Bernd MALLE, BSc.
Mat.Nr. 0130547

GRAPHINIUS
A Web based graph exploration and analysis platform

Master's Thesis

to achieve the university degree of

Master of Science (MSc)

Master's degree programme:

Software Development and Business Management

Supervisor:
Assoc. Prof. Dr. Andreas HOLZINGER
Institute for Information Systems and Computer Media
Graz University of Technology

Graz, May 2, 2016

This page intentionally left blank

STATUTORY DECLARATION

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

Graz, 03.05.2016

Bernd Malle

This page intentionally left blank

Acknowledgements

First and foremost, I would like to thank my family, friends and colleagues for their enduring mental and emotional support over the past several years while pursuing my studies.

A special thank you also goes to my supervisor Prof. Andreas Holzinger, who has not flinched when I changed the subject of my Master Thesis three times over the past two and a half years.

Finally, I need to thank many students and professors I met while studying economics at KF Uni Graz during the early 2000s. Without their continuous deterrent this Master thesis in Software Development could never have come into existence.

This page intentionally left blank

Abstract

Graphs are a fundamental tool of mathematics and can be applied to a very diverse field of modern scientific areas: network routing, social network & community analysis, image processing, even Anonymization of patient data or fraud detection via belief propagation networks.

A relatively novel addition to that spectrum is the emerging field of computational biology, in which we can find protein-protein interaction networks, metabolics, or connectome graphs. Since Biologists, Medical professionals or Privacy researchers are usually no tech experts, an intuitive, GUI-based research platform could facilitate rapid experimental iterations. Graphinius aims to be such a Web-based, graph theoretical platform offering real-time in-browser computations as well as tightly integrated visualization, interaction, and manipulation of graphs.

In this thesis I will mainly introduce GraphiniusJS and its underlying design principles; however, I am in the lucky position of already having guided the development of a graph visualization library called GraphiniusVIS in the context of a colleague's Master's Project.

Aside from presenting some real-world use cases I will also provide an outlook on the whole, emerging Graphinius platform and its exciting capabilities for research and education.

Keywords

Web based research platform, graph theory, graph visualization, graph mining, machine learning, data engineering, ML metrics, ML heuristics, algorithmic pipelines, WebGL, interactive Machine Learning

ÖSTAT classification

1140 Software-Engineering

ACM classification

Software infrastructure

This page intentionally left blank

Kurzfassung

Graphen sind ein fundamentales, mathematisches Werkzeug und können in vielen wissenschaftlichen Bereichen eingesetzt werden: Netzwerk routing, Soziale Netzwerke, Bildverarbeitung sowie Anonymisierung von Patientendaten oder Betrugsanalyse sind nur einige Beispiele.

Ein relativ neues Betätigungsfeld ist jenes der mathematischen Biologie, in welcher wir Protein-Protein Interaktions-Netzwerke oder metabolische bzw. neuronale Graphen vorfinden. Da Biologen, Ärzte und Datenschützer für gewöhnlich keine Experten in Programmierung sind, würde diesen eine GUI basierte Platform zur intuitiven Berechnung von Graphen entgegenkommen. Dieses Ziel verfolgt Graphinius, indem es eine Browser basierte Graphberechnungs, -interaktions sowie -visualisierungs Plattform bereitstellt.

In dieser Diplomarbeit lege ich hauptsächlich die Kernbibliothek Graphinius-JS und deren Designprinzipien dar; darüberhinaus hatte ich bereits die Ehre, ein Masterprojekt zur Entwicklung einer integrierten Visualisierungsbibliothek names GraphiniusVIS mitzubetreuen.

Nach der Präsentation dreier realistischer Anwendungsfälle und deren Ergebnisse, bildet ein Ausblick auf zukünftige Technologien und Potentiale für die weitere Entwicklung von Graphinius den Abschluss dieser Arbeit.

