask in which exactly the bits corresponding to the questions in the given list are 1, as well as*

$$m := m(q_1, a_1) \mid m(q_2, a_2) \mid \dots \mid m(q_k, a_k) ,$$

*which is 0 in all positions that are irrelevant, and for all other bits contains the value that they must have.*

*Then for every situation  $s \in \mathcal{S}$  it holds that  $s$  is element of  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$  if and only if*

$$F(s) \& M = m .$$

*(We use  $\&$  and  $\mid$  to represent bitwise AND and bitwise OR, respectively.)*

*Proof.* This follows directly from the construction:

If a situation  $s$  gives the demanded answers (and thus is part of  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$ ), then the corresponding parts of its fingerprint also have to match the values for these answers. In  $F(s) \& M$  we strip all other bits of the fingerprint that we do not need to verify. What remains therefore are exactly the expected values, so  $F(s) \& M = m$ .

Vice versa, if  $s$  gives a different answer for at least one question (and thus is not included in  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$ ), then the fingerprint will differ in that place, so  $F(s) \& M \neq m$ .  $\square$

This means that any list of remaining situations we will encounter can be expressed by such a combination of bitmask  $M$  and expected values  $m$ :

**Corollary 6.45.** *Let  $N$  be a node in the decision set graph and let  $\hat{\mathcal{S}}(N)$  be the set of remaining situations in  $N$ . Then  $\hat{\mathcal{S}}(N)$  can be described by a pair of bitmask  $M$  and expected values  $m$  as described in Lemma 6.44.*

*(That is, there exist values for  $M$  and  $m$  such that  $\hat{\mathcal{S}}(N) = \{s \in \mathcal{S} \mid F(s) \& M = m\}$ .)*

*Proof.* Each set of remaining situations that occurs in a node of the decision set graph also occurs as a set of remaining situations in a node in the behavioural game tree. By our construction, each such set of remaining situations can be described as  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$ .

Therefore, by Lemma 6.44, we can calculate bitmask  $M$  and expected values  $m$  that describe this set.  $\square$

However, we might notice two small flaws in this approach:

First, our way to calculate  $M$  and  $m$  is not constructive yet. We say that *some* list  $((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$  of (question, answer) tuples “exists”, but we do not give any way to find such a list.

Second, and closely related to that, the above list is not unambiguously defined, and for many nodes there will in fact exist multiple such lists. That is, while  $M$  and  $m$  unambiguously describe a certain list of situations  $\hat{\mathcal{S}}$ , this list can be described by several pairs  $(M, m)$ .

Let us look at an example for that:

**Example 6.46.** *We play a very simple version of the number guessing game: We have 10 situations  $\mathcal{S} = 1, 2, 3, \dots, 10$ , and we have 9 questions  $\mathcal{Q} = 1, 2, 3, 4, 5, 6, 7, 8, 9$  of the form “Is the value greater than  $x$ ?”.*

*For creating the fingerprints, we need one bit per question, and we sort the questions in such a way that question 1 will have the highest (leftmost) bit, question 2 the next one, and so on.*

This gives the following fingerprints:

<i>Situation</i>	<i>Fingerprint</i>
1	00000000
2	10000000
3	11000000
4	11100000
5	11110000
6	11111000
7	11111100
8	11111110
9	111111110
10	111111111

For example, we could have a player node in which we asked “Greater than 3?” and got a yes, and then asked “Greater than 7?” and got a no. This is satisfied by all situations that have “yes” as answer to the third and “no” as answer to the seventh question, that is, all situations whose fingerprint has a 1 at the third position from the left and a 0 at the seventh position from the left.

The corresponding values for  $M$  and  $m$  would be 001000100 and 001000000, respectively. In black and grey, that is 001000000.

Greying out irrelevant bits in the table the same way, we get the following:

<i>Situation</i>	<i>Fingerprint</i>
1	00000000
2	10000000
3	11000000
4	11100000
5	11110000
6	11111000
7	11111100
8	11111110
9	111111110
10	111111111

As we can see, situations 4, 5, 6 and 7 fulfil these requirements, which matches our expectations.

However, we might also have asked “Greater than 8?” in addition, and gotten a no there as well. Now we would have 001000110 for  $M$  and 001000000 for  $m$ . The grey and black table for this:

<i>Situation</i>	<i>Fingerprint</i>
1	00000000
2	10000000
3	11000000
4	11100000
5	11110000
6	11111000
7	11111100
8	11111110
9	111111110
10	111111111

Again, 4, 5, 6 and 7 fulfil the pattern.

Or, we could have asked “Greater than 1?” (yes), “Greater than 3?” (yes) and “Greater than 7?” (no), for  $M = 101000100$  and  $m = 101000000$ , and the following table:

Situation	Fingerprint
1	00000000
2	10000000
3	11000000
4	11100000
5	11110000
6	11111000
7	11111100
8	11111110
9	111111110
10	111111111

Not surprisingly, 4, 5, 6 and 7 are still the four situations that satisfy  $F(s) \& M = m$ .

### 6.6.3 Greatest common fingerprint mask

A definition for convenience:

**Definition 6.47** (fingerprint mask). We will refer to a pair  $(M, m)$  as described above as a FINGERPRINT MASK.

We will say that a fingerprint  $F(s)$  matches the fingerprint mask  $(M, m)$  if and only if  $F(s) \& M = m$ .

Since we want to describe sets of situations with these fingerprint masks, also the next definition will prove useful:

**Definition 6.48.** Let  $(M, m)$  be a fingerprint mask, and let  $\hat{S} \subseteq \mathcal{S}$  be a set of situations. Then we define

$$\hat{S}(M, m) := \{s \in \hat{S} \mid F \& M = m\}$$

as the subset of  $\hat{S}$  with the situations that match  $(M, m)$ .

Looking at runtimes, we get the following lemma:

**Lemma 6.49.** Let  $(M, m)$  be a fingerprint mask and  $\hat{S} \subseteq \mathcal{S}$  a set of situations. Then calculating  $\hat{S}(M, m)$  takes  $O(|\hat{S}| \cdot f)$  time.

*Proof.* For checking whether a single situation  $s$  matches  $(M, m)$ , we need to calculate whether  $F(s) \& M = m$ . We have  $F(s)$  already stored, so that does not take any calculation time. This leaves the operations on the fingerprints and bitmasks, which take  $O(f)$  time.

Doing this for all  $|\hat{S}|$  situations results in a runtime of  $O(|\hat{S}| \cdot f)$ .  $\square$

Finally, for the following proofs we might need a way to access individual bits:

**Definition 6.50.** Let  $m$  be any integer. Then  $m[k]$  denotes the bit of  $m$  (in binary representation) at the  $k$ -th position from the right. That is,  $m[0]$  denotes the rightmost bit.

What is still a little annoying for us is that there exist different fingerprint masks  $(M, m)$  that describe the same set of remaining situations. Since the set of remaining situations is our unique identifier for player nodes and stored results, it would be nice if we could also find values for  $M$  and  $m$  in such a way that  $(M, m)$  and  $(M', m')$  describe the same set of remaining situations if and only if  $M = M'$  and  $m = m'$ .

We looked at quite a few examples already to properly motivate the next step. What we can easily see in Example 6.46 is that even though we only demanded two or three bits to have certain values, the situations that fulfilled these requirements also had some other bits in common. For example, in the first case we only demanded the third bit to be 1 and the seventh bit to be 0. However, all values that satisfied these conditions also had 1 as answer for questions (bits) 1 and 2, and 0 as answer for questions (bits) 8 and 9. (This is also obvious from the game itself: any number that is larger than 3 is also larger than 1 and 2.)

This gives us the idea to calculate which bits are common to all situations in a set  $\hat{S} \subseteq \mathcal{S}$ .

**Definition 6.51** (greatest common fingerprint mask). Let  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  be a set of situations.

We define a bitmask  $M$  to be 1 for each bit for which the fingerprints of all situations in  $\hat{\mathcal{S}}$  have the same value, and 0 for each bit for which there exist at least two fingerprints in  $\hat{\mathcal{S}}$  that have different values at this position.

We define  $m$  to be 0 in all bits where  $M$  is 0. In all other bits, we define  $m$  to have the bit value that all situations in  $\hat{\mathcal{S}}$  have at this position.

We call  $(M, m)$  the greatest common fingerprint mask for  $\hat{\mathcal{S}}$ .

**Example 6.52.** In Example 6.46, all remaining situations in the set  $\{4, 5, 6, 7\}$  that we looked at (and for which we found several fingerprint masks describing it) have in common that they have 1 for the first three bits from the left, and 0 for the last three bits on the right. The other bits had different values among the situations in  $\{4, 5, 6, 7\}$ . Therefore, by our definition  $M = 111000111$  and  $m = 111000000$ . That is, the first three bits must be checked and must be 1. The next three bits are allowed to have any value. The last three bits must be checked and must be 0.

We see that  $\{4, 5, 6, 7\}$  is exactly the set of situations that fulfil this.

As we can see, Definition 6.51 only depends on the set  $\hat{\mathcal{S}}$  of situations. Now we will show that it also has some nice properties.

**Lemma 6.53.** Let  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  be a set of situations, and let  $(M, m)$  be the greatest common fingerprint mask of  $\hat{\mathcal{S}}$ .

Then  $\hat{\mathcal{S}} \subseteq \mathcal{S}(M, m)$ .

*Proof.* Let  $s$  be any situation in  $\hat{\mathcal{S}}$ . We will show that  $F(s)$  matches  $(M, m)$ .

Let us look at the  $k$ -th bit in  $s$ , denoted by  $F(s)[k]$ . If that bit is 0 in  $M$  (that is,  $M[k] = 0$ ), then it will not be checked, so everything is fine. (Mathematically speaking,  $(F(s) \& M)[k]$  will also be 0, and by definition,  $m[k]$  is 0. Therefore,  $(F(s) \& M)[k] = m[k]$ .)

If the bit is 1 in  $M$  instead (that is,  $M[k] = 1$ ), then this means that all situations  $s' \in \hat{\mathcal{S}}$  have the value stored in  $m[k]$  at this position. That is,  $F(s')[k] = m[k]$  for all situations  $s'$ . In particular, this therefore also applies to  $s$ . We get

$$\begin{aligned} F(s)[k] \& M[k] &= F(s)[k] \& 1 \\ &= F(s)[k] \\ &= m[k] \end{aligned}$$

and everything is shown.

Since  $(F(s) \& M)[k] = m[k]$  holds for each bit  $k$ , we also get  $F(s) \& M = m$ . Therefore,  $s \in \mathcal{S}(M, m)$ .  $\square$

Using these greatest common fingerprint masks, we can now finally describe sets of remaining situations:

**Theorem 6.54.** Let  $N$  be a node in the decision set graph, and let  $\hat{\mathcal{S}}(N)$  be the set of remaining situations in  $N$ . Let  $(M, m)$  be the greatest common fingerprint mask for  $\hat{\mathcal{S}}(N)$ .

Then  $\hat{\mathcal{S}}(N) = \mathcal{S}(M, m)$ .

*Proof.* We have to show that each situation contained in  $\hat{\mathcal{S}}(N)$  is contained in  $\mathcal{S}(M, m)$ , and that each situation not contained in  $\hat{\mathcal{S}}(N)$  is not contained in  $\mathcal{S}(M, m)$  either. Lemma 6.53 already shows the first of these two directions.

As seen in the proof of Corollary 6.45, there exists a list of questions  $q_i$  and answers  $a_i$  such that

$$\begin{aligned} \hat{\mathcal{S}}(N) &= \hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k)) \\ &= \{s \in \mathcal{S} \mid \tau(s, q_1) = a_1, \tau(s, q_2) = a_2, \dots, \tau(s, q_k) = a_k\} . \end{aligned}$$

Let us now look at a situation  $s' \in \mathcal{S} \setminus \hat{\mathcal{S}}(N)$ , that is, a situation that is *not* included in  $\hat{\mathcal{S}}(N)$ . This means that there must exist at least one question  $q_i$  from this list of given questions and answers such that  $\tau(s', q_i) = a'_i \neq a_i$ .

Consequently, the situation has a different answer for one question. However, all situations in  $\hat{\mathcal{S}}(N)$  have the same answer, so this part of the fingerprint is fixed and will be checked in the greatest common fingerprint mask of  $\hat{\mathcal{S}}(N)$ .

Mathematically,  $m(q_i, a'_i) \neq m(q_i, a_i)$ , since by our construction, different answers result in different fingerprints.

Furthermore, from the construction of the fingerprints we see that  $F(s') \& M(q_i) = m(q_i, a'_i)$ .

However, for all situations  $s \in \hat{\mathcal{S}}(N)$ , it holds that  $F(s) \& M(q_i) = m(q_i, a_i)$ : all such situations have identical fingerprints for the bits that correspond to the question  $q_i$ . Consequently,  $M$  is 1 for all bits that correspond to this part of the fingerprint, and  $m$  equals the values for these positions.

Mathematically, this means

$$M \& M(q_i) = M(q_i)$$

and

$$F(s) \& M(q_i) = m \& M(q_i) = m(q_i, a_i)$$

for all  $s \in \hat{\mathcal{S}}(N)$ .

In order for  $s'$  to match  $(M, m)$ , it would need to satisfy  $F(s') \& M = m$ . However,

$$\begin{aligned} F(s') \& M = m & \implies \\ F(s') \& M \& M(q_i) = m \& M(q_i) & \implies \\ F(s') \& M(q_i) = m(q_i, a_i) & \implies \\ m(q_i, a'_i) = m(q_i, a_i) & , \end{aligned}$$

which is a contradiction to our assumptions. □

Note that we needed to make use of the property that all sets of remaining situations that occur in decision set graphs can be represented as  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$ . If we took any set  $\hat{\mathcal{S}} \subseteq \mathcal{S}$  that does not have this property, the greatest common fingerprint mask might allow additional situations. To give an example:

**Example 6.55.** *Let us assume we have three situations with  $F(s_1) = 01$ ,  $F(s_2) = 10$  and  $F(s_3) = 11$ . Let us look at  $\hat{\mathcal{S}} = \{s_1, s_2\}$ .*

*For the greatest common fingerprint mask we get  $M = 00$  and  $m = 00$ . Therefore, all three situations match  $(M, m)$ , and we get  $\mathcal{S}(M, m) = \{s_1, s_2, s_3\}$ .*

Returning to our previous example, we can now have a look at the greatest common fingerprint mask:

**Example 6.56.** *Let us continue Example 6.46, and let us assume we are in the last case again, in which we had asked questions 1 (yes), 3 (yes) and 7 (no).*

*While we know that the fingerprint mask  $(M, m)$  that we used there is not unambiguous, it was quite useful for finding the situations that matched the requirements. Therefore, we will continue using it for that purpose.*

*Therefore, we have  $M = 101000100$  and  $m = 101000000$ , and the following table:*

Situation	Fingerprint
1	00000000
2	10000000
3	11000000
4	11100000
5	11110000
6	11111000
7	11111100
8	11111110
9	11111111
10	11111111

*We see that situations 4, 5, 6 and 7 match  $(M, m)$ , so we have  $\hat{\mathcal{S}} = \mathcal{S}(M, m) = \{4, 5, 6, 7\}$ .*

Now we calculate the greatest common fingerprint mask for this  $\hat{S}$ . Let us look at a table only containing these four situations and their fingerprints:

Situation	Fingerprint
4	111000000
5	111100000
6	111110000
7	111111000

For calculating the greatest common fingerprint mask, we need to determine which bit values are common to all situations in this set, and what their values are. Here, we can see that the first three bits are always 1, and the last three bits are always 0. The three bits in the middle each have both values at least once.

Hence we get  $M' = 111000111$  and  $m' = 111000000$  for the greatest common fingerprint mask  $(M', m')$ .

Using this to look at the entire table again, we get the following:

Situation	Fingerprint
1	000000000
2	100000000
3	110000000
4	111000000
5	111100000
6	111110000
7	111111000
8	111111100
9	111111110
10	111111111

As we can see, 4, 5, 6 and 7 still match the fingerprint mask, and no other values do.

We can also see that  $(M', m')$  only depended on the situations in  $\hat{S}$ , thus we would have gotten the same result, no matter with which of the initial fingerprint masks  $(M, m)$  from Example 6.46 we would have started.

This allows us to describe each set of remaining situations in a well-defined way; that is, given the same set of remaining situations, we always get the same greatest common fingerprint mask.

These greatest common fingerprint masks additionally guarantee uniqueness in the opposite way:

**Theorem 6.57.** *Let  $(M, m)$  be the greatest common fingerprint mask of a player node  $N$ , and  $(M', m')$  the greatest common fingerprint mask of a player node  $N'$ .*

*Then  $N = N'$  if and only if  $(M, m) = (M', m')$ .*

*Proof.* If  $N = N'$ , then  $(M, m)$  and  $(M', m')$  are created from the same set of remaining situations and are therefore identical according to the discussion above.

If  $(M, m) = (M', m')$ , then  $\mathcal{S}(M, m) = \mathcal{S}(M', m')$ . Therefore,  $N$  and  $N'$  have the same set of remaining situations, which can only be the case if they are the same node.  $\square$

Again, we needed  $(M, m)$  and  $(M', m')$  to be greatest common fingerprint masks of player nodes. If they simply are some fingerprint masks, there is no guarantee whatsoever about  $\mathcal{S}(M, m)$  and  $\mathcal{S}(M', m')$ . We have seen in Example 6.46 that there exist plenty of cases where  $(M, m) \neq (M', m')$ , but  $\mathcal{S}(M, m) = \mathcal{S}(M', m')$ .

This means that there exists a bijection between player nodes and their greatest common fingerprint masks. Consequently, we can now store and use these fingerprint masks instead of directly handling sets of situations.

Let us finally have a look at the calculation of this greatest common fingerprint:

**Algorithm 6.58.** Let  $\hat{S} = \{s_1, s_2, \dots, s_k\}$  be a set of situations. If all situations in  $\hat{S}$  have the same value for one bit, then the first situation in the list must obviously have this value as well. We therefore simply define  $c := F(s_1)$  for any situation  $s_1 \in \hat{S}$  as a constant with which we will compare all other fingerprints.

For convenience, we keep track of the positions in which we have already seen at least two different values. (This is exactly the bitwise inverse of  $M$ .) In the beginning we get  $d_0 = 0$  (where  $d_i$  denotes the state after having looked at  $i$  fingerprints).

We look at each situation  $s$ . By calculating  $F(s) \hat{\wedge} c$ , we get a number for which each bit is 1 if and only if the corresponding bits in  $F(s)$  and  $c$  are different. (We use  $\hat{\wedge}$  to denote bitwise XOR.) Therefore, the bits that are 1 here are exactly the bits for which we have different values in  $F(s_1)$  and  $F(s)$ , and which therefore cannot be common to all situations any more.

We therefore calculate as follows:

$$d_i = d_{i-1} \mid (F(s_i) \hat{\wedge} c)$$

In the end, we get a value  $d_k$  in which those bits are 1 that have been 1 for at least one of the  $k$  comparisons with  $c$ .

We claim that  $d_k$  is exactly the inverse of the  $M$  we are looking for: If a bit is 1 in  $d_k$ , then there has been at least one situation  $s_i$  such that  $s_i$  and  $s_1$  had different value for that bit. If a bit is 0, on the other hand, then all bits at that position were the same as the bit in  $c$ , and thus the same in all situations.

Therefore, we set  $M := \neg d_k$  (where  $\neg$  denotes bitwise NOT).

Since all situations have the same values for these positions, we can simply retrieve those values from any of the situations, set all undefined values to 0, and get  $m := M \& F(s_1)$ .

Algorithmically, this could look like this:

```

BigInteger comparisonFingerprint = validSituations[0].fingerprint;
BigInteger differentValuesSeen = BigInteger.ZERO;
for (Situation s : validSituations) {
    BigInteger comparisonResult = comparisonFingerprint.xor(s.fingerprint);
    differentValuesSeen = differentValuesSeen.or(comparisonResult);
}
M = differentValuesSeen.not();
m = comparisonFingerprint.and(M);

```

**Lemma 6.59.** Algorithm 6.58 takes  $O(\hat{s} \cdot f)$  time, where  $\hat{s}$  is the number of situations from which the greatest common fingerprint mask is calculated, i.e., the size of `validSituations` in the code above.

*Proof.* By Lemma 6.40, each operation on the bit masks takes  $O(f)$  time, and we have  $\hat{s}$  such operations.  $\square$

## 6.6.4 Calculating children from fingerprint masks

Those were problems number one and two, so it is time to tackle number three: we want to efficiently calculate the children that a node has. Since we use fingerprint masks rather than lists of situations to describe nodes, we ideally want to directly calculate the fingerprint mask of a child without having to go through the list of situations. We will need different approaches here for player nodes and for computer nodes.

### Children of player nodes

For performance comparison later, let us first determine the runtime of calculating the children of a player node without using fingerprints.

**Lemma 6.60.** The children of a player node can be calculated in  $O(qs)$  time.

*Proof.* For each question, we need to check whether two different answers still exist. We can do this by evaluating the question for each remaining situation. In the worst case, all situations except the last one have the same answer, so we really need to go through the entire list of situations. Evaluating a question for a situation takes  $O(1)$  time.

This gives a total runtime of  $O(qs)$ .  $\square$



Not only is  $O(qs)$  huge, we also need to evaluate a lot of questions for a lot of answers, which might only have an effort of  $O(1)$ , but often with a very large constant.

Let us therefore look at how we would do this with fingerprint masks instead.

In a player node, we always get one outgoing edge for the estimate. In addition, we get one outgoing edge for each question for which at least two different answers occur among the remaining situations. This means that for questions, we would like to have an easy way to determine whether there still exist two different answers. Fortunately, greatest common fingerprint masks basically give us this for free:

**Theorem 6.61.** *Let  $N$  be a player node and let  $\hat{S}(N)$  be the set of remaining situations in  $N$ . Let  $(M, m)$  be the greatest common fingerprint mask for  $\hat{S}(N)$ .*

*Let  $q_i \in \mathcal{Q}$  be a question, and let  $M(q_i)$  be the mask for this question, using the same terminology as in the previous section.*

*Then  $\hat{S}(N)$  contains two situations with different answers to  $q_i$  if and only if  $M \& M(q_i) \neq M(q_i)$ , that is, if there exists at least one bit corresponding to question  $q_i$  for which two different values are still possible.*

*Proof.* This is fairly obvious: If all situations in  $\hat{S}(N)$  have the same answer for  $q_i$ , then all bits corresponding to  $q_i$  are fixed. Vice versa, if all bits corresponding to  $q_i$  are fixed, then all situations must have the same answer.  $\square$

The runtime of checking this is obviously very low, especially if we do not always access the entire fingerprint.

**Theorem 6.62.** *Using fingerprints, calculating all children of a player node can be done in  $O(f + q)$  time.*

*Proof.* The trivial implementation would be to calculate  $M \& M(q_i)$  for each question  $q_i$ . However, as seen in Lemma 6.40, operations on the fingerprints take  $O(f)$  time, leading to a runtime of  $O(qf)$ .

We can however see that the parts of  $M$  that we access are disjoint: for each question  $q_i$ , we only need to know the values of the bits corresponding to  $q_i$ . Therefore, each bit of  $M$  is read only once, which results in a runtime of  $O(f + q)$  (since we still need to access each question once to find out which bits belong to it).  $\square$

This theoretical runtime is nice, but practically, implementing the algorithm in such a way that it accesses individual bits creates enough overhead to not be worth it. Typical fingerprints that we work with will have a length of 20 to 2000 bit; anything that is above that will probably be way out of reach for calculating anyway. Therefore,  $f$  is close to constant, and in practice we will usually just calculate the entire  $M \& M(q_i)$  for each  $q_i$ , in spite of its theoretically worse runtime of  $O(qf)$ .

### Children of computer nodes

For computer nodes, we want to know which answers can still occur at all among the remaining situations. In this case, we do not get the exact details only out of the fingerprint. However, we can at least rule out some answers for which individual bits do not match. We also will see that creating children with 0 remaining situations does not really harm us.

**Theorem 6.63.** *Let  $N$  be a computer node, and let  $(M, m)$  be the greatest common fingerprint mask that describes its list of remaining situations. Let  $q$  be the question to be answered, and let  $a_1 \in \mathcal{A}_q$  be one possible answer to that question.*

*If  $m \& M(q) \neq m(q, a_1) \& M$ , then no situation in  $\mathcal{S}(M, m)$  has  $a_1$  as answer to  $q$ .*

*Proof.* Once you see what this expression does, it is fairly obvious. Basically we simply compare those bits that are relevant to question  $q$  and defined in fingerprint mask  $(M, m)$ . If they are defined in the fingerprint mask in such a way that a situation with that answer would not match the fingerprint, then there is no child for that question. (For example, if a question corresponds to three bits and we know the middle bit has to be 0, then the answer 111 cannot apply to any situation that matches the fingerprint mask.)

On the left side, we take the expected values and set all values that do not correspond to  $q$  to 0. Values that are unknown are already 0 by definition.

On the right side, we take the values corresponding to answer  $a_1$  and set those bits to 0 that are not restricted by  $(M, m)$ . All bits that do not correspond to  $q$  are already 0 in  $m(q, a_1)$  by definition.

That means that the only values that are not set to 0 on both sides are those that have to be verified. On the left side, they are set to the values they are supposed to have, and on the right side, they are set to the values that they have for the given answer.

Let us assume that  $\mathcal{S}(M, m)$  contained a situation  $s$  with  $r(s, q) = a_1$ . Then the fingerprint of  $s$  would have the corresponding value at this position, that is,  $F(s) \& M(q) = m(q, a_1)$ .

Consequently, this would mean

$$\begin{aligned} F(s) \& M = m & \implies \\ F(s) \& M(q) \& M = m \& M(q) & \implies \\ m(q, a_1) \& M = m \& M(q) , \end{aligned}$$

which is a contradiction to the precondition. Therefore, no situation in  $\mathcal{S}(M, m)$  can have  $a_1$  as answer to  $q$ . □

For more clarity, let us look at an example:

**Example 6.64.** *Let us assume our fingerprint is 5 bits long, of which the leftmost 3 belong to question a, the next bit to question b and the last bit to question c. After asking c and getting 1 as answer, we have situations with the following fingerprints left:*

```
00111
01101
10101
00111
10011
10111
```

*The biggest common fingerprint mask therefore is  $M = 00001$  and  $m = 00001$ .*

*Next, we ask question b and get answer 1. Two of the situations do not have 1 as their bit on position 4, so they go away, and we have four fingerprints left:*

```
00111
00111
10011
10111
```

*The biggest common fingerprint mask now is  $M = 01011$  and  $m = 00011$ .*

*Colouring the table, we get:*

```
00111
00111
10011
10111
```

*Therefore, the middle bit of question a is already set, even though we have not asked a yet.*

*Let us look at 3 answers to a. Answer 101 can still occur, and also does so. Answer 110 is not possible any more, because we already know that the middle bit has to be 0. Answer 000 is not ruled out by Theorem 6.63, but does not occur any more either.*

Thus, Theorem 6.63 can rule out some impossible answers, but not all of them.

However, creating greatest common fingerprint masks for children is really easy now:

**Algorithm 6.65.** *Let  $N$  be a computer node, and let  $(M, m)$  be the greatest common fingerprint mask that describes its list of remaining situations. Let  $q$  be the question to be answered.*

*For each answer  $a \in \mathcal{A}_q$  that is not ruled out by Theorem 6.63, create a new fingerprint mask  $(M', m')$  with  $M' = M \mid M(q)$  and  $m' = m \mid m(q, a)$ .*

*Calculate the greatest common fingerprint mask  $(M'', m'')$  of  $\mathcal{S}(M', m')$ . If it turns out during doing so that  $\hat{\mathcal{S}}(M', m')$  is an empty set, then this answer obviously does not occur any more, so we also do not need to look at a child for it. Otherwise, we get a player node as child which can be described by the greatest common fingerprint mask  $(M'', m'')$ .*

**Lemma 6.66.** *Algorithm 6.65 produces all children of a computer node.*

*Proof.* We will show that for each answer  $a$ , we either generate the same child that would be generated with the classical method, or skip  $a$  using Theorem 6.63.

Let  $N$  be a computer node,  $(M, m)$  the greatest common fingerprint mask of the remaining situations,  $q'$  the question to be answered, and let  $a'$  be an answer that is not ruled out by Theorem 6.63. As seen in the proof of Corollary 6.45, the list of remaining situations in  $N$  can be expressed as  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k))$ .

Consequently, the list of remaining situations in the child that we reach by following  $a'$  can be expressed as  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k), (q', a'))$ . Alternatively, we can write this as  $\hat{\mathcal{S}}((q_1, a_1), (q_2, a_2), \dots, (q_k, a_k)) \cap \mathcal{S}((q', a'))$ .

Looking at some fingerprint masks of those – not necessarily the greatest common fingerprint mask –, we see that the first term is described by  $(M, m)$ , whereas we can describe the second by  $(M(q'), m(q', a'))$ .

Therefore, a situation  $s \in \mathcal{S}$  is element of that set if and only if  $F(s) \& M = m$  and  $F(s) \& M(q') = m(q', a')$ .

Since we ruled out all answers that do not adhere to this beforehand, we know that  $M(q') \& m = M \& m(q', a')$ . That is, all bits that are defined both by  $(M, m)$  and by  $(M(q'), m(q', a'))$  are defined the same way in both of them.

Therefore, we claim that

$$(F(s) \& M = m \quad \text{and} \quad F(s) \& M(q') = m(q', a')) \iff F(s) \& (M \mid M(q')) = m \mid m(q', a').$$

For the one direction, let us assume that  $F(s) \& M = m$  and  $F(s) \& M(q') = m(q', a')$  hold. We perform a bitwise OR on both sides:

$$\begin{aligned} & \left. \begin{array}{l} F(s) \& M = m \\ F(s) \& M(q') = m(q', a') \end{array} \right\} \implies \\ & (F(s) \& M) \mid (F(s) \& M(q')) = m \mid m(q', a') \iff \\ & F(s) \& (M \mid M(q')) = m \mid m(q', a') \iff \\ & F(s) \& M'' = m'' \end{aligned}$$

For the other direction, let us assume that  $F(s) \& (M \mid M(q')) = m \mid m(q', a')$  holds. Then we do a bitwise AND with  $M$  on both sides, and get the first equation using that  $M \& m = m$ , that  $M \& m(q', a') = M(q') \& m$ , and that  $X \mid (X \& Y) = X$ .

$$\begin{aligned} & F(s) \& (M \mid M(q')) = m \mid m(q', a') \implies \\ & M \& F(s) \& (M \mid M(q')) = M \& (m \mid m(q', a')) \implies \\ & F(s) \& ((M \& M) \mid (M \& M(q'))) = (M \& m) \mid (M \& m(q', a')) \implies \\ & F(s) \& (M \mid (M \& M(q'))) = (m) \mid (M(q') \& m) \implies \\ & F(s) \& M = m \end{aligned}$$

Almost analogously, we get the other equation by doing a bitwise AND with  $M(q')$  on both sides:

$$\begin{aligned} & F(s) \& (M \mid M(q')) = m \mid m(q', a') \implies \\ & M(q') \& F(s) \& (M \mid M(q')) = M(q') \& (m \mid m(q', a')) \implies \\ & F(s) \& ((M(q') \& M) \mid (M(q') \& M(q'))) = (M(q') \& m) \mid (M(q') \& m(q', a')) \implies \\ & F(s) \& ((M(q') \& M) \mid M(q')) = (M \& m(q', a')) \mid (m(q', a')) \implies \\ & F(s) \& M(q') = m(q', a') \implies \end{aligned}$$

□

Let us compare the performance of calculating children using fingerprint masks to that of the the straight-forward implementation.

**Lemma 6.67.** *Let  $N$  be a computer node with remaining situations  $\hat{\mathcal{S}}$  and question  $q$  to be answered. Let  $\hat{s} = |\hat{\mathcal{S}}|$  and  $\mathbf{a}_q = |\mathcal{A}_q|$ .*

*Calculating the children of a computer node directly (without using fingerprint masks) takes  $O(\hat{s}\mathbf{a}_q)$  time.*

*Proof.* We could, theoretically, go through the list of answers and then for each answer find all situations in  $\hat{\mathcal{S}}$  that give this answer. Doing this, we would get a runtime of  $\Theta(\mathbf{a}_q\hat{s})$ .

We can however also go through the list of situations instead. Obviously, each situation will be added to exactly one child. Therefore, we can do the following:

We will create lists containing the situations that give the same answer. We start with no lists. For each situation  $s$ , find the answer  $a = \tau(s, q)$ . If we already have a list for this answer  $a$ , then add  $s$  to this list. Otherwise, create a list for  $a$  and add  $s$  there.

Adding an element to a list can be implemented to run in  $O(1)$  time, and creating a list takes  $O(1)$  time as well. The harder step is to find the list corresponding to some answer  $a$  in a short time.

One possible way to do this is to use hash maps that have answers as key and the lists for those answers as value. Insertion can be implemented to run in  $O(1)$  in the worst case, but lookup has a worst-case runtime of  $O(\mathbf{a}_q)$  (since there will be at most  $\mathbf{a}_q$  elements in the list). Altogether, this results in a worst-case runtime of  $O(\hat{s}\mathbf{a}_q)$  (and expected runtime of  $O(\hat{s})$ ).  $\square$

**Lemma 6.68.** *Let  $N$  be a computer node with greatest common fingerprint mask  $(M, m)$  and question  $q$  to be answered. Let  $\hat{s} = |\mathcal{S}(M, m)|$ .*

*Calculating the children of a computer node using greatest common fingerprint masks takes an effort of  $O(\mathbf{a}_q \cdot \hat{s} \cdot \mathbf{f} + \mathbf{a}_q^2)$ . If we do not know the list of remaining situations  $\mathcal{S}(M, m)$ , we additionally need to create this list for an additional calculation effort of  $O(\mathbf{s} \cdot \mathbf{f})$ .*

*Proof.* If we do not have the list  $\hat{\mathcal{S}} := \mathcal{S}(M, m)$  yet, we need to calculate it from  $\mathcal{S}$  and  $(M, m)$ . This takes  $O(\mathbf{s} \cdot \mathbf{f})$  time.

For each possible answer  $a \in \mathcal{A}_q$ , we have to do the following three steps:

1. We have to calculate  $M'$  and  $m'$ , which takes  $O(\mathbf{f})$  time for the bitwise operations on the masks, plus  $O(\mathbf{a}_q)$  time for looking up  $m(q, a)$ .
2. Then, we need to calculate  $\hat{\mathcal{S}}(M', m')$ , which takes  $O(\hat{s} \cdot \mathbf{f})$  time.
3. Finally, we need to calculate the greatest common fingerprint, which takes  $O(|\hat{\mathcal{S}}(M', m')| \cdot \mathbf{f})$  time.

In sum over all possible answers, the first step therefore takes  $O(\mathbf{a}_q\mathbf{f} + \mathbf{a}_q^2)$  time and the second one takes  $O(\mathbf{a}_q\hat{s}\mathbf{f})$  time.

For the third step, we know that the subsets of  $\hat{\mathcal{S}}$  for different answers are disjoint and their union is  $\hat{\mathcal{S}}$ . Therefore,  $\sum_{a \in \mathcal{A}_q} |\hat{\mathcal{S}}(M'(a), m'(a))| = |\hat{\mathcal{S}}|$ . Calculating a greatest common fingerprint mask for  $\hat{s}$  situations takes  $O(\hat{s}\mathbf{f})$  time, so we get altogether

$$\begin{aligned} \sum_{a \in \mathcal{A}_q} O(|\hat{\mathcal{S}}(M'(a), m'(a))| \cdot \mathbf{f}) &= O\left(\sum_{a \in \mathcal{A}_q} |\hat{\mathcal{S}}(M'(a), m'(a))| \cdot \mathbf{f}\right) \\ &= O(|\hat{\mathcal{S}}| \cdot \mathbf{f}) \\ &= O(\hat{s} \cdot \mathbf{f}) \end{aligned}$$

as the total effort for the third step in sum over all answers.

Altogether, this therefore gives a total runtime of  $O(\mathbf{a}_q\mathbf{f} + \mathbf{a}_q^2 + \mathbf{a}_q\hat{s}\mathbf{f} + \hat{s}\mathbf{f}) = O(\mathbf{a}_q\hat{s}\mathbf{f} + \mathbf{a}_q^2)$  (plus potentially the  $O(\mathbf{s}\mathbf{f})$  time for creating  $\mathcal{S}(M, m)$ ).  $\square$

At first glance, the approach that uses fingerprints appears to behave a lot worse than the original one – even without having to calculate the list, we get a runtime of  $O(\mathbf{a}_q \cdot \hat{s} \cdot \mathbf{f} + \mathbf{a}_q^2)$  instead of  $O(\hat{s}\mathbf{a}_q)$ . However, note that the factor  $\mathbf{f}$  is close to constant, since fingerprints will usually not be longer than 2000 bits. (Problems with larger fingerprints than that are almost impossible to calculate anyway.) Also,  $\mathbf{a}_q^2$  will in all practical cases be a lot smaller than  $\mathbf{f}$ . Assuming  $\mathbf{f}$  to be constant and  $\mathbf{a}_q^2$  to be smaller than  $\mathbf{a}_q\hat{s}\mathbf{f}$ , we get  $O(\mathbf{a}_q\hat{s})$  as runtime, and are thus equivalent to the original approach.