Schlüsselwörter

Webbasierte Forschungsplattform, Graphentheorie, Graphenvisualisierung, Graph-Auswertung, Maschinelles Lernen, Dateninfrastrukturen, ML Metriken, ML Heuristiken, Algorithmische Sequenzen, WebGL, interaktives Machinelles Lernen

ÖSTAT Klassifikation

1140 Software-Engineering

ACM Klassifikation

Software infrastructure

This page intentionally left blank

Table of Contents

1. Motivation

Choosing a suitable topic for a Master's Thesis is not a readily decided matter. On one hand a student desires to show some insight into contemporary and maybe even complex problems, but on the other hand the task has to be achievable within a certain timeframe. The project should be theoretical enough to be of interest even for professors while at the same time (at least in Software Development) be practical enough to remaining motivating to the student. In the case of this Thesis I tried to combine scientific and engineering aspects into a project which could be expanded in future endeavors while keeping it sufficiently self contained to represent a work of its own.

1.1 Scientific motivation

Having been a member of the HCI-KDD.org group for over 2 years now, I have developed a genuine interested in graph theory, machine learning, HCI as well as their applications in modern information systems - not least in the context of biomedical applications. Although finalizing my Master's study at a relatively advanced age, I am still open to pursuing a PhD in those areas. Therefore, it seemed logical to tackle some problems related to those fields; however significant progress in such matters are not easily achieved even by professional scientists, leave alone a single MSc student on a deadline.

For this reason I intended to contribute to the future work of Professor Holzingers Team by concerning myself with scientific matters without the pretension of being capable of improving on complex research efforts. As a result, I built on work already conducted in my Master's project by underpinning it with a more general and expandable Software architecture. Assisting (data) scientists by providing a better underlying infrastructure to accelerate their experimental iterations and making demanding processing steps available to researchers outside the field of computer science or software development, has been dubbed 'data engineering' over the recent

years (Lorica, 2013a).

Popular projects like Apache Hadoop or Spark, Google's recently released TensorFlow and many other, more specialized clowd-based research platforms, fall under this category - albeit they boast very different properties and advantages and are therefore suitable for different, although overlapping, use cases. The Graphinius library (and future platform) presented in this thesis is unique in the sense that it allows computations directly on the client, but without requiring any installation, by utilizing a piece of software contained in any browser - the JavaScript virtual machine (JSVM).

1.2 Engineering motivation

As a student of Software Development and Business Management the two aspects of modern software architectures and their repercussions on the next generation of business models are fascinating and of great importance to me. Especially the emergence of powerful JavaScript virtual machines in modern browsers as well as access to the GPU from inside the browser sandbox open up new opportunities for startup companies in many fields that were hitherto restricted to conservative Software Development paradigms. Physics simulations, graph theoretical machine learning tasks, the visualization and manipulation of complex data structures as well as console games can now be implemented on a general, ubiquitous platform without much loss of performance or usability. In the field of traditional Web applications, servers can act more and more as simple database-abstracting backends with near-zero computational and scalability requirements - especially if combined with cheap, global content delivery networks. Several factors on the other hand, like 1) handing much of the business logic over to the client side, 2) an ever increasing complexity in cooperation between backend services, as well as 3) emerging standards like websockets (enabling realtime communication through publish/subscribe over traditional networking architectures) put new challenges to a guild of developers used to mainly write server-side code such as Java, PHP or Ruby on Rails.

As a consequence, the following thesis is an attempt at merging my scientific curiosity with my drive towards new, exciting Software Development methodologies into a working, expandable prototype of a future Web based graph exploration,

analysis & research platform. I am glad if I was partially able to live up to that goal.

1.3 Business motivation

Many people in our time might be disconcerted about today's frantic pace of development and progress in all areas related to science and engineering. To the flexible, energetic, ambitious and well-educated mind however, those same processes hold great promise for personal development, career opportunities and entrepreneurship. Especially in the field of Web-based software development, it has never been easier to achieve business success and acknowledgement than today. Supported by a technology infrastructure that is easy to use, easy to scale, powerful and at the same time elegant to handle (not to forget exceptionally affordable), the modern web developer is always but one good idea away from potentially founding the 'next big thing'.

On the other hand, as opportunities are to be found everywhere, developers from all over the world are eager to jump on the newest trends, may they be social or mobile or virtual. In this setting, it becomes ever more important to combine programming knowledge with scientific and economic competency.

The successful software platform of the future will, in my humble opinion, consist of three distinct pillars: 1) A real-time Web based communication platform, 2) Machine Learning capabilities, especially smart recommendations based on insights into personal behavior and goals), and 3) Compelling interactive visualization. Based on this insight and the fact that almost any modern system exhibits network characteristics, I am motivated to develop Graphinius either into its own platform covering all three areas mentioned above, or into one powerful client-side graph component driving a larger system as described in Section 3.1.2.

2. Introduction

The following sections shall give a brief introduction into how and why project Graphinius came into existence, what challenges we want to tackle building it, as well as the general objectives and potential new applications it might enable or make feasible in the future.

2.1 What is Graphinius?

Graphs are a fundamental tool of mathematics and can be applied to a very diverse field of modern scientific areas: network routing (information, traffic, logistics), social network / community analysis, image processing, NLP operating on graphs of document spaces and topic clusters as well as fraud detection via belief propagation networks (all of the mentioned and a few others will be described in Chapter 3).

A relatively novel addition to that spectrum is the emerging field of computational biology, in which we can find PPI, metabolics, or connectome graphs. Since Biologists, Medical professionals as well as Health-Science or Privacy / Data Security Researchers are usually no tech experts, an intuitive, GUI-based research platform could facilitate rapid experimental iterations. Graphinius aims to be such a Web-based, graph theoretical platform offering real-time in-browser computations as well as tightly integrated visualization, interaction, and manipulation of graph structures.

In this thesis I will mainly introduce GraphiniusJS and its underlying design principles; however, I am in the lucky position of already having guided the development of a graph visualization library in the context of a colleague's Master's Project. This WebGL / Three.js based library called GraphiniusVIS uses low-level datastructures and concepts, enabling it to visualize up to 15k nodes / 40k edges fluently (25 FPS) inside a web browser even on middle-class laptops. I will also provide an outlook on the whole, emerging Graphinius platform and its exciting capabilities for research and education.

2.2 The history of Graphinius

When I met Andreas Holzinger in November 2013 he presented me with a 'crazy' idea: To analyze dermatological images not by traditional image processing methods, but by first transforming them into a graph structure and subsequently apply graph mining algorithms with the goal of obtaining results at least comparable to the quality achievable as of date. My first experiments in what I came to term *graph extraction from images* were conducted in Matlab and progressed rather modestly; being a developer used to employ C-style programing languages and Object oriented paradigms, Matlab seemed a peculiar (and very costly) way to perform specific, handwritten algorithms which did not lend themselves especially well to matrix representations and calculations. Moreover I was concerned with the fact that even if I succeeded in doing a great job in extracting usable graphs (and it wasn't even clear what that meant) I would not easily be able to share my work with colleagues outside the research community. Providing C++ / Python or Java Code in a Github repository for developers having working installations of those environments to reproduce, is standard by today. Installations of Matlab (or Octave with specific add-on packages installed) however are not very widespread. Furthermore, in case my results turned out to be interesting to the public, any live demonstration of the software on unprepared systems would be clearly impossible.

After years of having utilizing Web based technologies and continuously researching new frameworks and improvements in that area, it seemed to me that if a document suite (Google Docs, Office365) could be implemented in JavaScript (and even more demanding projects like an entire virtual box inside the browser (Bellard, 2015)), the time might be ripe to apply modern JavaScript Engines to the task of graph extraction and processing too. Despite my expectations Professor Holzinger did not have any objections and I was given green light to develop a prototype.

This early work was focused purely on extracting graphs out of images without additional processing steps or subsequent graph analysis. Because I had to implement mathematical algorithms, I decided on using TypeScript, a typed superset of JavaScript for the development process; no frameworks or numerical libraries were used. Within a little more than a month, I had an example website up and running which could load images into a canvas, segment them by either a Watershed or

a Kruskal-based region merging algorithm and extract a graph from the label image; the results are available at http://berndmalle.com/graphext and have partly been published in Holzinger, Malle, and Giuliani, 2014. As far as performance was concerned, our crude and unoptimized JavaScript implementation fell just a little short of a comparable algorithm in Matlab, with great upward potential for using in-browser optimizations or even GPU-accelerated processing.