We will talk a bit more about performance comparison in Section 6.6.6 and Section 6.6.7.

### 6.6.5 Storing and retrieving known results

Last but not least, problem number four: we need a good way to store known results and retrieve them again.

Since we only store known results for player nodes, storing the list of remaining situations is sufficient to identify a node.

The naive way to handle this without fingerprints is to store tuples of (situation list, value). However, this is highly impractical: The lists take a lot of storage space, and for looking up the value for some situation list  $\hat{S}$ , we need to compare  $\hat{S}$  to all stored situation lists to find a matching one.

What we can do is to define a unique label string for each situation list.

Idea 1: We give each situation a number, and concatenate these with some separator. For example, if we have  $\hat{S} = \{3, 5, 8\}$ , we use 3\_5\_8 as label. We can enumerate the situations at the beginning of the program, so looking up the number of one situation takes  $O(1)$  time. Therefore, for a list of  $\hat{s}$  situations, it takes  $O(\hat{s})$  time to retrieve and concatenate these numbers. Each number is at most  $\lceil \text{ld}(s) \rceil$  long, resulting in a total length of at most  $\hat{s} \cdot (\lceil \text{ld}(s) \rceil + 1) - 1 < \hat{s} \cdot (\text{ld}(s) + 2) = O(\hat{s} \cdot \text{ld}(s))$ .

Idea 2: We use one bit per situation to indicate whether it is included in the list. For example, if we have a total of 10 situations, then  $\hat{S} = 3, 5, 8$  would be denoted by 0010100100. Here, we have a constant length of  $s$ . To create the label, we can either simply go through the list of situations  $\mathcal{S}$ , and check for each situation whether it is contained in  $\hat{S}$ , appending 0 or 1 as appropriate. If both lists are already sorted the same way (which is easy to achieve in our case at no additional cost, since in each node we only use subsets of situations), then this takes  $O(s)$  time. Alternatively, we can initialize the label to 0, and for each situation in  $\hat{S}$ , we look up the number of the situation and set the corresponding bit to 1. Runtime depends on how efficiently we can implement setting one bit in that number, but assuming that a runtime of  $O(1)$  is achievable, we can get a total runtime of  $O(\hat{s})$ .

Either way, the runtime is  $O(\hat{s})$ . The storage space is either  $O(\hat{s} \cdot \text{ld}(s))$ , or  $O(s)$ .

Once we have these labels, we can store them in some convenient data structure for dictionaries. Since this is the only really large data structure we will use in the entire program – the lists of situations, questions, answers and estimates are all tiny by comparison –, choosing a good data structure pays off here. We can take advantage of the fact that we do not need any deletions or modifications. What we want is good behaviour for insertion and lookup.

For the further analysis, let us assume that we use a data structure with *average* runtime  $O(1)$  for both insertion and lookup, such as, for example, a hash map.

For looking up a value for a list of  $\hat{s}$  situations, we therefore need to create the label for the list, and retrieve the value for that label. The total runtime is  $O(\hat{s})$ .

Now, let us look at greatest common fingerprint masks in contrast to this. Given a mask  $(M, m)$ , we can create a unique label by simply concatenating these two. Since they have fixed length, we do not even need a separator. For example, for  $M = 1101$  and  $m = 1001$ , we create the label 11011001. We do not need strings and can simply use numbers for this: if the length of the fingerprint is  $f$ , we can simply shift  $M$  left by  $f$  bits and use  $(M \ll f) \mid m$  as label (where  $\ll$  denotes a left-shift). Creating such a label therefore takes  $O(f)$ .

Since we only store greatest common fingerprint masks here, we know that two such masks are identical if and only if they refer to the same node. Obviously, this now also applies to the labels.

We again use the same data structure for storing these values, with the label defined above as key. Looking up a value now works in an expected runtime of  $O(f)$ : we need  $O(f)$  time for creating the label from the fingerprint mask, and expected constant time for retrieving the associated value.

The storage space for each key is  $O(f)$ .

### 6.6.6 A critical word on the performance of greatest common fingerprint masks

Let  $s = |\mathcal{S}|$ ,  $q = |\mathcal{Q}|$  and  $a = \max_{q \in \mathcal{Q}} |\mathcal{A}_q|$ .

While it is awesome that greatest common fingerprint masks are sufficient to describe the remaining situations in a node and in addition even have useful properties for calculating a node's children, we should keep in mind that calculating the greatest common fingerprint mask can take a lot of calculation effort, up to  $O(s \cdot f)$ . It might therefore occasionally make sense to keep an actual list of remaining situations *in addition to* the greatest common fingerprint mask.

Why keeping both?

For calculating children of a player node, using the greatest common fingerprint mask performs a lot better than the list of situations, since we do not need to evaluate even a single question, let alone find two situations that produce different answers. Calculating the children from the fingerprints takes  $O(f)$  time (with a small constant factor), whereas calculating them directly would take  $O(qs)$  time (with a large constant factor). Thus, we want to use fingerprint masks there.

Also, for storing and retrieving known results, fingerprint masks vastly outperform the original approach with  $O(1)$  expected time for insertion and lookup instead of  $O(s)$ , and  $O(f)$  memory usage instead of  $O(s)$ .

For calculating children of a computer node, however, the effort for calculating the greatest common fingerprint mask should not be underestimated. Let  $\hat{s}$  be the number of remaining situations in a computer node. Then the original implementation without fingerprints has a runtime of  $O(a_q \hat{s})$ , whereas the method that uses fingerprints has a runtime of  $O(a_q \cdot \hat{s} \cdot f) + O(sf)$ .

Explicitly keeping lists of remaining situations around can help to improve this performance by removing the second part of that sum. Without going into too much detail, we can create the corresponding list of remaining situations of a child during the calculation of the greatest common fingerprint mask at almost no additional cost.

Of course, keeping situation lists in addition to fingerprint masks does require some extra storage space. If we run the algorithm recursively, we have to keep one such list in memory for each level of recursion, that is, for at most  $q + 1$  levels (which is the maximal height from the root to a leaf that can occur).

Creation time for such lists can by the way be a good place to start tweaking performance – if we really create a list for each child of a computer node, we will create many such lists over time. Even though we will not be able to create a list with  $k$  elements in less than  $O(k)$  time, reducing the constant factor here really pays off.

### 6.6.7 Performance comparison

Finally, let us compare those two approaches: In the first approach, we do not use fingerprints, and instead work directly with the lists of remaining situations. In the second approach, we use fingerprint masks and keep the lists in addition.

The calculation step itself is the same in both approaches, therefore we only compare the runtimes of the parts that differ.

	Without fingerprints	With fingerprints
<b>Creating children of a player node</b>	$O(qs)$	$O(f)$
<b>Creating children of a computer node</b>	$O(a_q \hat{s})$	$O(a_q \hat{s} f + a_q^2)$
<b>Storing and retrieving known results</b>	$O(\hat{s})$ (average)	$O(f)$ (average)

Storage space is only interesting in the data structure for storing known results. Without fingerprints, we have  $O(\hat{s} \cdot \text{ld}(s))$  or  $O(s)$  per key, with fingerprints we have  $O(f)$  per key.

As mentioned before,  $f$  is in practice close to constant. If we therefore again assume  $f$  to be constant and  $a_q$  to be small, we get the following table instead:

	Without fingerprints	With fingerprints
<b>Creating children of a player node</b>	$O(qs)$	$O(1)$
<b>Creating children of a computer node</b>	$O(a_q \hat{s})$	$O(a_q \hat{s})$
<b>Storing and retrieving known results</b>	$O(\hat{s})$ (average)	$O(1)$ (average)

What we cannot see from these asymptotic runtimes yet are the constant factors hidden in them. Looking at the algorithm in detail, we can see that in the old approach, those “constant” operations were evaluating questions for situations, moving elements between lists, adding and retrieving things from hash maps, concatenating long strings as labels, and so on. All of these take relatively long. In the new approach on the other hand, the “constant” operations are for the most part bitwise AND, OR and XOR, things that are very fast on a computer.

Since this is nonetheless a bit handwaving, in Chapter 10 we will look at actual runtimes of these two algorithms for some small problems.

## 6.7 Caching known results

So far, we assumed that we store all known results for player nodes, but hinted already that in practice we might not have the main memory to do so.

We therefore have to decide which ones we want to store.

One very generic approach would be to set an upper limit for the number of results we can store. For deciding which ones to keep, we can, for example, keep those that were accessed most recently, or those that were accessed most often. Good caching strategies are a topic that fills several books and whole lectures, so we will not go into detail about it here.

We will however have a look at one heuristic approach that performs reasonably well for our particular problem:

### 6.7.1 Caching by node size

Let us have a rough look at the shape of the decision set graph. For each node, we define the size of the node as the number of remaining situations. As seen earlier, outgoing edges of player nodes lead to nodes with the same size, and outgoing edges of computer nodes to nodes with strictly smaller size. Thus, we can draw the graph in such a way that the height of each node equals its size, where the root is the highest node with size  $s = |\mathcal{S}|$ , and all edges go downward (if we consider computer nodes and estimate leafs to have  $(\text{size} - \epsilon)$  as height).

Let us estimate how many player nodes of each size we have. There is only one node of size  $s$ , the root itself. There can be at most  $s$  nodes of size  $s - 1$  – there are  $s$  ways to choose one node not included in the set. There are at most  $\binom{s}{2}$  nodes of size  $s - 2$ , and so on. For each size  $k$ , there are at most  $\binom{s}{k}$  nodes of that size. This means that we have very few nodes with very large size, very few nodes with very small size, and a lot of nodes with medium size.

Let us also look at how many ways there are to reach a node. For the root, there obviously is only one way: We start there. For nodes with size  $s - 1$ , there can exist at most  $q = |\mathcal{Q}|$  ways to get there, one for each question. (Since each computer node leads to a node with smaller size and we are only 1 below the maximum size, there can be at most one computer node on the way there.)

For nodes with size  $s - 2$ , there can be up to  $q(q - 1)$  ways to get there. For nodes with size  $s - 3$  there can be  $q(q - 1)(q - 2)$  ways, and so on. For a node with size  $s - k$ , there are  $\frac{s!}{(s-k)!}$  ways to reach this node. In the game from Theorem 5.11, we see that this number can really be reached.

Therefore, we basically have three different zones:

- Near the top of the decision set graph we have **few nodes** with a large size that are **accessed few times**,
- In the middle there is a **huge number of nodes** with a medium size that are **accessed moderately often**, and
- Towards the bottom there are **few nodes** with a small size that are **accessed very often**.

Ideally we would like to store all known results. However, a typical computer does not provide more than a few gigabytes of main memory. If we wanted to store more than that, then it would have to go to the hard disk, and the hard disk is a very uncomfortable place for keeping a cache.

If we therefore have to decide on some nodes for which we will store the values, then we will want to save those in the lower zone, since this replaces a lot of calculations with simple lookups. (Optionally, we might want to save a few from the highest zone in addition, since there are also few of them, and every node cached there spares us the calculation of a rather large subtree. We are however certainly not interested in caching nodes from the middle zone.)

There are two simple ways to determine the approximate size of a node. Either we use lists of situations anyway, then we can just look up the size of this list and thereby get the exact size of the node. If we do not have those lists and work only with greatest common fingerprint masks, then the number of bits in the bitmask  $M$  that are 1 is a good indicator of the size: by and large, the more bits are set, the smaller the size of the corresponding node.

Two short lemmata to describe this:

**Lemma 6.69.** *Let  $(M, m)$  be a greatest common fingerprint mask and let  $k$  be the number of bits in  $M$  that are 0. Then  $\mathcal{S}(M, m)$  contains at most  $2^k$  distinguishable situations. There exist games in which this limit is reached.*

*Proof.* A situation fingerprint matches  $(M, m)$  if it has the same value as  $m$  for all bits that are 1 in  $M$ . Thus, all bits that are 1 in  $M$  are fixed, and all bits that are 0 in  $M$  are free. A bit can be 1 or 0, which leads to  $2^k$  different values for the  $k$  free bits, so up to  $2^k$  situation fingerprints that match  $(M, m)$ .

Let us look at a game with  $2^b$  situations, numbered from 0 to  $2^b - 1$ , and  $b$  questions of the form “Is the  $x$ -th bit of the situation 1?”. Thus, the fingerprint of a situation equals the numbers of that situation itself, and all possible fingerprint values exist. After having asked  $k$  questions,  $b - k$  of the bit positions are fixed, and we indeed have  $2^{b-k}$  distinguishable situations that are still possible.  $\square$

**Lemma 6.70.** *Let  $(M, m)$  be a greatest common fingerprint mask for which at least one bit in  $M$  is 0. Then  $\mathcal{S}(M, m)$  contains at least 2 distinguishable situations. There exist games in which this limit is reached.*

*Proof.* Let us assume that there existed only one situation  $s \in \mathcal{S}$  that matches  $(M, m)$ . Since a greatest common fingerprint mask can be generated from the list of remaining situations it describes,  $(M, m)$  can be generated from  $\{s\}$ . However, if we generate it from only one situation, then all bits in  $M$  must be 1, which contradicts the assumption. Therefore there must exist at least two situations.

For showing that two situations are enough, let us assume that there exists one situation  $s$  whose fingerprint equals  $m$  for all bits where  $M$  is 1, and is 0 for all bits where  $M$  is 0. Let us assume we have another situation  $s'$  whose fingerprint also equals  $m$  for all bits where  $M$  is 1, but is 1 for all bits where  $M$  is 0. Then the fingerprints of  $s$  and  $s'$  both match  $(M, m)$ , and if we generated the greatest common fingerprint mask of  $\{s, s'\}$  we would get  $(M, m)$  again. Therefore, these two situations are sufficient to generate  $(M, m)$ , no matter how many free bits  $(M, m)$  has. It is easy to generate a game in which  $\{s, s'\}$  occurs as set of remaining situations.  $\square$

In the most simple form, we can therefore manually define an upper limit at the beginning of the calculation that describes up to what size of the node (or up to what number of 1s in the bitmask of the greatest common fingerprint mask) we store known results.

Alternatively, we can set an upper limit for the number of known results that can be stored at the same time, and dynamically keep track of the number of the highest allowable size. If we get above the storage limit, we discard the results for the largest nodes first.

## 6.8 Distributed calculation

A little anecdote from the thesis author: One day a colleague at Google asked me about my diploma thesis, so I told him that I work on an NP-hard problem in game theory. I also told him that sadly I will not be able to calculate the results for the one game I am personally most interested in, because the total calculation time for it would be several decades. “So,” he asked, genuinely perplexed, “why don’t you just run it on a few thousand machines?”

Sadly, we do not have a few thousand machines at our disposal, but let us look at what we would do if we had.

Looking back at the first version of the algorithm that did not use any caching or other optimizations, we can see that we can easily calculate all children in parallel, since there is no interaction between them at all, and the order of their evaluation does not matter. For using the branch and bound optimization, we however have to evaluate the children one after the other. For using a cache of known results, we also have to keep this cache in a central location.

However, we can quickly see that the children of a computer node have disjoint sets of remaining questions. Consequently, their children will never again access the same node – a computer node effectively splits the tree below it into disjoint subtrees.



**Theorem 6.71.** *Let  $N$  be a computer node, and let  $C_1$  and  $C_2$  be two of its children. Then there exists no node  $T$  such that  $T$  can be reached both from  $C_1$  and from  $C_2$  by following forward edges.*

*Proof.* Let us assume that such a node  $T$  existed, and let  $\hat{S}(T)$  be the set of remaining situations in  $T$ . The remaining situations in each node are a subset of the remaining situations in its parent, and therefore recursively in each of its ancestors. Since  $C_1$  and  $C_2$  are both ancestors of  $T$ , we would get that  $\hat{S}(T) \subseteq \hat{S}(C_1)$  and  $\hat{S}(T) \subseteq \hat{S}(C_2)$ . This, however, is a contradiction to the properties of  $N$ , since the children of  $N$  have disjoint sets of remaining situations.  $\square$

We therefore can run the calculation for each child on a separate machine with a separate cache of known results – since they never reach the same player node, storing these results in different locations does not cause any loss of interesting information.

### 6.8.1 Splitting after the first few questions

If the root of the decision set graph was a computer node, we could therefore easily split the problem into around  $|\mathcal{A}|$  disjoint sub-problems. Unfortunately, the root is a player node, and player nodes do not have any such properties. Also, we might want to split the problem into more than  $|\mathcal{A}|$  parts.

We therefore split the game after the first few questions instead, at the cost of calculating some parts multiple times.

Let  $k$  denote the “layer” at which we want to split. Then we start at the root and find all player nodes that we can reach with exactly  $k$  questions. That is, we find all nodes for which there exists at least one path from the root with  $2k$  forward edges ( $k$  questions and  $k$  answers). Let  $\hat{N}$  be the set of such nodes. If we know the expected future costs of all of these nodes, then we can calculate the expected future costs of the root without touching any nodes below this layer.

As seen earlier in Theorem 5.13, for  $k = 1$  there can be at most  $qa$  such nodes, for  $k = 2$  at most  $q(q-1)a^2$ , for  $k = 3$  at most  $q(q-1)(q-2)a^3$ , and so on. Consequently, there can also be at most  $qa + q(q-1)a^2 + q(q-1)(q-2)a^3 + \dots + \frac{q!}{(q-k)!}a^k$  nodes on and above this layer.

In practice, for example, this might mean that we look at all the nodes in which two questions have been asked and answered already.

What we therefore do is this:

**Algorithm 6.72.**

- Choose some (rather small)  $k$ .
- Starting from the root and doing a simple depth-first search, find all player nodes at a distance of  $2k$  from the root. This has a runtime of at most  $O(kq^k a^k)$ . Let  $\hat{N}$  be the list of such nodes.
- For each node  $N \in \hat{N}$ , calculate the expected future costs of  $N$  on a separate machine, using all of the optimizations described above. More precisely, this means solving the subgame with  $N$  as root, which can be defined by the list of remaining situations in  $N$  and the list of questions that are still interesting there. For solving this subgame, we do not even need to know that it originally was part of a larger game.
- Calculate the expected future costs of the original game, using the values calculated for the nodes in  $\hat{N}$  whenever we reach one of those nodes. We therefore need to calculate a tree with  $O(kq^k a^k)$  nodes, which can easily be done on a single machine.