Of course there were downsides too - JavaScript engines are naturally single-threaded even in 2016, so long-running, intense computing tasks tend to block other functions. This is even true for other Browser-intern modules like the rendering engine or input event handling, so that the whole user experience is severely diminished when conducting excessive operations. However, a possibility to avert this behavior is the usage of Web Workers which have been introduced as a working draft as early as 2009, but until recently haven't been widely supported. Also, possible downsides to their usage in situations where large datastructures need to be copied from the main thread to a worker (and back) have to be considered. More interestingly though, it became clear rather soon that implementing our suite of algorithms as a Web based platform would bring with it several major advantages over traditional software approaches: amongst others are 1) easy reproducibility, effortless scalability and online meta machine learning opportunities; all of those advantages will be discussed in detail in subsequent chapters.

Building upon those insights Professor Holzinger and I came up with a project called iKNOdis.net - Interactive Knowledge Discovery in Networks - which was designed to cover the whole processing pipeline from image preprocessing over graph extraction to graph analysis and result visualization. In the process of defining those modules it became clear that alternative algorithms at the same stage in the processing pipeline would have to be interchangeable so that other developers could contribute new algorithms to a common algorithmic database. This in turn would benefit all users of the platform who would be able to apply 'community'-contributed algorithms to their own problems (and at their own peril). The originally proposed iKNOdis.net architecture can be seen in Figure 2.1.

In the process of creating an FWF proposal out of iKNOdis, the project was split into two separate endeavors in order to make them amenable to independent, smaller funding efforts as well as enabling separate groups of developers to work on their

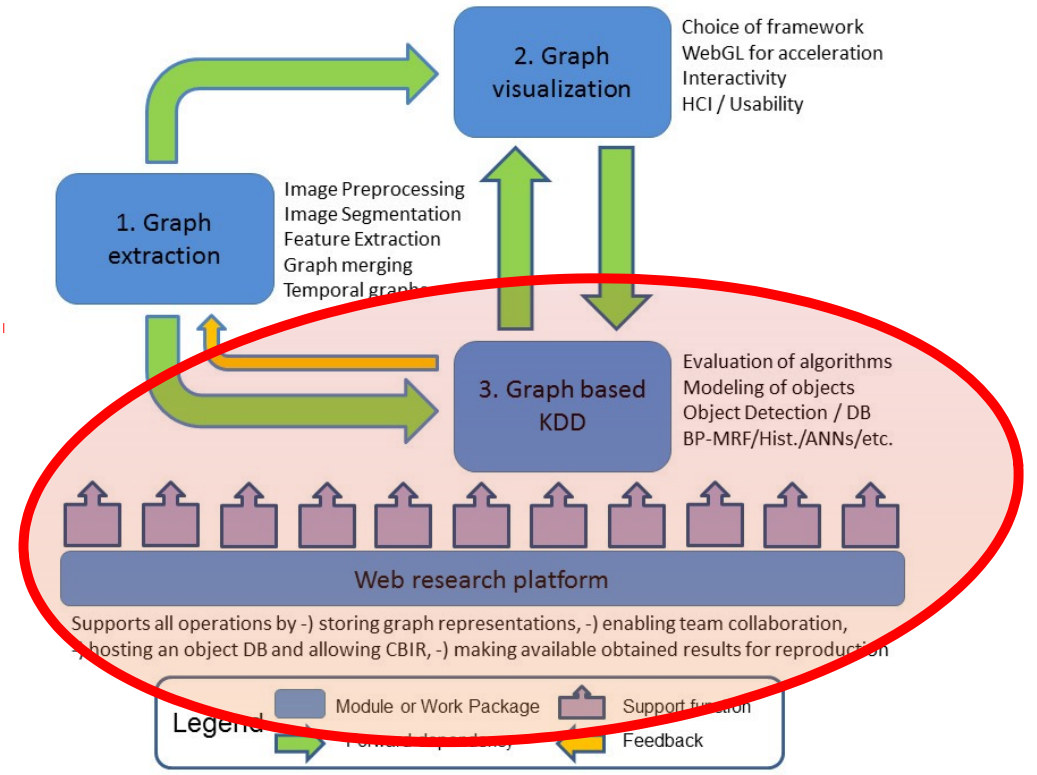


Figure 2.1: Former project iKNOdis architecture overview

Whereas iKNOdis.net was meant to be used exclusively for image processing via graph-theoretical approaches, we broadened the idea of application areas while reducing the aim for specific scientific advances in image processing. The result is a Web-based, graph-theoretical processing platform whose base functionality (an in-browser graph library called GraphiniusJS) as well as visualization capabilities (GraphiniusVIS) are the main subjects of the Master Thesis at hand.

respective areas of interest. This resulted in the formation of project Graphinius covering all graph-theoretical aspects and basic data-engineering infrastructure Lorica, 2013a by providing a Browser based graph processing and visualization platform.