For example, we might want to choose  $k$  in such a way that we get around 1000 nodes in  $\hat{N}$ . Then we run these sub-calculations on 1000 machines, and combine them later again.

Since the sub-problems might have very different running times, we might alternatively want to split it into 10000 such sub-calculations, for example. Some of the 1000 CPUs might then calculate hundreds of those, while others will need equally long for one single (harder) sub-calculation.

### 6.8.2 Variations of this method

There are many other ways available to define a layer after which we split the calculation. Also, if we find more nodes in  $\hat{N}$  than we need, we can calculate some of those on the same machine and keep the cache between the calculations, rather than starting every time from scratch. If doing so, there probably are many ways to optimize which parts should run on the same machine.

Also, we can try to use a commonly used cache. Sending all requests to this cache is likely to create a performance bottleneck there, but sharing frequently used values could well pay off. Again, good caching strategies are a vast topic that would go far beyond the scope of this thesis.

## 6.9 Summary

A short summary of what we did in this chapter:

After defining the input in Section 6.1 and looking at the most trivial possible implementation of the recursive calculation in Section 6.2, we optimized this recursive calculation step by step by first making the decision set graph implicit in Section 6.3, then simplifying the used weights in Section 6.4, and finally introducing upper bounds that allow us to abort unnecessary calculations in Section 6.5.

After that, the recursive calculation was fairly streamlined, with the last version of it shown in Algorithms 6.25 (for player nodes), Algorithm 6.30 (for average-optimal computer nodes) and Algorithm 6.32 (for worst-case-optimal computer nodes).

What still had been left vague in those algorithms at this point was the exact implementation of:

1. the internal representation of nodes, in particular concerning how they store their lists of remaining situations;
2. the functions `calculateComputerNodeChildren(Node N)` and `calculatePlayerNodeChildren(Node N)` for calculating the children of a node; and
3. the data structure `KnownResults`.

In Section 6.6 we introduced fingerprints, which in very much detail described the implementation of the first two of these points. In Section 6.7.1 finally we looked at possible implementations and caching strategies for the third, thereby finishing the description of all details needed to implement the algorithm.

Section 6.8 adds a short discussion of possible ways to distribute the calculation onto several machines.

## Chapter 7

# Efficiently finding best estimates for particular games

As mentioned, finding efficient functions  $\mathbf{p}'$  and  $\mathbf{p}''$  is crucial for fast calculations, and depends on the problem definition. We will therefore now look at our games and try to find good functions for determining the best estimate for each of them.

One general observation before we start:

We defined  $\mathbf{p}'$  as the best possible expected penalty multiplied with the sum of weights. If  $\hat{\mathcal{S}}$  is the set of remaining situations, then the expected penalty for an estimate  $e \in \mathcal{E}$  is

$$\tilde{\mathbf{p}}'(\hat{\mathcal{S}}, e) = \frac{\sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot \mathbf{p}(s, e)}{\sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s)} .$$

Therefore we can define

$$\mathbf{p}'(\hat{\mathcal{S}}) = \left( \min_{e \in \mathcal{E}} \tilde{\mathbf{p}}'(\hat{\mathcal{S}}, e) \right) \cdot \left( \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \right)$$

according to Definition 6.1.

Since

$$\begin{aligned} \mathbf{p}'(\hat{\mathcal{S}}) &= \left( \min_{e \in \mathcal{E}} \tilde{\mathbf{p}}'(\hat{\mathcal{S}}, e) \right) \cdot \left( \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \right) \\ &= \min_{e \in \mathcal{E}} \left( \tilde{\mathbf{p}}'(\hat{\mathcal{S}}, e) \cdot \left( \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \right) \right) \\ &= \min_{e \in \mathcal{E}} \left( \frac{\sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot \mathbf{p}(s, e)}{\sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s)} \cdot \left( \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \right) \right) \\ &= \min_{e \in \mathcal{E}} \left( \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot \mathbf{p}(s, e) \right) , \end{aligned}$$

we can equivalently say that  $\mathbf{p}'(\hat{\mathcal{S}})$  is the smallest possible sum of penalties for a fixed estimate  $\bar{e} \in \mathcal{E}$  and all situations in  $\hat{\mathcal{S}}$ .

For simplicity, we define

$$\bar{\mathbf{p}}'(\hat{\mathcal{S}}, e) = \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot \mathbf{p}(s, e)$$

as the sum of penalties for one estimate, and get

$$\mathbf{p}'(\hat{\mathcal{S}}) = \min_{e \in \mathcal{E}} \bar{\mathbf{p}}'(\hat{\mathcal{S}}, e) .$$

## 7.1 Situation has to be guessed exactly

We have several games in which guessing the exact value is free, but guessing wrongly has infinite cost:

- Number guessing
- Number guessing with bit questions
- MasterMind
- BattleShips
- Minesweeper
- Minesweeper with free empty fields
- Classification
- Classification with different question costs
- Life (first variant, second possible penalty definition)
- Production parameters (first variant)

Fortunately, finding best estimates for these is very simple: if there are still two or more situations possible, we simply should not guess.

Therefore,

$$p'(\hat{S}) = \begin{cases} 0 & |\hat{S}| = 1 \\ \infty & |\hat{S}| > 1 \end{cases} .$$

The same holds of course for the worst case, so  $p''(\hat{S}) = p'(\hat{S})$ .

The calculation effort is  $O(1)$  (compared to  $O(|\hat{S}| \cdot |\mathcal{E}|)$  when making the computer calculate this with generic means).

## 7.2 Part of situation has to be guessed exactly

In the following games, we have to guess some part of the situation  $S$  correctly, but do not need to know anything about the rest of the information in  $S$ :

- Life (second variant, second possible penalty definition)
- Production parameters (second variant)
- Scales

In this variant of the Life game, for example, we have to find the best decisions for all choices, but we do not need to know the importance of the choices. Likewise, for finding the best production parameters, we want to know how to set these parameters, but do not care how important they are. In the Scales puzzle, we need to determine which coin is fake, but not whether it is heavier or lighter.

Consequently, we can safely take a guess whenever all remaining situations have the same values *for these parameters*, and have infinite costs if there exist two remaining situations with different values for them.

For Life and Production parameters, which both use  $c_i(s)$  to denote the value of choice/parameter  $i$  in situation  $s$ , we therefore get

$$p'(\hat{S}) = \begin{cases} 0 & c_i(s) = c_i(s') \quad \forall s, s' \in \hat{S}, i \in I \\ \infty & \text{otherwise} \end{cases} ,$$

where  $I$  denotes the set of choices/parameters.

For Scales we similarly get

$$p'(\hat{S}) = \begin{cases} 0 & [\text{fake coin in } s] = [\text{fake coin in } s'] \quad \forall s, s' \in \hat{S} \\ \infty & \text{otherwise} \end{cases} .$$

In both cases, we get  $p''(\hat{S}) = p'(\hat{S})$ .

The calculation effort is  $O(|\hat{S}|)$  (compared to  $O(|\hat{S}| \cdot |\mathcal{E}|)$  when making the computer calculate this with generic means).

### 7.3 Number guessing with finite penalty

While the following would apply to any game with 0 penalty for a correct guess and constant penalty  $C$  for a wrong one (and  $\mathcal{S} = \mathcal{E}$ ), the only such game we looked at is number guessing with finite penalty. We would expect that choosing the most likely estimate is the best choice here, and this is exactly what happens, as shown below.

Looking at the sum of penalties, we get a fixed penalty of  $C$  for each situation we did not choose as an estimate, so for all situations except for at most one (since we can only choose one as our estimate). More mathematically, let us look at the sum of penalties for some estimate  $e \in \mathcal{E} = \mathcal{S}$ :

$$\begin{aligned} \bar{p}'(\hat{\mathcal{S}}, e) &= \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot \mathfrak{p}(s, e) \\ &= \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot \begin{pmatrix} 0 & s = e \\ C & s \neq e \end{pmatrix} \\ &= \left( \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot C \right) - \mathfrak{w}(e) \cdot C \\ &= C \cdot \left( \sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \right) - \mathfrak{w}(e) \cdot C \end{aligned}$$

The first term only depends on  $\hat{\mathcal{S}}$  and is not influenced at all by the estimate we choose. We can therefore only try to maximize what we subtract. The best strategy therefore is simple: Choose the number that is most likely.

However, in the number guessing game all situations have weight 1 anyway. We therefore can just choose any of them. No matter which one we choose, we get

$$\mathfrak{p}'(\hat{\mathcal{S}}) = C \cdot (|\hat{\mathcal{S}}| - 1) .$$

For the worst case, we either get  $C$  if at least two numbers are still possible – there always is the risk to chose wrongly –, or 0 if only one number remains:

$$\mathfrak{p}''(\hat{\mathcal{S}}) = \begin{cases} 0 & |\hat{\mathcal{S}}| = 1 \\ C & |\hat{\mathcal{S}}| > 1 \end{cases}$$

The calculation effort is  $O(1)$  again, assuming we can determine  $|\hat{\mathcal{S}}|$  in  $O(1)$  time.

### 7.4 Number guessing with estimates

In the classical number guessing game, we always have a range of consecutive integers that are still possible. Let us assume that  $a$  is the smallest and  $b$  the largest number that we have not ruled out (and all numbers between them are possible).

We have a cost function of the form  $C \cdot \text{distance}^D$ . Given any estimate  $x$ , the sum of penalties  $\bar{\mathbf{p}}'$  is the following:

$$\begin{aligned} \bar{\mathbf{p}}'(\hat{\mathcal{S}}, x) &= \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot C \cdot |x - s|^D \\ &= \sum_{s \in \hat{\mathcal{S}}} 1 \cdot C \cdot |x - s|^D \\ &= C \cdot \sum_{s \in \hat{\mathcal{S}}} |x - s|^D \\ &= C \cdot \sum_{y=a}^b |x - y|^D \end{aligned}$$

Simple mathematics show that  $x = \lfloor \frac{a+b}{2} \rfloor$ , the median of this range, is the best choice. If we calculate this sum directly, this takes  $O(|\hat{\mathcal{S}}|)$  time.

For specific values of  $D$ , the expected penalty can even be expressed in a closed form. For simplicity we define  $l := \lfloor \frac{a+b}{2} \rfloor - a = \lfloor \frac{b-a}{2} \rfloor$ , the distance between the first and the middle number in the range. For  $D = 1$  we then get the following simplified form:

$$\begin{aligned} \mathbf{p}'(\hat{\mathcal{S}}) &= C \cdot \sum_{y=a}^b \left| \left\lfloor \frac{a+b}{2} \right\rfloor - y \right| \\ &= C \cdot \left( l + (l-1) + (l-2) + \dots + 2 + 1 + 0 + 1 + 2 + \dots + (l-1) + l + \left( \begin{matrix} 0 & b-a \text{ even} \\ l+1 & b-a \text{ odd} \end{matrix} \right) \right) \\ &= C \cdot 2 \cdot \frac{l(l+1)}{2} + C \cdot \left( \begin{matrix} 0 & b-a \text{ even} \\ l+1 & b-a \text{ odd} \end{matrix} \right) \\ &= C \cdot l(l+1) + C \cdot \left( \begin{matrix} 0 & b-a \text{ even} \\ l+1 & b-a \text{ odd} \end{matrix} \right) \end{aligned}$$

Using this, we have a calculation effort of  $O(1)$  (assuming we can find  $a$  and  $b$  in  $O(1)$  time).

For the worst-case-optimal estimate, we trivially again choose the value in the middle of the range. The worst possible penalty is the one for a value at the end of the range. We therefore get

$$\mathbf{p}''(\hat{\mathcal{S}}) = \begin{cases} C \cdot l^D & (b-a) \text{ even} \\ C \cdot (l+1)^D & (b-a) \text{ odd} \end{cases} .$$

## 7.5 Number guessing with estimates and bit questions

In this game, we might end up with ranges that are not continuous. For example, if we know that the number is larger than 5, smaller than 23, and has 1 as the third bit from the right, then the set of remaining situations is  $\hat{\mathcal{S}} = \{6, 7, 12, 13, 14, 15, 20, 21, 22\}$ .

For  $D = 1$  we have to minimize a simple sum of distances. We can use a neat little trick for this: In the example above we can see that for any estimate  $6 \leq e \leq 22$  we can choose, we always get  $|6 - e| + |22 - e| = e - 6 + 22 - e = 16$  as sum of the penalties for situations 6 and 22; if we choose  $e$  outside this range, it will be higher than that. Likewise, the sum of penalties for situations 7 and 21 is highest if  $e$  is outside the interval  $[7, 21]$ , and constantly at a minimum of 14 if it is within. Looking at the sums of penalties for situations 12 and 22, this sum is smallest if  $e$  is in the interval  $[12, 22]$ , and so on. Therefore, the sum of the penalties of all situations is minimal if we choose the situation in the middle of this list as estimate.

That is, for  $D = 1$  let  $\hat{\mathcal{S}} = \{s_1, s_2, \dots, s_k\}$  be the list of situations ordered by number, then the sum of penalties is minimal if we choose  $s_{\lfloor \frac{k}{2} \rfloor}$  as estimate. We still have to calculate this sum, so we get a runtime of  $O(|\hat{\mathcal{S}}|)$ .

For  $D = 2$  or higher, however, finding such tricks can be very hard. Therefore, we might not have a better choice here than calculating the sum of penalties for each estimate and choosing the estimate for which it is lowest, for a runtime of  $O(|\hat{\mathcal{S}}| \cdot |\mathcal{E}|)$ . This shows that it is not necessarily possible for every game to find something more efficient.

## 7.6 BlackBox

In BlackBox we have to guess where the  $k$  dots on the  $n \times n$  grid are by marking any  $k$  dots. Using some mathematics we will show that in BlackBox we get the best average penalty when we choose those  $k$  fields that contain dots in most of the remaining situations – even if this combination of  $k$  dots itself is not among the remaining situations any more.

Let  $d(s, x, y) \in \{1, 0\}$  denote whether in situation  $s$  there is a dot on field  $(x, y)$  (where 1 means that there is a dot, and 0 means the field is empty). Even though mathematically a little sloppy, we will write  $(x, y) \in [n \times n]$  to denote that  $(x, y)$  is a field on the  $n \times n$  grid.

We know that  $\mathbf{p}'$  is the minimal sum of the penalties. For the sum of penalties  $\bar{\mathbf{p}}'$  for one estimate  $e \in \mathcal{E}$  we can transform the sum:

$$\begin{aligned} \bar{\mathbf{p}}'(\hat{\mathcal{S}}, e) &= \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot \mathbf{p}(s, e) \\ &= \sum_{s \in \hat{\mathcal{S}}} 1 \cdot 5 \cdot \#[\text{dots marked in } s \text{ that are not marked in } e] \\ &= 5 \sum_{s \in \hat{\mathcal{S}}} \#[\text{dots marked in } s \text{ that are not marked in } e] \\ &= 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x, y) \in [n \times n]} \left( \begin{cases} 1 & d(s, x, y) = 1 \wedge d(e, x, y) = 0 \\ 0 & \text{otherwise} \end{cases} \right) \end{aligned}$$

Let us first simplify the summands:

$$\begin{aligned} &\left( \begin{cases} 1 & d(s, x, y) = 1 \wedge d(e, x, y) = 0 \\ 0 & \text{otherwise} \end{cases} \right) \\ &= \left( \begin{cases} 1 & d(s, x, y) = 1 \wedge d(e, x, y) = 0 \\ 0 & d(s, x, y) = 1 \wedge d(e, x, y) = 1 \\ 0 & d(s, x, y) = 0 \end{cases} \right) \\ &= \left( \begin{cases} d(s, x, y) & d(s, x, y) = 1 \wedge d(e, x, y) = 0 \\ d(s, x, y) - 1 & d(s, x, y) = 1 \wedge d(e, x, y) = 1 \\ d(s, x, y) & d(s, x, y) = 0 \end{cases} \right) \\ &= d(s, x, y) - \left( \begin{cases} 0 & d(s, x, y) = 1 \wedge d(e, x, y) = 0 \\ 1 & d(s, x, y) = 1 \wedge d(e, x, y) = 1 \\ 0 & d(s, x, y) = 0 \end{cases} \right) \\ &= d(s, x, y) - d(s, x, y)d(e, x, y) \end{aligned}$$

We use this for the calculation of  $\bar{\mathbf{p}}'$ :

$$\begin{aligned} \bar{\mathbf{p}}'(\hat{\mathcal{S}}, e) &= 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x, y) \in [n \times n]} \left( \begin{cases} d(s, x, y) & d(s, x, y) = 1 \wedge d(e, x, y) = 0 \\ 0 & \text{otherwise} \end{cases} \right) \\ &= 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x, y) \in [n \times n]} (d(s, x, y) - d(s, x, y)d(e, x, y)) \\ &= 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x, y) \in [n \times n]} d(s, x, y) - 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x, y) \in [n \times n]} d(s, x, y)d(e, x, y) \end{aligned}$$

By definition, in each situation  $s$  there are exactly  $k$  fields that contain a dot, which means that exactly  $k$

of the  $d(s, x, y)$  are 1 and the others are all 0. This leads to the following:

$$\begin{aligned} \bar{p}'(\hat{\mathcal{S}}, e) &= 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x,y) \in [n \times n]} d(s, x, y) - 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x,y) \in [n \times n]} d(s, x, y) d(e, x, y) \\ &= 5 \sum_{s \in \hat{\mathcal{S}}} k - 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x,y) \in [n \times n]} d(s, x, y) d(e, x, y) \\ &= 5k \cdot |\hat{\mathcal{S}}| - 5 \sum_{s \in \hat{\mathcal{S}}} \sum_{(x,y) \in [n \times n]} d(s, x, y) d(e, x, y) \end{aligned}$$

Now we simplify the term on the right by changing the order of summation:

$$\begin{aligned} \bar{p}'(\hat{\mathcal{S}}, e) &= 5k \cdot |\hat{\mathcal{S}}| - 5 \sum_{(x,y) \in [n \times n]} \sum_{s \in \hat{\mathcal{S}}} d(s, x, y) d(e, x, y) \\ &= 5k \cdot |\hat{\mathcal{S}}| - 5 \sum_{(x,y) \in [n \times n]} d(e, x, y) \cdot \left( \sum_{s \in \hat{\mathcal{S}}} d(s, x, y) \right) \\ &= 5k \cdot |\hat{\mathcal{S}}| - 5 \sum_{(x,y) \in [n \times n]} d(e, x, y) \cdot D(\hat{\mathcal{S}}, x, y) \end{aligned}$$

where  $D(\hat{\mathcal{S}}, x, y) = \sum_{s \in \hat{\mathcal{S}}} d(s, x, y)$  denotes the number of situations in  $\hat{\mathcal{S}}$  which have a dot on field  $(x, y)$ .

Again, there can only be  $k$  fields  $(x, y)$  for which  $d(e, x, y) = 1$ . Let  $\{(x_1, y_1), (x_2, y_2), \dots, (x_k, y_k)\}$  be the set of fields for which there is a dot in  $e$ , then we get

$$\begin{aligned} \bar{p}'(\hat{\mathcal{S}}, e) &= 5k \cdot |\hat{\mathcal{S}}| - 5 \sum_{(x,y) \in [n \times n]} d(e, x, y) \cdot D(\hat{\mathcal{S}}, x, y) \\ &= 5k \cdot |\hat{\mathcal{S}}| - 5 \sum_{i=0}^k D(\hat{\mathcal{S}}, x_i, y_i) \end{aligned}$$

The left term and the values of  $D(\hat{\mathcal{S}}, x, y)$  cannot be influenced by the estimate, therefore we can consider them to be constants. This means that we have a fixed sum from which we can subtract  $k$  constants. To minimize this, we of course subtract the  $k$  highest of the available constants  $D(\hat{\mathcal{S}}, x, y)$ .

(Less mathematically, we can think of it like this: We overlay all remaining situations, that is, we allow multiple dots on one field, and for each remaining situation we add one dot to each field where this situation has a dot. Then we are allowed to choose  $k$  fields that are free. For each dot that is not on one of these free fields, we have to pay 5.)

We can calculate this in  $O(\hat{\mathcal{S}})$  time: We start with an empty grid. For each situation, we add its  $k$  dots to the grid. Then we find the  $k$  fields with most dots, and calculate the sum of the remaining  $n^2 - k$  fields. Since  $k$  and  $n$  are constants, this takes  $O(\hat{\mathcal{S}})$  time altogether.

(Technically,  $|\hat{\mathcal{S}}|$  is of course related to  $k$  and  $n$ , but since  $|\mathcal{S}| = \binom{n^2}{k}$ , we can consider  $k$  and  $n$  to be so small that they are almost constant in comparison.)

## 7.7 Life (both variants, first way to define costs)

In Life, fortunately all choices are independent of each other. After some experiments, for some decisions we might know for sure what the better choice is. For others, we might not have a definite answer, but might have indicators that one answer is more likely to be correct than the other. Instinctively we would for each choice take the decision that is more likely, evaluating the choices independently of each other. Now we just need some mathematics to prove this to be correct.



We know that  $\mathbf{p}'$  is the minimal sum of the penalties. For the sum of penalties  $\bar{\mathbf{p}}'$  for one estimate  $e \in \mathcal{E}$  we can transform the sum:

$$\begin{aligned}
 \bar{\mathbf{p}}'(\hat{\mathcal{S}}, e) &= \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot \mathbf{p}(s, e) \\
 &= \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot 50\mathbf{c}(s, e) \\
 &= \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s) \cdot 50 \sum_{i=1}^n \left( \begin{cases} b_i(s) & c_i(s) \neq c_i(e) \\ 0 & c_i(s) = c_i(e) \end{cases} \right) \\
 &= 50 \sum_{s \in \hat{\mathcal{S}}} \sum_{i=1}^n \mathbf{w}(s) \left( \begin{cases} b_i(s) & c_i(s) \neq c_i(e) \\ 0 & c_i(s) = c_i(e) \end{cases} \right) \\
 &= 50 \sum_{s \in \hat{\mathcal{S}}} \sum_{i=1}^n \left( \begin{cases} \mathbf{w}(s)b_i(s) & c_i(s) \neq c_i(e) \\ 0 & c_i(s) = c_i(e) \end{cases} \right) \\
 &= 50 \sum_{i=1}^n \sum_{s \in \hat{\mathcal{S}}} \left( \begin{cases} \mathbf{w}(s)b_i(s) & c_i(s) \neq c_i(e) \\ 0 & c_i(s) = c_i(e) \end{cases} \right) \\
 &= 50 \sum_{i=1}^n \sum_{\substack{s \in \hat{\mathcal{S}} \\ c_i(s) \neq c_i(e)}} \mathbf{w}(s)b_i(s) \\
 &= 50 \sum_{i=1}^n \left( \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)b_i(s) - \sum_{\substack{s \in \hat{\mathcal{S}} \\ c_i(s) = c_i(e)}} \mathbf{w}(s)b_i(s) \right) \\
 &= 50 \sum_{i=1}^n \left( \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)b_i(s) \right) - 50 \sum_{i=1}^n \left( \sum_{\substack{s \in \hat{\mathcal{S}} \\ c_i(s) = c_i(e)}} \mathbf{w}(s)b_i(s) \right)
 \end{aligned}$$

Again, we have no influence at all on the first term. We want the second term that we subtract to be as large as possible.

That sum is maximal if each summand is maximal. Fortunately, each summand only looks at the values  $b_i$  and  $c_i$  for one  $i$ . For each  $i$ , let  $c'_i$  be the value for which

$$\sum_{\substack{s \in \hat{\mathcal{S}} \\ c_i(s) = c'_i}} \mathbf{w}(s)b_i(s)$$

is maximal. Then we define an estimate  $\bar{e}$  by setting  $c_i(\bar{e}) = c'_i$  for each  $i$ .

An alternative way to see this: We split the set of situations  $\hat{\mathcal{S}}$  into disjoint subsets  $\hat{\mathcal{S}}(x) = \{s \in \hat{\mathcal{S}} \mid c_i(s) = x\}$  depending on the correct decision for choice  $i$  (where  $x \in \{\text{"true"}, \text{"false"}\}$  in our definition). Then we set  $c_i(\bar{e})$  to the decision that is correct (and important) most often.

Since all ways to set the  $c_i$  are valid estimates, we get  $\bar{e} \in \mathcal{E}$ . Therefore,  $\bar{e}$  is the best estimate. The sum of penalties can be calculated as  $\bar{\mathbf{p}}'(\hat{\mathcal{S}}, \bar{e})$ . Without going into too much detail, both the calculation of  $\bar{e}$  and of  $\bar{\mathbf{p}}'(\hat{\mathcal{S}})$  can be done in  $O(\hat{\mathcal{S}})$  time.

The worst case is trickier than the average case here: Let us assume that we have only two situations left, we have three decisions, and each decision can be expressed as 1 or 0. In one situations the correct choices are 001, while in the other we should choose 110. Then for each choice, we could decide wrongly, which would give us a worst case of 1 per choice, so a total penalty of 3. However, if we choose 111, then the worst penalty we can get is 2.

As we see in this example, minimizing the choices independently is therefore not sufficient. For all choices in which in all remaining situations the same decision is correct, we can trivially set the estimate

for that choice to this decision as well. Other than that, there does not seem to exist a very obvious way to choose a best estimate, other than looking at all estimates. We therefore have a runtime of  $O(|\hat{S}| \cdot |\mathcal{E}|)$  for calculating  $\mathbf{p}''$ .

## 7.8 Milk

Again, the Milk “game” is small enough that we can manually calculate best estimates and define them in a table (assuming  $\mathbf{w} = 1$  for all situations):

remaining situations	best estimate on average	$\mathbf{p}'$
okay	use	0
slightly sour	throw away	0
very sour	throw away	0
mouldy sour	throw away	0
okay, slightly sour	throw away	$20 + 0 = 20$
very sour, mouldy	throw away	0
okay, slightly sour, very sour	throw away	$20 + 0 + 0 = 20$
okay, slightly sour, very sour, mouldy	throw away	$20 + 0 + 0 + 0 = 20$

The worst-case-optimal estimates look almost the same:

remaining situations	best estimate in worst case	$\mathbf{p}''$
okay	use	0
slightly sour	throw away	0
very sour	throw away	0
mouldy sour	throw away	0
okay, slightly sour	throw away	$\max(20, 0) = 20$
very sour, mouldy	throw away	0
okay, slightly sour, very sour	throw away	$\max(20, 0, 0) = 20$
okay, slightly sour, very sour, mouldy	throw away	$\max(20, 0, 0, 0) = 20$

For other weights  $\mathbf{w}$ , we might get a different table. Let us look at the following weights where we are almost certain that the milk is still good:

$$\begin{aligned} \mathbf{w}(\text{“okay”}) &= 120 \\ \mathbf{w}(\text{“slightly sour”}) &= 3 \\ \mathbf{w}(\text{“very sour”}) &= 2 \\ \mathbf{w}(\text{“mouldy”}) &= 1 \end{aligned}$$

For those, we get the following table:

remaining situations	best estimate on average	$\mathbf{p}'$
okay	use	0
slightly sour	throw away	0
very sour	throw away	0
mouldy sour	throw away	0
okay, slightly sour	use	$120 \cdot 0 + 3 \cdot 100 = 300$
very sour, mouldy	throw away	0
okay, slightly sour, very sour	use	$120 \cdot 0 + 3 \cdot 100 + 2 \cdot 500 = 1300$
okay, slightly sour, very sour, mouldy	throw away	$120 \cdot 20 + 3 \cdot 0 + 2 \cdot 0 + 1 \cdot 0 = 2400$

The worst cases are not influenced by weights, so they look the same as before.

## Chapter 8

# Not optimal strategies

All of our algorithms so far find the best possible strategy, but need a lot of calculation effort for doing so. For performance comparison, we could look on the one hand at three simple heuristic algorithms that are similar to those used in decision tree learning, and on the other hand try to find out how well the player would do if he simply played entirely randomly.

### 8.1 Heuristic methods

For each question  $q$  and set of remaining situations  $\hat{\mathcal{S}}$ , we can have a look at the sizes of the children that the corresponding computer node has. For example, if we play number guessing and have numbers from 1 to 100, then for the question “Is the number larger than 15?” we have two possible outcomes; there are 15 situations for which the answer is “no”, and 85 situations for which the answer is “yes”.

For the following section, let  $a_q = |\mathcal{A}_q|$  be the number of answers, let  $\mathcal{A}_q = \{a_1^{(q)}, a_2^{(q)}, \dots, a_{a_q}^{(q)}\}$  be the set of these answers, and let

$$b_i^{(q)} = |\{s \in \hat{\mathcal{S}} \mid \tau(s, q) = a_i^{(q)}\}|$$

be the number of situations for each answer  $i$ . Let furthermore

$$\bar{b}_i^{(q)} = \sum_{\substack{s \in \hat{\mathcal{S}} \\ \tau(s, q) = a_i^{(q)}}} \mathfrak{w}(s)$$

be the sum of weights of the situations for each answer  $i$ . (If all weights are 1, then  $b_i^{(q)} = \bar{b}_i^{(q)}$ .)

All heuristic methods we look at give a simple way to decide which question to ask next by only looking at the distribution of the answers in the available questions. That is, rather than calculating the entire subgame for each question, these strategies look only one question ahead, and then decide on one of the available questions.

This has several advantages: On the one hand, it is easy to describe the strategy without having to write a long list what to do in each situation. In contrast, strategies found with our algorithms can often only be expressed by explicitly saving the next question to be asked in each player node that can be reached in an optimal strategy.

On the other hand, also calculating their average performance is faster than calculating the average performance of a perfect strategy, because we only need to evaluate one child for each player node. (That is, only for one child all those child’s successors are calculated recursively. The other children are only shortly peeked at to decide which one we take.)

The drawbacks are that obviously this does not give an optimal strategy. Also, many of our optimizations, the upper bounds for example, do not work any more.

The basic algorithm that we will use in all three heuristic methods looks roughly like this:

```
Algorithm 8.1. calculateCostsPlayerNode(Node N) :=
  children = calculatePlayerNodeChildren(N);
  chosenChild = chooseBestChild(children);
  return getCosts(chosenChild);
```

The differences between the algorithms are in the definition of `chooseBestChild(...)`.

Since we will have to fit estimates into this somehow, we will at the end of the calculation simply check whether an estimate would have been better:

```
Algorithm 8.2. calculateCostsPlayerNode(Node N) :=
  children = calculatePlayerNodeChildren(N);
  chosenChild = chooseBestChild(children);
  costsOfChosenChild = getCosts(chosenChild);
  costsOfestimate = getBestEstimateCosts(N.getSituations());
  return min(costsOfChosenChild, costsOfestimate);
```

Note that these heuristics are mostly used for decision tree learning, where usually all questions are equally expensive, and certainly all answers to one question are equally expensive. Consequently, they do not handle all of our special qualities very gracefully.

There are in particular some games where questions can have infinite costs. What we can do is checking at the same time whether we are looking at such a question, and just manually forbid choosing questions like that.

### 8.1.1 Greedy

#### Choose question with smallest largest child

A very simple strategy is to always choose the question for which the largest child is minimal. That is, for each question  $q$  let  $l(q) = \max_{i \in \{1, 2, \dots, a_q\}} b_i^{(q)}$ . Choose the question for which  $l(q)$  is minimal.

#### Choose question with lowest highest weight of a child

Alternatively, we can also use the weighted child sizes: we define  $\bar{l}(q) = \max_{i \in \{1, 2, \dots, a_q\}} \bar{b}_i^{(q)}$  and choose the question for which  $\bar{l}(q)$  is minimal.

### 8.1.2 Information Gain

Like in the previous chapters, let  $\bar{\mathbf{w}}(\hat{\mathcal{S}}) = \sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)$ , and let  $\hat{\mathcal{S}}((q, a)) = \{s \in \hat{\mathcal{S}} \mid \mathbf{r}(s, q) = a\}$ .

We follow the definitions of the information gain from [46] (which is based on [45], but more readable).

For defining the entropy  $H(\hat{\mathcal{S}})$ , we need the probability  $\hat{\mathbf{w}}(s)$  for each situation  $s \in \hat{\mathcal{S}}$ . This is calculated as

$$\hat{\mathbf{w}}(s) = \frac{\mathbf{w}(s)}{\sum_{s \in \hat{\mathcal{S}}} \mathbf{w}(s)} = \frac{\mathbf{w}(s)}{\bar{\mathbf{w}}(\hat{\mathcal{S}})}.$$

Using this, we can define the entropy

$$H(\hat{\mathcal{S}}) = - \sum_{s \in \hat{\mathcal{S}}} \hat{\mathbf{w}}(s) \cdot \text{ld}(\hat{\mathbf{w}}(s)),$$

where  $\text{ld}$  stands for the logarithm of base 2.

The information gain is then defined as

$$\text{IG}(\hat{\mathcal{S}}, q) = H(\hat{\mathcal{S}}) - \sum_{a \in \mathcal{A}_q} \frac{|\hat{\mathcal{S}}((q, a))|}{|\hat{\mathcal{S}}|} \cdot H(\hat{\mathcal{S}}((q, a))).$$

### Choose highest Information Gain

The obvious strategy is to choose the question with the highest information gain.

### Choose highest Information Gain per cost

For decision trees, Information Gain Ratio is often used instead of Information Gain. This is however only to avoid overfitting, which would make generalization harder. As an example, let us assume we have a customer data base of an online shop and want a decision tree that tells us based on that data whether a user has bought a computer before. Then we might, for example, ask what the profession of this user is, what country he lives in, et cetera. That way, we might after some questions be able to take a good guess whether the user has bought a computer before or not. However, asking for the customer number as first question instead immediately gives us the desired information with only one question. The problem is that the former approach can easily be generalized, and is likely to give good estimates for new customers we might encounter, whereas the latter is useless for this purpose. Therefore, Information Gain Ratio can be used to favour questions with few different answers.

However, this is the exact opposite of what we want here. If there is one question whose answer can immediately identify the situation, we really *want* to use this question, not artificially downgrade it.

Therefore, even though we will also use fractions in the next paragraph, this *has nothing to do with Information Gain Ratio*.

What the problem with the question for the customer number could be for our games is that such a question will usually be very expensive. (If MasterMind had a “Tell me all used colours” question, it would most probably not be cheap, otherwise the game would become boring.) Our current approach does not factor in question costs at all.

For simplicity, we simply define the average question cost for a question  $q$  as

$$C_q(\hat{\mathcal{S}}) = \frac{\sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s) \cdot c(s, q)}{\overline{\mathfrak{w}}(\hat{\mathcal{S}})},$$

and the Information Gain per cost as

$$\text{IGc}(\hat{\mathcal{S}}, q) = \frac{\text{IG}(\hat{\mathcal{S}}, q)}{C_q(\hat{\mathcal{S}})}.$$

Then we choose the question with the highest Information Gain per cost.

### 8.1.3 Gini coefficient

As in the previous section, let

$$\hat{\mathfrak{w}}(s) = \frac{\mathfrak{w}(s)}{\sum_{s \in \hat{\mathcal{S}}} \mathfrak{w}(s)} = \frac{\mathfrak{w}(s)}{\overline{\mathfrak{w}}(\hat{\mathcal{S}})}.$$

Using the definitions from [56], we define the Gini coefficient as

$$\text{GC}(\hat{\mathcal{S}}, q) = 1 - \sum_{s \in \hat{\mathcal{S}}} (\hat{\mathfrak{w}}(s))^2,$$

and similar to information gain we get a “Gini Gain” defined as

$$\text{GG}(\hat{\mathcal{S}}, q) = \text{GC}(\hat{\mathcal{S}}) - \sum_{a \in \mathcal{A}_q} \frac{|\hat{\mathcal{S}}((q, a))|}{|\hat{\mathcal{S}}|} \cdot \text{GC}(\hat{\mathcal{S}}((q, a))).$$

**Choose highest Gini Gain**

We can again decide on the question with the highest Gini gain.

**Choose highest Gini Gain per cost**

Again, we can divide this by the expected cost and get

$$C_q(\hat{\mathcal{S}}) = \frac{\sum_{s \in \mathcal{S}} \mathbf{w}(s) \cdot c(s, q)}{\bar{\mathbf{w}}(\hat{\mathcal{S}})}$$

and

$$\text{GGC}(\hat{\mathcal{S}}, q) = \frac{\text{GG}(\hat{\mathcal{S}}, q)}{C_q(\hat{\mathcal{S}})}$$

as the Gini Gain per cost.

We then choose the question with the highest Gini gain per cost.

## 8.2 Random

Besides using heuristics, it is also interesting to see how well the player would do if he played entirely randomly. We restrict “entirely” a little, though, to prevent the user from running into endless loops and asking pointless questions.

That is, we define a random strategy as follows: “Of all the questions that can still give at least two different answers, choose one at random, using a uniform random distribution.”