As a side-effect, Graphinius follows no specific research goal itself; we rather envision it as a future engineering infrastructure for data scientists, may they concern themselves with Machine Learning, Biomedical applications or any other field. This mental 'expansion' to a diverse set of interesting areas of application is reflected by an in-depth discussion of several potential candidates in Chapter 3 as well as 3 different demo applications described in Chapter 8.

As evolution (of ideas) never stops, there are already new ideas springing up concerning the future of Graphinius - the most obvious one regarding a spin-off company providing the platform as a hosting service for the community - such a startup could profit from several different advantages over its competitors. Second, GraphiniusJS alone (without integration into the larger platform) could power web applications of the future (see Section 3.1.2), enabling graph-based recommender systems for social networks that could be truly organized as *peer networks* - without or with only a minimal server infrastructure.

2.3 How this thesis is structured

The rest of this chapter is composed of a short description of how scientific computing / machine learning tasks are handled today. We will explore the benefits individual users as well as the whole research community could derive from conducting experiments on such a platform; therefore we have to outline potential properties and focal points of the proposed technology.

In Chapter 3, Theoretical background / applications we are introducing different potential areas of application for Graphinius. The reader will notice that practically all of the fields discussed may be of interest to researchers outside the 'hardcore' Machine Learning or Data Science communities.

Chapter 4 deals with possible economic opportunities involving the Graphinius platform and its conceivable business model, and will discuss some potential competitors.

An outline of the basic functionality of Graphinius will be given in Chapter 5, comprising technical features, graph-theoretical design decisions as well as handling characteristics important to its users.

The modern web development process will be discussed in-depth in Chapter 6, where we will become acquainted with a set of requirements for modern Web based software and then go on to explore suitable technologies in each category and compare them to one another. At the end of this chapter, we will have all the tools in our hand to start building Graphinius.

I will go into details about the project implementation in Chapter 7, introducing

each relevant subsystem in sequence, pointing out its role in the larger context and explain how and why it was built the way it was built.

In Chapter 8 the reader will find three demo applications that have been implemented on top of Graphinius as a proof of feasibility of the platform. I will introduce some specific (mathematical) background knowledge and provide the sample results obtained using Graphinius.

Chapter 9 will provide some structural as well as performance metrics about the Graphinius (JS) library and (proudly) present the test coverage achieved as of the date of this writing.

Taking a look into the future in Chapter 10, we will discuss some promising new concepts and technologies which hold great potential for the advancement of any Web based, community-oriented research platform.

I will finish this thesis with a summary and conclusion in Chapter 11, and give some additional information about anonymization outputs in Appendix A and provide the whole Graphinius API as of the time of this writing in Appendix B.

2.4 Today's Machine Learning / KDD approach

When i was taking the famous Machine Learning MOOC tought by Prof. Andrew Ng from Stanford University on Coursera in 2013, one story he conveyed during a section on optimizing Machine Learning Pipelines had especially caught my attention. As a specialist in high demand Prof. Ng is frequently consulting for Silicon Valley companies in matters of Machine Learning and Artificial Intelligence. On one of these occasions, the client company had been trying to optimize their ML pipeline for the better parts of 2 years without any significant improvements in their results. After looking at the different stages of their pipeline and conducting a so-called ceiling analysis, Prof. Ng concluded that two developers had spent 18 months on optimizing their background separation algorithm while the most significant potential for improvement really lay in a latter stage of the process. Based on this analysis the company was able to remedy the shortcomings in a relatively short period of time.