There are three ways to handle estimates:

- Choose estimates with the same probability as each question. (That is, if there are 4 questions, choose each question with a probability of  $\frac{1}{5}$ , and choose the estimate with probability  $\frac{1}{5}$ ).
- Choose estimates with some fixed probability. (For example, choose the estimate with a probability of 50%, otherwise choose one of the questions with equal probability for each question.)
- Choose to take an estimate if and only if the expected cost of taking an estimate is lower than the expected cost of asking a random question. (Note that in order to do so, we first need to know – through calculation or magical intuition – the expected costs when asking a random question.)

One problem we face with calculating these is that in order to calculate the expected average, we now need to evaluate all children of all player nodes. Also, the upper bounds we used to reduce calculation effort do not work here. Thus, the calculation effort for finding out how well a random strategy performs is actually higher than the effort for calculating an optimal strategy.

# Chapter 9

## Implementation details

A few short words on the implementation that accompanies this thesis:

### 9.1 `BigIntegerInfinity` and `BigIntegerInfinityFraction`

For unfathomable reasons, Java does not provide a way to work with `BigInteger`s that can be infinite or NaN (unlike the `Double` class, which is able to handle them). We therefore provide our own implementation, called `BigIntegerInfinity`. (There are various implementations around already that do roughly the same, but copyright seemed a bit tricky for all of them.)

For fractions we introduce `BigIntegerInfinityFraction`, which simply consist of a pair of `BigIntegerInfinity`s, one representing the numerator and one the denominator. Looking at the code, all non-integer values we come across in any of the calculations can be expressed as fractions. Therefore, exact calculations are possible.

### 9.2 Input

Now for a really practical definition of the input, which is of course directly related to the one in Definition 6.1.

First of all, we have to define some computer-readable way to represent our situations, questions, answers, evaluation functions, and so on.

We basically provide the input as a set of Java classes: For each game, there has to be one class that describes a situation, one class that describes a question, and so on. The answer function  $\tau$ , cost function  $c$  and estimate penalty function  $p$  are all given as Java functions that are implemented separately for each game. In addition, we need functions that provide lists of all situations, of all questions, et cetera.

As we can see, defining a game in this way requires implementing it directly in Java, as opposed to just reading it in as a list of values. Thus, running the calculations on a new game actually requires modifying the program, which might be considered inelegant. However, defining  $p'$  and  $p''$  as lists is not necessarily possible in less than  $O(2^s)$  space, and therefore usually not desirable.

Without much further ado, let us have a look at the these interfaces:

- `Situation` contains one situation, and provides the following functions:
  - `BigInteger getWeight()`, the weight ( $w$ ) of the situation; and
  - `String toTinyString()`, a very short unique representation of the situation (that may be used as label for retrieving known results).
- `Question<S extends Situation>` contains one question, providing the following functions:
  - `BigInteger getAnswer(S situation)`, the answer ( $\tau$ ) to one question;
  - `BigIntegerInfinity getAnswerCost(S situation)`, the cost ( $c$ ) of that answer;
  - `BigIntegerInfinity getAnswerCost(BigInteger answer)`, the cost ( $c$ ) of the given answer; and
  - `Iterator<BigInteger> getAvailableAnswers()`, a list ( $\mathcal{A}_q$ ) of available answers.

- `Estimate<S extends Situation>` represents an estimate and provides only one function:
  - `BigIntegerInfinity getPenalty(S situation)`, the penalty ( $p$ ) for the estimate in the given situation.
- `Estimator<S extends Situation>` returns best estimates as described in Chapter 7, providing the following functions:
  - `BigIntegerInfinity getBestAverageEstimateCosts(Iterator<S> remainingSituations)`, the function  $p'$ ; and
  - `BigIntegerInfinity getBestWorstCaseEstimateCosts(Iterator<S> remainingSituations)`, the function  $p''$ , as defined in Definition 6.1.
- `Game<S extends Situation, Q extends Question<S>, E extends Estimate<S>>` finally represents the game, providing the following:
  - `Iterator<S> getSituations()`, the list of situations;
  - `Iterator<Q> getQuestions()`, the list of questions;
  - `Iterator<E> getEstimates()`, the list of estimates; and
  - `Estimator<S> getEstimator()`, the estimator for the game.

Note that we use iterators rather than lists so that their contents only have to be created on demand instead of having to keep the entire lists in memory.

This is the set of classes that have to be provided per game. To allow strict typing without having to use downcasts anywhere, all classes are parameterized with all other classes used for the same game (which, as a downside, turns it into quite a Java generics fest at the brink of unreadability).

## 9.3 Analysis

The whole implementation is written in such a way that it is easy to swap in and out different options in order to compare their performance. Of course, this does create some overhead, and if one wanted to implement a really efficient algorithm, it would be much better to decide on one algorithm and implement only that one (which takes at most a few hundred lines).

The options are:

- Analyzation algorithms (described in Sections 6.2 through 6.5, as well as Chapter 8):
  - `StandardWeightAnalyzer`, the basic algorithm from Section 6.3;
  - `SimplifiedWeightAnalyzer`, the algorithm with simplified weights from Section 6.4;
  - `SimplifiedWeightAnalyzerWithBounds`, the algorithm using upper bounds from Section 6.5;
  - `SimplifiedWeightAnalyzerWithBoundsAndRandomization`, the algorithm using upper bounds from Section 6.5, and additionally randomizing the order in which questions are evaluated;
  - `HeuristicAnalyzer`, calculating the expected performance when using one of the heuristics from Section 8.1 for choosing moves; and
  - `RandomAnalyzer`, calculating the expected performance when playing entirely randomly as described in Section 8.2.
- Fingerprinting:
  - `ListNode`, storing the set of remaining situations as lists;
  - `FingerprintNode`, describing the set of remaining situations with greatest common fingerprint masks; and
  - `FingerprintAndListNode`, describing the set of remaining situations with greatest common fingerprint masks and keeping lists in addition.



- Caching of known results:
  - `NoResultsCache`, turning off caching and thereby downgrading the algorithm to Algorithm 4.36 that worked on the behavioural game tree without making use of the properties of decision set graphs at all;
  - `SaveAllResultsCache`, saving all results using strings as labels; and
  - `SaveAllResultsFingerprintedCache`, saving all results using fingerprint masks as labels (if available).
- Caching of known limits:
  - `NoLimitsCache`, turning off caching of known upper limits;
  - `SaveAllLimitsCache`, saving all known upper limits using strings as labels; and
  - `SaveAllLimitsFingerprintedCache`, saving all known upper limits using fingerprint masks as labels (if available).

As we can see, this gives a fairly large number of options that we can compare.

For the sake of keeping the thesis short, all further documentation relevant to this implementation can be found in the accompanying JavaDocs.

# Chapter 10

## Results

### 10.1 Upper bounds

First, let us have a look how our multiple upper bounds look like in some games, shown in Tables 10.1 and 10.2. The lowest upper bound for each game is highlighted.

	NG (10)	NG (100)	MM (4 × 4)	MM (4 × 6)
<b>s</b>	10	100	256	1 296
<b>q</b>	9	99	256	1 296
<b>a</b>	2	2	14	14
$2^s - 1$	<b><math>\approx 1.02 \cdot 10^3</math></b>	<b><math>\approx 1.26 \cdot 10^{30}</math></b>	$\approx 1.15 \cdot 10^{77}$	$\approx 1.36 \cdot 10^{390}$
$2^qs$	$\approx 5.12 \cdot 10^3$	$\approx 6.33 \cdot 10^{31}$	$\approx 2.96 \cdot 10^{79}$	$\approx 1.76 \cdot 10^{393}$
$2^qs - s + 1$	$\approx 5.11 \cdot 10^3$	$\approx 6.33 \cdot 10^{31}$	$\approx 2.96 \cdot 10^{79}$	$\approx 1.76 \cdot 10^{393}$
$2^{q-1}s + \frac{s}{2}$	$\approx 2.56 \cdot 10^3$	$\approx 3.16 \cdot 10^{31}$	$\approx 1.48 \cdot 10^{79}$	$\approx 8.83 \cdot 10^{392}$
$2^q \lfloor \frac{s}{2} \rfloor + 2$	$\approx 2.56 \cdot 10^3$	$\approx 3.16 \cdot 10^{31}$	$\approx 1.48 \cdot 10^{79}$	$\approx 8.83 \cdot 10^{392}$
$2^{aq}$	$\approx 2.62 \cdot 10^5$	$\approx 4.01 \cdot 10^{59}$	$\approx 7.78 \cdot 10^{1078}$	$\approx 7.73 \cdot 10^{5461}$
$(1 + a)^q$	$\approx 1.96 \cdot 10^4$	$\approx 1.71 \cdot 10^{47}$	$\approx 1.20 \cdot 10^{301}$	$\approx 1.63 \cdot 10^{1524}$
<b>sq!e</b>	$\approx 9.86 \cdot 10^6$	$\approx 2.53 \cdot 10^{158}$	$\approx 5.96 \cdot 10^{509}$	$\approx 3.91 \cdot 10^{3476}$
$1 + s + s \frac{q!}{2}(e + 1)$	$\approx 6.74 \cdot 10^6$	$\approx 1.73 \cdot 10^{158}$	$\approx 4.08 \cdot 10^{509}$	$\approx 2.67 \cdot 10^{3476}$
<b>q!a<sup>q</sup>e<sup>1/2</sup></b>	$\approx 1.85 \cdot 10^8$	$\approx 5.91 \cdot 10^{185}$	$\approx 2.19 \cdot 10^{800}$	$\approx 2.67 \cdot 10^{4958}$

Table 10.1: Upper bounds for the number of player nodes in the number guessing game and in MasterMind. “NG (*n*)” denotes the number guessing game with the computer choosing values from 1 to *n*, “MM (*n* × *k*)” denotes the MasterMind game with *n* positions and *k* colours.

The first important thing to notice here is that not only is the problem NP-hard, also the input variables are huge. Even a simple MasterMind game with 4 colours and 4 positions already has 256 situations and 256 questions, which makes upper bounds containing factors like  $2^s$ ,  $2^q$  or  $q!$  rather painful. BlackBox (with the standard configuration of 5 dots on an 8 × 8 grid) has almost 8 million situations, and Minesweeper has a whopping  $1.7 \cdot 10^{13}$  of them, which is already a large number by itself, before using it as an exponent.

The second interesting observation is that not the same of the upper bounds is optimal in all games; which one is lowest depends on the configuration of *s*, *q* and *a*. The bounds with  $2^qs$  are however among the lower bounds for all games we looked at, whereas  $2^s$  is optimal in some and far from it in others.

Even the upper bound **sq!e**, in spite of containing  $q!$ , is relatively low in several of the games in the tables. **q!a<sup>q</sup>e<sup>1/2</sup>**, on the other hand, is fairly high in all games. However, we remember that the bound **sq!e** came from the standard game tree, whereas **q!a<sup>q</sup>e<sup>1/2</sup>** came from the behavioural game tree. We also know that the behavioural game tree has less player nodes than the standard game tree, since each node in the behavioural tree is equivalent to an entire information set of player nodes in the standard game tree. Therefore, **q!a<sup>q</sup>e<sup>1/2</sup>** does not seem to be a very good upper bound for the number of nodes in the behavioural game tree.

	MS ( $10 \times 10$ )	BB ( $8 \times 5$ )	BB ( $5 \times 2$ )	Life V1 (5)	Life V2 (5)
s	17 310 309 456 440	7 624 512	300	32	46 656
q	100	32	20	32	32
a	10	34	22	32	46 656
$2^s - 1$	$\approx 10^{5\,193\,092\,836\,932}$	$\approx 10^{2\,287\,353}$	$\approx 10^{90}$	$\approx 10^9$	$\approx 10^{13\,996}$
$2^q s$	$\approx 10^{43}$	$\approx 10^{16}$	$\approx 10^8$	$\approx 10^{11}$	$\approx 10^{14}$
$2^q s - s + 1$	$\approx 10^{43}$	$\approx 10^{16}$	$\approx 10^8$	$\approx 10^{11}$	$\approx 10^{14}$
$2^{q-1} s + \frac{s}{2}$	$\approx 10^{42}$	$\approx 10^{16}$	$\approx 10^8$	$\approx 10^{11}$	$\approx 10^{14}$
$2^q \lfloor \frac{s}{2} \rfloor + 2$	$\approx 10^{42}$	$\approx 10^{16}$	$\approx 10^8$	$\approx 10^{11}$	$\approx 10^{14}$
$2^{aq}$	$\approx 10^{300}$	$\approx 10^{326}$	$\approx 10^{132}$	$\approx 10^{307}$	$\approx 10^{447\,897}$
$(1 + a)^q$	$\approx 10^{400}$	$\approx 10^{192}$	$\approx 10^{100}$	$\approx 10^{192}$	$\approx 10^{512}$
sq!e	$\approx 10^{745}$	$\approx 10^{216}$	$\approx 10^{110}$	$\approx 10^{199}$	$\approx 10^{209}$
$1 + s + s \frac{q!}{2} (e + 1)$	$\approx 10^{745}$	$\approx 10^{216}$	$\approx 10^{110}$	$\approx 10^{199}$	$\approx 10^{209}$
$q! a^q e^{\frac{1}{s}}$	$\approx 10^{1100}$	$\approx 10^{384}$	$\approx 10^{200}$	$\approx 10^{384}$	$\approx 10^{704}$

Table 10.2: Upper bounds for the number of player nodes in Minesweeper, BlackBox, and the Life game. “MS ( $n \times k$ )” denotes the Minesweeper game with  $k$  mines on an  $n \times n$  grid, “BB ( $n \times k$ )” denotes the BlackBox game with  $k$  dots on an  $n \times n$  grid, “Life V1 ( $n$ )” and “Life V2 ( $n$ )” denotes versions one and two of the Life game with  $n$  choices. The numbers are only approximated, since in some cases already calculating the exact upper bounds can take rather long.

For some small games, we can in addition compare these upper bounds directly to the actual number of player nodes, shown in Table 10.3.

	NG (100)	MM ( $3 \times 3$ )	MM ( $2 \times 6$ )	BB ( $4 \times 3$ )	BB ( $5 \times 2$ )
s	100	27	36	560	300
q	99	27	36	16	20
a	2	10	6	18	22
$2^s - 1$	$\approx 1.26 \cdot 10^{30}$	$\approx 1.34 \cdot 10^8$	$\approx 6.87 \cdot 10^{10}$	$\approx 3.77 \cdot 10^{168}$	$\approx 2.03 \cdot 10^{90}$
$2^q s$	$\approx 6.33 \cdot 10^{31}$	$\approx 3.62 \cdot 10^9$	$\approx 2.47 \cdot 10^{12}$	$\approx 3.67 \cdot 10^7$	$\approx 3.14 \cdot 10^8$
$2^q s - s + 1$	$\approx 6.33 \cdot 10^{31}$	$\approx 3.62 \cdot 10^9$	$\approx 2.47 \cdot 10^{12}$	$\approx 3.66 \cdot 10^7$	$\approx 3.14 \cdot 10^8$
$2^{q-1} s + \frac{s}{2}$	$\approx 3.16 \cdot 10^{31}$	$\approx 1.81 \cdot 10^9$	$\approx 1.23 \cdot 10^{12}$	$\approx 1.83 \cdot 10^7$	$\approx 1.57 \cdot 10^8$
$2^q \lfloor \frac{s}{2} \rfloor + 2$	$\approx 3.16 \cdot 10^{31}$	$\approx 1.81 \cdot 10^9$	$\approx 1.23 \cdot 10^{12}$	$\approx 1.83 \cdot 10^7$	$\approx 1.57 \cdot 10^8$
$2^{aq}$	$\approx 4.01 \cdot 10^{59}$	$\approx 1.89 \cdot 10^{81}$	$\approx 1.05 \cdot 10^{65}$	$\approx 4.97 \cdot 10^{86}$	$\approx 2.83 \cdot 10^{132}$
$(1 + a)^q$	$\approx 1.71 \cdot 10^{47}$	$\approx 1.31 \cdot 10^{28}$	$\approx 2.65 \cdot 10^{30}$	$\approx 2.88 \cdot 10^{20}$	$\approx 1.71 \cdot 10^{27}$
sq!e	$\approx 2.53 \cdot 10^{158}$	$\approx 7.99 \cdot 10^{29}$	$\approx 3.64 \cdot 10^{43}$	$\approx 3.18 \cdot 10^{16}$	$\approx 1.98 \cdot 10^{21}$
$1 + s + s \frac{q!}{2} (e + 1)$	$\approx 1.73 \cdot 10^{158}$	$\approx 5.46 \cdot 10^{29}$	$\approx 2.48 \cdot 10^{43}$	$\approx 2.17 \cdot 10^{16}$	$\approx 1.35 \cdot 10^{21}$
$q! a^q e^{\frac{1}{s}}$	$\approx 5.91 \cdot 10^{185}$	$\approx 1.08 \cdot 10^{55}$	$\approx 3.83 \cdot 10^{69}$	$\approx 2.54 \cdot 10^{33}$	$\approx 1.71 \cdot 10^{45}$
Actual number	5 050	662	2 139	73 216	14 133

Table 10.3: Comparing the upper bounds for the number of player nodes to the actual number of nodes. (NG = NumberGuessing, MM = MasterMind, BB = BlackBox.)

As we can see, all our upper bounds are still several orders of magnitude away from the actual problem sizes. Thus, there seems to be still a lot of room for improvement of the upper bounds discussed in Chapter 5.

## 10.2 Runtime comparison

As we have seen in Section 9.3, there are many ways to combine optimization options. Comparing the runtimes for all of them at the same time would be rather unwieldy, so we will instead step by step show some comparisons of things that we had claimed to make the algorithm faster.

We will compare the runtimes for three games:

- Number guessing, where “NumberGuessing ( $n$ )” denotes the number guessing game with the computer choosing values from 1 to  $n$ ;
- MasterMind, where “MasterMind ( $n \times k$ )” denotes the MasterMind game with  $n$  positions and  $k$  colours; and
- BlackBox, where “BlackBox ( $n \times k$ )” denotes the BlackBox game with  $k$  dots on an  $n \times n$  grid.

Unless stated otherwise, all measured runtimes are for calculating the average-optimal solution. All measurements were performed on a Lenovo W510 with 4GB RAM and 1.73GHz Quad-Core (though calculations typically use only one core, except for Java’s garbage collection in the background).

### 10.2.1 Behavioural game tree vs. Decision set graph

Turning off the cache effectively downgrades Algorithm 6.10 (solving the game on the decision set graph) to Algorithm 4.36 (solving it on the behavioural game tree). This allows us to easily compare these two algorithms.

The performance gains we can see in Table 10.4 are quite impressive, thereby showing that all the hard work in Chapter 4 paid off.