This incident shows how much effort is potentially squandered by trying to imple-

ment sophisticated algorithms within isolated teams in a non-standardized fashion: Proprietary approaches - both in technology as in methodology - hinder the exchange of information with other members of the research community, thus opening up vulnerabilities to making mistakes which could have easily been avoided by considering the experience of other professionals. The following properties of data analysis / ML projects seem to give rise to such vulnerabilities:

- **Isolation.** Working on common machine learning problems in isolated teams without communication makes comparison of approaches as well as results unnecessarily hard. Dealing with errors at any stage of the algorithmic pipeline takes more effort than necessary due to a lack of reference values, while achieving superb results has little to no effect on the potential of other projects.
- **Proprietary Software.** Countless professionals prefer developing data analysis pipelines in highly proprietary software environments like Matlab or Mathematica. This prevents an influx of solid, community-tested algorithms while preventing others from gaining knowledge acquired in such organizations (as long as they are unwilling to pay horrendously for the software).
- **Irreproducibility.** As unpublished code cannot be perfectly reverse engineered, experiments conducted in isolation can't be easily corroborated. This might be advantageous with respect to product development and patent procedures, but is usually detrimental to the efforts of researchers trying to get published and spreading their insights.
- **Lack of scalability.** Last but not least, heterogeneous and highly customized data processing pipelines might not lend themselves well to parallelization, which might prevent the use of such algorithms on quickly expanding datasets.

This leads us to the insight that a readily available (no complex setup or configuration), public and open source, standardized and scalable infrastructure which promotes the cross-fertilization of ideas and insights from individual experiments, would form the ideal model of a future, successful Machine Learning platform (graph-theoretical or otherwise).

2.5 A Web based approach to benefit the world

Over the past several years, many platforms have emerged in the realm of *data engineering*, providing means for researchers and data professionals to easily learn, deploy and scale machine learning models Lorica, 2013b. The greatest disadvantage of those platforms however, is that they all target the tech-savvy programming experts and experienced system administrators. While this may not be a problem for core ML researchers and programmers, it presents a serious obstacle to experts in the Life Sciences etc.

Second, most modern frameworks allow for the setup of huge and very efficient clusters on a hardware / filesystem virtualization level. They do not provide built-in community networking functions, however, which could bridge the experience gap between disciplines (e.g. via hyper heuristics) or allow for easy reproducibility of results. The following list shall give a short overview of potential advantages of the Graphinius Platform:

- **Ease of access.** With configuration and loading of binaries happening automatically from the outside, barely any costs for technical configuration and conducting experiments arise.
- **Effortless scalability** As users of the platform provide their own computing power via their browsers, the server role can initially be reduced to that of a static document server / database server.
- **Automatic grid potential.** As browsers were intended to always be connected to a larger network of computers, and 2-way communication via Web Sockets has recently become commonplace, we can imagine the JavaScript Virtual Machine (JSVM) as a natural node of a dynamic virtual grid computer - akin to a client node in Berkeley's BOINC framework (Seti@Home).
- **Centralization of experiment meta data.** As code and configuration will be stored and transmitted by a Web servers anyways, nodes will be able to send descriptions of conducted experiments back to the servers for storage and evaluation. Given enough participants on the platform, this would lead to the formation of ML meta knowledge, as to which algorithms in what sequence

would perform best given some inputs and specified problem class (prediction, description, ...).

- **Hyper Heuristics** are a way of selecting or configuring algorithms by searching a space of lower level heuristics instead of searching the solution space itself. Hyper heuristics are different from Meta Learning in that they work independent of the problem domain and therefore promise to be generally applicable; the challenges lie in producing algorithms with a good overall runtime behavior - therefore, meta-data about diverse experiments are required.

- **A Pipeline recommender** with the ability to automatically instantiate and execute the recommended (sequence) of algorithms could be built upon this knowledge, which would further enhance a researcher's capacity to conduct more experiments faster, potentially leading to an increase in frequency or quality of scientific output.

- **Instant deployment.** In traditional scientific programming, there is a large gap between developing a model suitable for a specific research question and actually deploying it in a production environment (Lorica, 2013c). Being Web based, Graphinius' development and production environment are one and the same, reducing the deployment procedure to posting a link to an experiment (for scalability, see above).

- **Reproducibility / Corroboration of results** This follows naturally considering the previous aspect, as clicking on a published link to an experiment, automatically downloading its code to one's browser and pressing the *start* button is really all that is needed to run a colleague's latest experiment on one's own computer.

- **Great visual capabilities.** Although there exist great visualization toolkits for many of the data science platforms out there today, the sheer mass of 2D and 3D libraries running in the browser enable completely new ways to demonstrate, interact with, and even manipulate results in real time. Moreover, exchanging of 'live' results (via simple bookmarks) instead of 'dead' graphs (in scientific papers) would soon become standard amongst members of such a community.

3. Theoretical background / applications

As graphs occur almost everywhere in reality (physics, chemistry, biology, society etc.) there are obviously countless opportunities for applying graph theoretical models to problems that were either graph-related from their inception / formulation as well as to cross-disciplinary challenges. The following is a microscopic selection of contemporary opportunities to apply graph theory to interesting problems - in areas that the author believes could be suitable to in-browser exploration and processing, thereby presenting ideal incentives for the development of Graphinius.

3.1 Social networks

Social networks are today's natural candidates for graph based algorithms, as they have been rising to power and fame over the previous decade and a half. Of course most social graphs in use today are far too big for any client or server side application to handle, and are therefore only interesting to programmers and architects of database clusters, high performance grid-computing developers and data-center engineers. Because of this, I am going to confine myself to the topic of local sphere recommenders, where I believe small graph computing to be able to have some real world influence. In order to get to this point, we will first need to take a look at the shape and size of typical recommendation processes (themselves forming subgraphs of larger networks), which in the following section will be termed 'cascades'.

3.1.1 Network recommendation analysis

Leskovec, Singh, and Kleinberg (2006) have examined recommendation networks crystallizing from purchases based upon previously received product recommendations. In order to do this, they employed an online shopping system observing the product categories of DVDs, Music, Books and Videos (VHS). Users of that system were modelled as nodes in a graph, with the graph initially being completely unconnected. In this system people who bought a product (and only actual purchasers)

were able to recommend the bought product to as many people as they wanted via email; this resulted in a *temporary* recommendation edge added between the two (user) nodes. This edge was then handled according to the following two criteria:

1. Recommendations received after a product was already bought by the receiving person were immediately deleted.

2. Recommendations received which did not result in the product being bought (during the observational period) were also deleted.

This procedure resulted not in a graph comprising all of the users and products bought throughout the system, but only a collection of - fragmented - subgraphs representing the recommendation cascades. The main question of the study then was to the size distribution of those cascades w.r.t. their count, and if properties of the original social network (e.g. density, degree distribution) had any influence on that distribution.

A second point of interest concerned the isomorphism classes of cascades, meaning their shape and size similarity. Therefore, a similarity measure had to be established, as the graph isomorphism problem is NP-hard and therefore impractical to use on a real-world study. This is why exact isomorphism matching was only used on cascades up to a graph size of 9 nodes; above that a graph *signature* was computed including singular values (via SVD) of the graph adjacency matrix up to a size of 500 nodes. Above that, the signature only consisted of the number of nodes and edges as well as a histogram of in- and out-edges per node (degree distribution). The relation between cascade amount and size can be seen in Figure 3.1 on page 31.

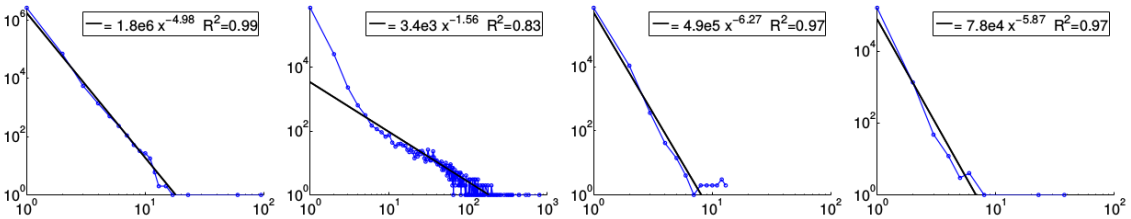
The results of this study after a two-year period can be summarized as follows:

- The largest cascade (which also form connected components) accounted for less than 2.5% of all nodes.

- Cascades did not only come in the form of trees (snowball effect) but form arbitrary graphs with splits, collisions as well as cycles.