Game	Behavioural game tree	Decision set graph
NumberGuessing (10)	157ms	2ms
NumberGuessing (15)	18 302ms	12ms
NumberGuessing (20)	3 566 172ms	9ms
MasterMind ( $2 \times 5$ )	81 487ms	106ms
MasterMind ( $3 \times 3$ )	13 535ms	110ms
BlackBox ( $3 \times 2$ )	18 234ms	54ms
BlackBox ( $6 \times 1$ )	1 070 315ms	73ms

Table 10.4: Runtime comparison of Algorithm 4.36 (solving the game on the behavioural game tree) and Algorithm 6.10 (solving it on the decision set graph).

We can also simply compare the number of player nodes in the behavioural game tree to that in the decision set graph (by counting how many nodes were calculated in each algorithm), shown in Table 10.5.

Game	Behavioural game tree	Decision set graph
NumberGuessing (10)	19 683	55
NumberGuessing (15)	4 782 969	120
NumberGuessing (20)	1 162 261 467	210
MasterMind ( $2 \times 5$ )	4 489 456	766
MasterMind ( $3 \times 3$ )	554 677	662
BlackBox ( $3 \times 2$ )	1 229 041	578
BlackBox ( $6 \times 1$ )	29 177 585	305

Table 10.5: Comparison of the number of player nodes in the behavioural game tree and in the decision set graph.

### 10.2.2 Original weights vs. Simplified weights

Next, we claimed that simplifying the weights to use only integers improved performance, so let us in Table 10.6 compare the original weights from Algorithm 6.10 to the simplified ones in Algorithm 6.16.

Game	Original weights	Simplified weights
NumberGuessing (100)	3 393ms	2 457ms
NumberGuessing (120)	5 362ms	5 123ms
MasterMind (2 × 6)	1 255ms	406ms
MasterMind (5 × 3)	2 478 656ms	1 826 198ms
BlackBox (5 × 2)	4 379ms	3 531ms
BlackBox (4 × 3)	16 903ms	17 562ms

Table 10.6: Runtime comparison of Algorithm 6.10 (using fractions as weights) and Algorithm 6.16 (using integers as weights).

While being far away from the previous result in terms of impressiveness, we see that the simplified weights perform slightly better, up to a factor of 3 in some smaller problems.

However, we also see that in case of BlackBox (4 × 3), they seem to perform a little *worse* than the original weights; a result that could be reproduced in several test runs. This might be due to the fact that fractions can be cancelled after every step, whereas the integer weights can become fairly large.

### 10.2.3 Calculation of all children vs. Upper bounds

After the simplified weights, we introduced upper bounds. What is surprising is that even on the behavioural game tree, upper bounds by themselves already vastly outperform the naive algorithm. Of course, using upper bounds *and* the decision set graph still performs better than either of the optimizations by itself, as we can see in Table 10.7.

Game	Orig-BGT	UB-BGT	Orig-DSG	UB-DSG
NumberGuessing (20)	1 913 056ms	9 665ms	31ms	27ms
MasterMind (3 × 3)	13 715ms	396ms	543ms	193ms
BlackBox (6 × 1)	1 040 888ms	1 821ms	750ms	160ms
BlackBox (3 × 2)	17 090ms	2 430ms	198ms	156ms

Table 10.7: Runtime comparison of upper bounds (UB) to the original algorithm (Orig) on the behavioural game tree (BGT) and the decision set graph (DSG). (Simplified weights were used for all calculations.)

Since the numbers for the decision set graph are all in a very low runtime range, let us in Table 10.8 compare their runtimes for some slightly larger problems.

Game	Orig-DSG	UB-DSG
NumberGuessing (100)	2 911ms	2 907ms
MasterMind (4 × 3)	9 786ms	4 755ms
BlackBox (6 × 2)	27 809ms	44 215ms

Table 10.8: Runtime comparison of upper bounds (UB) to the original algorithm (Orig) on the decision set graph (DSG). (Simplified weights were used for all calculations.)

We can see that MasterMind is indeed faster by around 50%, and there is no noticeable change in NumberGuessing. However, BlackBox is suddenly doing worse by a factor of 1.5. After being a bit puzzled

by this, we can however see that this result is explained by the amount of player nodes that are calculated: 53 749 without using upper bounds, compared to 134 691 when using them. The problem here is that the calculations of player nodes are very often aborted because of getting above the limit, thereby preventing the results for the node from being stored in the cache.

Let us therefore also look at the number of player nodes calculated for each of these games, shown in Table 10.9.

Game	Orig-DSG	UB-DSG
NumberGuessing (100)	5 050	5 050
MasterMind (4 × 3)	15 646	14 139
BlackBox (6 × 2)	53 749	134 691

Table 10.9: Comparison of the number of player nodes calculated when using upper bounds (UB) and when using the original algorithm (Orig), both on the decision set graph (DSG).

We see three different situations here:

- For NumberGuessing, exactly the same number of nodes is calculated, and the runtimes are almost identical.
- For MasterMind, fewer nodes are calculated, and the calculated nodes are aborted earlier, thereby leading to lower runtimes.
- For BlackBox, *more* nodes are calculated than before, and the runtimes are a fair bit higher.

Using or not using upper bounds therefore is an important consideration when deciding on an algorithm.

### 10.2.4 Given question order vs. Randomized question order

We also mentioned in Section 6.5.4 that in practice, randomly changing the order of questions can improve performance by finding good upper bounds more quickly. Some values supporting that claim are shown in Table 10.10.

Game	Given order		Randomized order	
	Runtime	Nodes	Runtime	Nodes
NumberGuessing (100)	3 010.3ms	5 050	21 844.4ms	40 694.3
MasterMind (4 × 3)	4 833.6ms	14 139	4 686.2ms	13 484.4
MasterMind (2 × 6)	642.7ms	3 708	631.1ms	3 579.8
BlackBox (6 × 2)	44 807.3ms	134 691	43 174.1ms	126 704.1
BlackBox (4 × 3)	58 914.1ms	221 225	53 486.6ms	193 092.2

Table 10.10: Comparison of runtime and number of calculated player nodes between evaluating questions in the given order or randomizing their order. (Simplified weights and upper bounds were used for all calculations. Values are the average of 10 test runs.)

In BlackBox and MasterMind we see very small performance improvements (though in case of BlackBox by far not enough to outweigh the performance loss compared to not using upper bounds at all).

However, what is an optimization for these games completely obliterates the performance of NumberGuessing. Once again it has to be pointed out that upper bounds have an advantage and a disadvantage: the advantage is that unnecessary calculations can be aborted, the disadvantage however is that the results of calculations that are aborted are not cached, so these calculations potentially have to be repeated. When evaluating the questions of the number guessing game in the given order from 1 to  $n - 1$ , which is actually a very inefficient order, then the upper bounds for all sub-calculations are so high that no calculation is aborted ever (so using upper bounds does not help at all, but at least does not harm either). When evaluating the questions in random order, however, then good upper bounds are found more quickly; and as a consequence, many calculations are aborted and have to be repeated later.

### 10.2.5 Normal question evaluation vs. Question evaluation with fingerprints

In the next step, we started calculating fingerprints. The first advantage that we claimed they had was that the calculation of the answer to a question would be faster. We therefore in a first experiment use fingerprints only to evaluate answers, but use the normal lists of remaining situations otherwise. The following runtimes shown in Table 10.11 do not include the time for calculating the fingerprints.

Game	Normal question evaluation	Question evaluation from fingerprints
NumberGuessing (120)	6 693ms	11 730ms
MasterMind (5 × 3)	916 645ms	839 012ms
BlackBox (4 × 3)	57 953ms	26 824ms
BlackBox (6 × 2)	44 125ms	22 371ms

Table 10.11: Runtime comparison between using the original game, or calculating fingerprints and using them for evaluating questions. (Simplified weights and upper bounds were used for all calculations.)

As we can see, the performance improvements are most prominent in BlackBox, where evaluating the path of a light ray takes some time. They are negligible in MasterMind, and in NumberGuessing, fingerprints even perform worse than direct calculation (which is not surprising, given that evaluating a question there is a simple comparison).

### 10.2.6 Lists of remaining situations vs. Greatest common fingerprint masks

A big part of Chapter 6 was about greatest common fingerprint masks as descriptors for the set of remaining situations. We will compare three options here: using lists of remaining situations, using greatest common fingerprint masks, and using both. When using greatest common fingerprint masks, we can in addition either use the original string labels for the results cache, or the integer labels created from the greatest common fingerprint masks. This gives five combinations that we can compare, all shown in Table 10.12 (using rows for the options this time, for easier comparability within one game).

Game	NumberGuessing (120)	MasterMind (3 × 5)	BlackBox (6 × 2)
L + S	11 223ms	194 727ms	23 993ms
GCFM + S	18 496ms	2 393 570ms	457 659ms
GCFM + I	14 867ms	1 939 506ms	442 883ms
L+GCFM + S	12 251ms	198 386ms	32 508ms
L+GCFM + I	10 996ms	204 496ms	30 100ms

Table 10.12: Runtime comparison between using lists (L), greatest common fingerprint masks (GCFM), or both (L+GCFM), using either string labels (S) for the cache or integer labels (I) created from the greatest common fingerprint mask. (Simplified weights and upper bounds were used for all calculations.)

As we can see, fingerprint masks do not perform quite as well as we might have hoped. The runtimes for using only greatest common fingerprint masks are very high in all scenarios; when using lists in addition, then the performance in NumberGuessing and MasterMind is at least not worse than before.

For BlackBox, however, the performance suffers a fair bit from using greatest common fingerprint masks. Thus, scientific honesty demands to acknowledge that while fingerprints seemed like a good idea, they simply do not meet the expectations we had for them. We may however take solace from the fact that they do reduce the storage space needed for the label of a known result. In BlackBox (8 × 5), for instance, fingerprints have a length of 152 bits, whereas saving situation lists directly (as a sequence of bits with one bit per situation, indicating whether it is included in the set) takes 7 624 512 bits.

Thus, once our problem sizes get to such a range that we cannot store all results in the cache at the same time any more, fingerprints will start to pay off again by allowing more results to be cached, thereby reducing the number of nodes that have to be (re-)calculated (while at the same time not being too detrimental to the calculation effort for each step).

### 10.2.7 Summary

The one thing that hugely improved performance was the transformation into the decision set graph. For all other optimizations, there are different factors that have to be considered; they pay off in some cases, but harm in others:

- The simplified weights pay off in most cases, though there are some exception where the fractions, which allow cancelling numerator and denominator after each step, perform better than handling the large integers we get.
- Upper bounds behave very differently in different games. In some games, they improve performance by up to 50%. In other games they reduce performance by up to 50%, since they prevent some intermediate results from being cached.
- Randomization of the order improved performance slightly in two games, but let the runtimes explode in the third. Again, it is something to be chosen with care.
- Fingerprints, finally, reduce performance for smaller to medium-sized problems, but might help to improve performance in larger problems where being able to cache more results starts to outweigh their slightly higher calculation costs.

## 10.3 Heuristics and random strategies

Let us see how well the heuristic and random strategies presented in Chapter 8 perform compared to the optimal one. Average scores achieved by these strategies in some games are shown in Table 10.13, and the corresponding runtimes for calculating them can be found in Table 10.14.

Game	BB ( $6 \times 2$ )	BB ( $4 \times 3$ )	MM ( $4 \times 3$ )	MM ( $3 \times 5$ )	NG (200)
Best solution	5.1746	6.1839	2.0370	2.6080	7.7200
Greedy	5.6683	6.7536	2.5679	2.9440	7.7200
Weighted Greedy	5.6683	6.7536	2.5679	2.9440	7.7200
Information Gain	5.6254	6.6571	2.2222	2.8240	7.7200
Information Gain per cost	5.4143	6.2946	2.0741	2.6240	7.7200
Gini Gain	5.5413	6.7571	2.1975	2.8400	100.4950
Gini Gain per cost	5.3905	6.5000	2.0741	2.6240	100.4950

Table 10.13: Comparison of average scores achieved by several heuristic algorithms, as well as average scores achieved in the optimal strategy. (BB = BlackBox, MM = MasterMind, NG = NumberGuessing.)

Game	BB ( $6 \times 2$ )	BB ( $4 \times 3$ )	MM ( $4 \times 3$ )	MM ( $3 \times 5$ )	NG (200)
Best solution	45 941ms	62 786ms	6 304ms	204 890ms	43 682ms
Greedy	238ms	139ms	223ms	61ms	113ms
Weighted Greedy	237ms	134ms	42ms	68ms	73ms
Information Gain	3 367ms	2 155ms	833ms	2 755ms	5 109ms
Information Gain per cost	2 846ms	2 297ms	917ms	2 674ms	5 510ms
Gini Gain	409ms	254ms	112ms	267ms	7 180ms
Gini Gain per cost	365ms	318ms	115ms	267ms	7 346ms

Table 10.14: Comparison of runtimes needed for evaluating the heuristic algorithms from Table 10.13. (BB = BlackBox, MM = MasterMind, NG = NumberGuessing. For calculating the optimal solution, simplified weights and upper bounds were used.)

As we can see, those are all fairly close to the optimal results (with the exception of Gini Gain in NumberGuessing), while allowing much lower runtimes. Already a simple greedy strategy seems to perform well both in terms of results and in terms of runtime.



For comparison, let us in Table 10.15 also have a look at the average costs that a player will have to pay when he plays completely randomly. (As mentioned earlier, calculating these values takes quite some effort, which is why we can only compare slightly smaller problems than for heuristics.)

We have three options: In option one (A), the player chooses estimates with the same probability as everything else. In the second (B), he chooses estimates with a probability of 50%. In the third option (C), he magically knows whether it is better to guess or to ask a random question. That is, he chooses an estimate if and only if the expected costs of an estimate are lower than the expected costs for asking a random question.

For comparison, Table 10.15 also includes the optimal and two heuristic strategies.

Game	NG (200)	MM ( $3 \times 4$ )	MM ( $4 \times 3$ )	BB ( $4 \times 3$ )	BB ( $5 \times 2$ )
Best solution	7.7200	2.0370	2.2188	6.1839	4.8100
Greedy	7.7200	2.5679	2.7813	6.7536	5.3033
Information Gain per cost	7.7200	2.0741	2.2500	6.2946	4.9133
Random A	$\infty$	$\infty$	$\infty$	8.4551	6.7359
Random B	$\infty$	$\infty$	$\infty$	11.0099	8.4702
Random C	9.7561	2.7861	3.0050	7.4083	6.2907

Table 10.15: Comparison of average scores achieved by several random algorithms, as well as average scores achieved in the optimal and in heuristic strategies. (NG = NumberGuessing, MM = MasterMind, BB = BlackBox; Random A = Random (estimate with same probability as question), Random B = Random (estimate with probability 50%), Random C = Random (estimate if better than asking).)

(In MasterMind and NumberGuessing, taking an estimate without having a clue gives infinite penalties.)

As we can see, heuristics perform worse than the optimal strategy, but (usually) still better than playing randomly. By and large, it seems that optimal, heuristic and random strategies are rather close to each other; even playing these games entirely randomly, the results are in the same order of magnitude as playing an optimal strategy.

## 10.4 MasterMind ( $4 \times 4$ )

We of course have to have a look at the classical MasterMind with 4 pegs and 4 colours. Looking at the runtime comparisons above, we see that MasterMind does benefit from simplified weights, upper bounds, and randomized child evaluation order, but not from fingerprints. Using these settings, we therefore get, within a runtime of a mere 10 hours, the result that the best achievable average score in  $4 \times 4$  MasterMind takes on average  $\frac{649}{256} \approx 2.535$  questions (not counting the last one with the correct code) – a result that matches the one from [30].

## 10.5 BlackBox ( $8 \times 5$ )

Last but certainly not least, we of course want to estimate how long calculating the optimal strategy for the standard BlackBox game with 5 dots on an  $8 \times 8$  grid would take.

To do so, we look at the player nodes in the behavioural game tree that we can reach with three questions (and answers), as described in Section 6.8.1 on distributing the calculation. We then try to estimate on the one hand how many such player nodes there are and how many subgames we consequently would have to calculate, and on the other hand how long each of these calculations would take.

### 10.5.1 Calculating fingerprints

In our implementation, calculating fingerprints for all situations takes around 11 minutes (on the computer described earlier in Section 10.2).

### 10.5.2 Estimating the number of nodes after three questions

In order to estimate the number of nodes we can first count, with a calculation effort of around 2 days, that there are 396 player nodes after asking 1 question (and getting 1 answer), and 61 464 player nodes after asking 2 questions (and getting 2 answers)<sup>1</sup>.

For the third layer, we reuse the observations from Section 5.3.5, where we saw that each set of remaining situations can be described as an intersection of some of the sets of remaining situations reached after asking one question. In particular, this means that every set of remaining situations that we can reach after asking three questions can be represented as the intersection of one set we reach after one question and one set we reach after two questions. (Let  $N$  be a player node reached after three questions and answers, and let  $(q_1, a_1)$ ,  $(q_2, a_2)$  and  $(q_3, a_3)$  be those questions and answers. Using the notations from Section 5.3.5, the set of remaining situations in  $N$  therefore is  $\hat{S}((q_1, a_1), (q_2, a_2), (q_3, a_3))$ . Then there exists one player node after two questions where  $\hat{S}((q_1, a_1), (q_2, a_2))$  is the set of remaining situations, and one player node after asking one question where  $\hat{S}((q_3, a_3))$  is the set of remaining situations. Therefore, by Corollary 5.20 the set of remaining situations in  $N$  can be calculated as the intersection of these two sets.)

If we represent the player nodes after asking one or two questions by their greatest common fingerprint masks, then we can calculate their intersection directly from the masks:

**Theorem 10.1.** *Let  $(M_1, m_1)$  and  $(M_2, m_2)$  be two fingerprint masks. We define  $M := M_1 \mid M_2$  and  $m := m_1 \mid m_2$ , as well as  $\overline{M} := M_1 \& M_2$ .*

*Then*

$$\hat{S}(M_1, m_1) \cap \hat{S}(M_2, m_2) = \begin{cases} \hat{S}(M, m) & m_1 \& \overline{M} = m_2 \& \overline{M} \\ \emptyset & \text{otherwise} \end{cases}.$$

*Proof.*  $\hat{S}(M_1, m_1) \cap \hat{S}(M_2, m_2)$  contains those situations in  $\mathcal{S}$  that match both fingerprint masks,  $(M_1, m_1)$  and  $(M_2, m_2)$ .

$\overline{M}$  represents the bits which are fixed in both fingerprints. If they are fixed to different values (and thus,  $m_1 \& \overline{M} \neq m_2 \& \overline{M}$ ), then no situation can simultaneously match both fingerprints, so the intersection is empty.

Otherwise, all bits that are fixed in at least one fingerprint are fixed in the intersection, and have to take the values to which they are fixed in the two fingerprints. This is a necessary and sufficient condition to describe the situations that match both fingerprints.  $\square$

Note that the fingerprint mask  $(M, m)$  defined in this theorem does not necessarily have to be the greatest common fingerprint mask of  $\hat{S}(M_1, m_1) \cap \hat{S}(M_2, m_2)$ ; it is only one out of possibly many such fingerprint masks.

We therefore take each of the 396 greatest common fingerprint masks of player nodes we reach after one question, and intersect it with each of the 61 464 greatest common fingerprint masks of player nodes we reach after two questions. From the 24 339 744 fingerprint masks that we get that way, we remove duplicates and get 11 541 000 different fingerprint masks (though some of them might still represent the same set of remaining situations, as they are not necessarily greatest common fingerprint masks).

Consequently, each set of remaining situations in a player node reached after three questions is represented by one of these 11 541 000 fingerprint masks, so there can be *at most* 11 541 000 such player nodes.

### 10.5.3 Estimating the calculation time for a subgame after three questions

For estimating the calculation time for each of these subgames, we randomly choose 200 of them and calculate optimal strategies for them (using simplified weights, no upper bounds, and fingerprints<sup>2</sup>). In our experiments, this took on average around 236 seconds.

### 10.5.4 Estimating the total calculation time

Combining these two estimates – there are at most 11 541 000 player nodes that can be reached by asking three questions, and calculating the subgame for each of them might on average take around 236 seconds –, we can therefore estimate that the total calculation time would be around 86 years.

<sup>1</sup>Lists of those nodes are included on the DVD that is part of this thesis.

<sup>2</sup>We use fingerprints here in spite of their worse performance, since we need them for storing the list of player nodes after the first 3 questions efficiently.

# Chapter 11

## Conclusions and outlook

There are basically three conclusions to be drawn here:

The first conclusion is that no matter what we do, the problem stays NP-hard. We might have squeezed out every little bit of performance improvement we could get<sup>1</sup>, but at the end of the day, there are still exponential factors in the asymptotic runtimes, and calculation times for reasonably sized problems are far out of reach.

The second conclusion is that heuristics might be interesting to look at.

The currently known ones, like Information Gain and Gini coefficient, have two drawbacks: On the one hand, they cannot take different questions costs into account, and different *answer* cost even less so. On the other hand, they have been optimized over decades for problems in artificial intelligence, which have different properties than deduction games: In artificial intelligence, the most important thing is how well a decision tree created from some training data classifies new data. From that point of view, a strategy that needs a bit more questions on average but performs well on test data is much better than a strategy that needs fewer question, but fails on practical examples. In game theory, on the other hand, we do not care about such things at all; we simply want a strategy whose average is as low as possible. From the game theoretic point of view, these heuristics have therefore been fine-tuned for the wrong criteria.

Finding better heuristics than those might therefore be an interesting and mostly uncharted area.

The third conclusion is that maybe our problem statement was too general. Looking at Chapter 2, we see what a large number of completely different games and problems can be expressed in our model. However, a too broad problem definition can also be a burden.

Restricting the problem definition one way or the other might therefore allow optimizations that have not been possible in this very general approach.

---

<sup>1</sup>Except maybe for having implemented it in Java. No matter how well Java might do on recent performance benchmarks for numeric tasks (see [6], for example), it simply is not the first choice when it comes to performance optimization. However, we chose Java mostly for convenience and readability; the implementation itself (for any fixed choice of optimization options) is only a few hundred lines long and can easily be ported to any other procedural language, like C++ or Python.

# Index

- actual dimension, 91
- available answers, 10
- average-optimal, 38
- average-optimal strategy, 38, 39, 70, 106
  
- BattleShips, 5, 8, 12, 17, 33, 134
- behavioural game tree, 53–57, 62, 65–67, 69, 70, 73–80, 84, 104, 105, 112, 144, 175, 176, 178, 179
- BlackBox, 4, 5, 8–10, 12, 22–24, 34, 42, 134, 135, 165, 176–183
- Bulls and Cows, 15, 16, 32, 33
  
- classification, 12, 25, 34, 35, 103–106, 162
- classification game, 103
- code-breaker, 5, 8
- code-maker, 5, 8
- complete information, 36
- computer, 5, 8
- cost solution, 39
  
- decision set, 49
- decision set graph, 5, 6, 62, 63, 65–72, 74–76, 84, 94, 110, 112, 113, 134, 144, 159, 160, 178–180
- decision tree, 25, 32, 34, 35, 102–104, 106, 169–171, 185
- decision tree, 34, 35, 103, 171
- deduction game, 5, 8, 10
- deductive game, 8
  
- encompasses, 65
- encompassing set, 67
- estimate, 11, 13, 14, 16, 34, 40–42, 44–47, 53–56, 58–60, 62–72, 74, 101, 103–107, 109, 110, 112, 114, 117, 120–124, 151, 157, 161, 163, 164, 167, 168, 170, 172–174, 183
- extensive-form game, 37
  
- fingerprint mask, 6, 146–158, 175, 181
- full actual dimension, 91–94, 96, 99
  
- game tree, 37
- greatest common fingerprint mask, 146–158, 174, 181, 184
  
- MasterMind, 10, 12, 15, 19, 32, 33, 39, 42, 134, 171, 176, 178–181, 183
- Minesweeper, 12, 19–21, 33, 134, 162, 176, 177
  
- NP-complete, 33, 34, 103, 106
- NP-hard, 5, 102, 158
- number guessing, 8, 9, 12–14, 43, 51, 52, 54, 75, 134, 136, 144, 162–164, 169, 176, 178, 179, 181
- number guessing with fixed penalty, 14
- NumberGuessing, 180
  
- optimal strategy, 38
  
- perfect information, 36
- perfect path, 38
- perfect play, 38
- player, 5, 8
- possible answers, 10
  
- reasonable strategy, 103
- remaining situations, 5, 6, 52, 57–59, 62–70, 73–76, 78–80, 84, 85, 89, 90, 94, 103, 104, 110, 111, 113, 114, 116, 117, 121–123, 134, 143, 144, 146–149, 151–160, 162, 164–169, 174, 181
  
- scales, 12, 29, 30, 162, 168
- situation fingerprint, 134
- situation subtree, 49
- size, 65
- standard game tree, 41, 54–57, 62, 73, 74, 176
- strategy, 38
- strategy solution, 39
- strategy subtree, 104
- subgame, 37
- subgame perfect equilibrium, 38
  
- worst-case-optimal, 38
- worst-case-optimal strategy, 38, 39, 58, 70

# Bibliography

- [1] <http://brettspielwelt.de/Hilfe/Anleitungen/BlackBox/>.
- [2] [http://en.wikipedia.org/wiki/Battleship\\_\(game\)](http://en.wikipedia.org/wiki/Battleship_(game)).
- [3] [http://en.wikipedia.org/wiki/Black\\_Box\\_\(game\)](http://en.wikipedia.org/wiki/Black_Box_(game)).
- [4] <http://mathworld.wolfram.com/Mastermind.html>.
- [5] <http://nothings.org/games/minesweeper/>.
- [6] <http://shootout.alioth.debian.org/u64q/java.php>.
- [7] [http://www.answerbag.com/q\\_view/168447](http://www.answerbag.com/q_view/168447).
- [8] <http://www.boardgamegeek.com/boardgame/165/black-box>.
- [9] <http://www.boardgamegeek.com/boardgame/2425/battleship>.
- [10] <http://www.linuxlinks.com/Software/Games/Strategy/Warfare/Battleships/>.
- [11] <http://www.minesweeper.info/archive/MinesweeperStrategy/website.html>.
- [12] <http://www.minesweeper.info/archive/PlanetMinesweeper/avant.php>.
- [13] <http://www.minesweeper.info/wiki/Strategy>.
- [14] <http://www.squidoo.com/battleships>.
- [15] <http://www.top500.org/>.
- [16] Robert J. Aumann and Sergiu Hart, editors. *Handbook of Game Theory with Economic Applications, Vol. 1*. North-Holland, Amsterdam, 1992.
- [17] Kristin P. Bennett. Global tree optimization: A non-greedy decision tree algorithm. Department of Mathematical Sciences, Rensselaer Polytechnic Institute, Mar 1994.
- [18] Lotte Berghman, Dries R. Goossens, and Roel Leus. Efficient solutions for Mastermind using genetic algorithms. *Computers & OR*, 36(6):1880–1885, 2009.
- [19] Leo Breiman, Jerome Friedman, Charles J. Stone, and R. A. Olshen. *Classification and Regression Trees*. Wadsworth, 1984.
- [20] Shan-Tai Chen and Shun-Shii Lin. Novel algorithms for deductive games. In *International Computer Symposium, Dec. 15-17, 2004, Taipei, Taiwan*, pages 517–522, 2004.
- [21] Shan-Tai Chen and Shun-Shii Lin. Optimal algorithms for  $2 \times n$  Mastermind games – a graph-partition approach. *The Computer Journal*, 47(5):602–611, May 2004.
- [22] Shan-Tai Chen, Shun-Shii Lin, and Li-Te Huang. A two-phase optimization algorithm for Mastermind. *The Computer Journal*, 50(4):435–443, 2007.
- [23] Shan-Tai Chen, Shun-Shii Lin, Li-Te Huang, and Sheng-Hsuan Hsu. Strategy optimization for deductive games. *European Journal of Operational Research*, 183(2):757–766, 2007.

- [24] Vašek Chvátal. Mastermind. *Combinatorica*, 3(3):325–329, 1983.
- [25] Saher Esmeir and Shaul Markovitch. Anytime learning of decision trees. *Journal of Machine Learning Research*, 8:891–933, May 2007.
- [26] Glenn Firebaugh. Empirics of world income inequality. *American Journal of Sociology*, 104:1597–1630, 1999.
- [27] Oleg German and Evgeny Lakshtanov. "Minesweeper" and spectrum of discrete Laplacians. *Applicable Analysis*, 89(12), Dec 2008.
- [28] Corrado Gini. *Variabilità e mutabilità (Variability and Mutability)*. 1912.
- [29] Wayne Goddard. A computer / human Mastermind player using grids. *South African Computer Journal*, 30:3–8, 2003.
- [30] Wayne Goddard. Mastermind revisited. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 51:215–220, 2004.
- [31] Shahar Golan. Minesweeper on graphs. *Applied Mathematics and Computation*, 217(14):6616–6623, Mar 2011.
- [32] Earl Harris. Information Gain versus Gain Ratio: A study of split method biases. In *Seventh International Symposium on Artificial Intelligence and Mathematics (AMAI, AI&M, ISAIM)*, number 11-2002, 2002.
- [33] Albrecht Heeffer and Harold Heeffer. Near-optimal strategies for the game of Logik. *ICGA Journal*, 2007.
- [34] Manfred J. Holler and Gerhard Illing. *Einführung in die Spieltheorie*. Springer, Berlin Heidelberg New York, 5th edition, 2003.
- [35] Li-Te Huang, Shan-Tai Chen, and Shun-Shii Lin. Exact-bound analyzes and optimal strategies for Mastermind with a lie. In H. Jaap van den Herik, Shun-Chin Hsu, Tsan-Sheng Hsu, and H. H. L. M. Donkers, editors, *Advances in Computer Games, 11th International Conference, ACG 2005, Taipei, Taiwan, September 6-9, 2005, Revised Papers*, volume 4250 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 2006.
- [36] Li-Te Huang and Shun-Shii Lin. Optimal analyses for  $3 \times n$  AB games in the worst case. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games*, volume 6048 of *Lecture Notes in Computer Science*, pages 170–181. Springer, 2009.
- [37] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5:15–17, 1976.
- [38] R. W. Irving. Towards an optimum Mastermind strategy. *Journal of Recreational Mathematics*, 11(2):81–87, 1978-1979.
- [39] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer*, 22(2):9–15, 2000.
- [40] Richard Kaye. Infinite versions of Minesweeper are Turing complete. University of Birmingham, May 2007.
- [41] Donald E. Knuth. The Computer as Mastermind. *Journal of Recreational Mathematics*, 9(1):1–6, 1976.
- [42] Ron Kohavi. Scaling up the accuracy of Naive-Bayes classifiers: A decision-tree hybrid. In *Proceedings on the Second International Conference on Knowledge Discovery and Data Mining*, pages 202–207. AAAI Press, Aug 1996.
- [43] Barteld P. Kooi. Yet another Mastermind strategy. *ICGA Journal*, 28(1):13–20, 2005.
- [44] Kenji Koyama and Tony W. Lai. An optimal Mastermind strategy. *Journal of Recreational Mathematics*, 25:251–256, 1993.

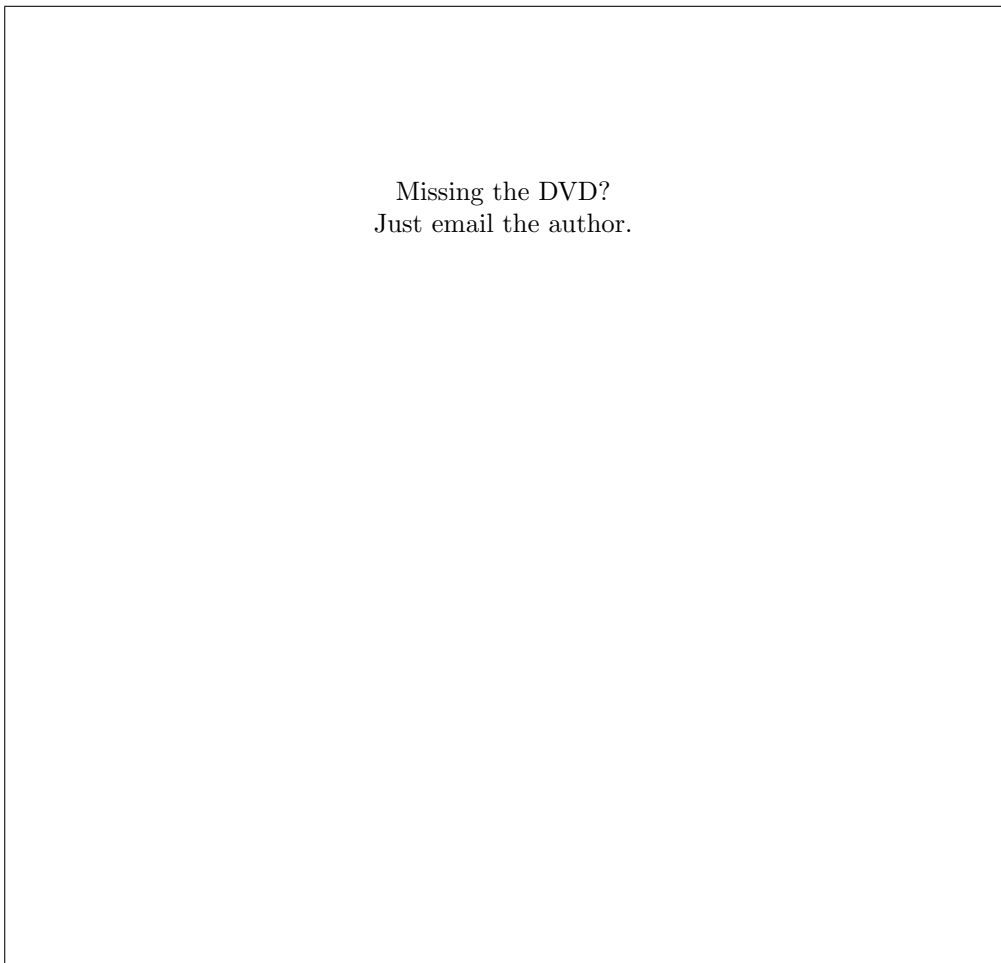
- 
- [45] Solomon Kullback and Richard A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86, 1951.
- [46] Andrew W. Moore. Information Gain (tutorial slides). School of Computer Science, Carnegie Mellon University, <http://www.autonlab.org/tutorials/infogain11.pdf>, 2001.
- [47] Preslav Nakov and Zile Wei. MINESWEEPER, #MINESWEEPER. Department of Computer Science, University of Warwick, 2003.
- [48] Erich Neuwirth. Some strategies for Mastermind. *Zeitschrift für Operations Research*, 26:257–278, 1982.
- [49] Hung Son Nguyen and Sinh Hoa Nguyen. From optimal hyperplanes to optimal decision trees. *Fundamenta Informaticae*, 34(1-2):145–174, 1998.
- [50] Lourdes Peña Castillo and Stefan Wrobel. Learning Minesweeper with multirelational learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 533–538. Academic Press, Aug 2003.
- [51] Kasper Pedersen. The complexity of Minesweeper and strategies for game playing. Technical report, Department of Computer Science, University of Warwick, 2003–2004.
- [52] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.
- [53] Merlijn Sevenster. Battleships as a decision problem. *ICGA Journal*, 27(3):142–149, 2004.
- [54] Ehud Shapiro. Playing Mastermind logically. *Sigart*, 85:28–29, 1983.
- [55] Jeff Stuckman and Guo-Qiang Zhang. Mastermind is NP-complete. Department of Electrical Engineering and Computer Science, Case Western Reserve University, Dec 2005.
- [56] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Addison-Wesley, 2005.
- [57] Roman Timofeev. Classification and Regression Trees (CART), Theory and applications. Master’s Thesis, Humboldt University, Berlin, Dec 2004.
- [58] Marta Vomlelová and Jiří Vomlel. Applying Bayesian networks in the game of Minesweeper. In *Proceedings of the Twelfth Czech-Japan Seminar on Data Analysis and Decision Making under Uncertainty, Litomyšl, Czech Republic, September, 2009*, pages 153–162, 2009.
- [59] Sichun Wang. Solving the optimal solution of weight vectors on GP-decision tree. In *Second International Conference on Intelligent Computation Technology and Automation, ICICTA '09*, volume 4, pages 329–332, Changsha, Hunan, China, Oct 2009.

# Appendix A

## DVD

Contents of the DVD:

- Folder **thesis**: L<sup>A</sup>T<sub>E</sub>Xsource and .pdf file of the thesis.
- Folder **implementation**: Java implementation of the algorithms.
- Folder **results**:
  - Folder **fingerprints**: Fingerprints (optimized version from Algorithm 6.41) of various games.
  - Folder **blackbox\_8\_5**: List of the player nodes of BlackBox ( $8 \times 5$ ) after asking at most three questions.



Missing the DVD?  
Just email the author.