- Splits are more common than collisions, however (as one would expect).

- The frequency of a cascade type (as computed by graph isomorphism) is not a strict monotonic function of cascade size, which points to the recommendation propagation process to be influenced by more subtle factors of the underlying social graph than just the network structure alone.
- Most cascades observed exhibited fewer than 9 nodes (with the exception of DVD recommendation cascades) and were of very small degree (just a little over 1 according to the visual representation found in the paper



(a) Book (b) DVD (c) Music (d) Video

Figure 3.1: Size distribution of recommendation cascades for four product categories
This diagram was taken from (Leskovec, Singh, and Kleinberg, 2006), page 7.

The above analysis holds several insights which in combination lead to a remarkable conclusion:

1. The cascades presented in the paper represent only 'successful' recommendations, i.e. the ones which receivers perceived as valuable enough to actually buy the product.
2. The goal of any recommender algorithm (regardless on which item space it operates) is to produce exactly such valuable recommendations.
3. Because most cascade sub-graphs were of very small size and degree, 'successful' recommendations can be assumed to originate from places in the direct neighborhood of a node.

3.1.2 The local sphere (idea)

The concept of a local sphere and computations applied to it comes from the author's (possibly incorrect, but natural) insight that the relevance of recommendations be-

haves as a function of node vicinity:

- Lets call the whole social graph and all interactions in it the 'global sphere'.
- Recommendations to users are then computed over the global sphere, which takes an amount of resources exponential to the size of the underlying graph.
- Let's further assume that 95% of all relevant (accepted) recommendations in a social network like facebook are those that are derived from the immediate local neighborhood of a node (less than 2 degrees, see section above..)
- This assumption is corroborated by the fact that two degrees are also what Facebook allows programmers (as of 2013) to query via their graph API from any authenticated user, apparently in an effort to prevent automatic traversal / exploration of their most valuable business asset.
- Let's call this immediate local neighborhood the 'local sphere'

Now let's also consider how modern publish/subscribe based frameworks (like Sails, Meteor, Hapi or Derby, only to mention some JavaScript libraries) handle data communication between server and client:

- The server offers some subscriptions on it's data, usually limiting access to items based on identity, authorization or user role provided by the client.
- The client defines some subscriptions on server-side data collections (tables), representing the client's wish for information regardless of it's status or authority.
- Publication as well as subscription can be seen as a mathematical subset of all the data in the database.
- An algorithm inside the respective framework resolves those (potentially conflicting) interests by computing the intersection set of the data provided / requested.
- The intersection data set is then pushed to the client (in our case the browser) as soon as it becomes available or is updated, which makes this model ideal for real-time interaction and communication between clients.

- The sum total of all the data pushed to the client is equivalent to the 'local sphere' we described earlier - HOWEVER - their inherent graph structure is lost during the transmission, so that the client can only see them as isolated fragments without context.

The combination of those two ideas now enables us to envision the following scenario:

- Instead of interpreting all data items in the local sphere as isolated entities, we retain their graph structure enabling the client to gain hitherto unachieved knowledge and insights into its already available data.
- We therefore need a graph library in the browser, not only to represent the local sphere graph, but also to analyze it in order to take intelligent actions that were previously reserved for the server-side (data center) infrastructure.
- No complex graph partitioning algorithm on the server is necessary, as we can use the natural set contraints inherent in any web application:
  - e.g. in a social network, the client will have access to all its immediate friends, social activities and interest groups
  - in a project management tool, the client naturally has access to the data of all team members, to-do lists, milestones, resources etc.

  If our 95%-relevance assumption mentioned earlier holds, we can achieve great scaling efficiency by introducing the local sphere concept:
  - the client can immediately perform computations like recommendations on the subgraph of the local sphere.
  - only recommendations accepted will have to be stored on the server (that is, cause additional network traffic).
  - the client is easily able to recompute the relatively small local graphs in real-time, offering responsiveness far beyond today's best (server-side) infrastructures.
  - as modern web frameworks transport all of the required data into the client store anyways, we do not add extra complexity to our servers and databases.

- On the other hand, questions of data security / privacy will have to be dealt with, as we are talking about preemptively filling the client memory with possibly otherwise unnecessary or superfluous data.

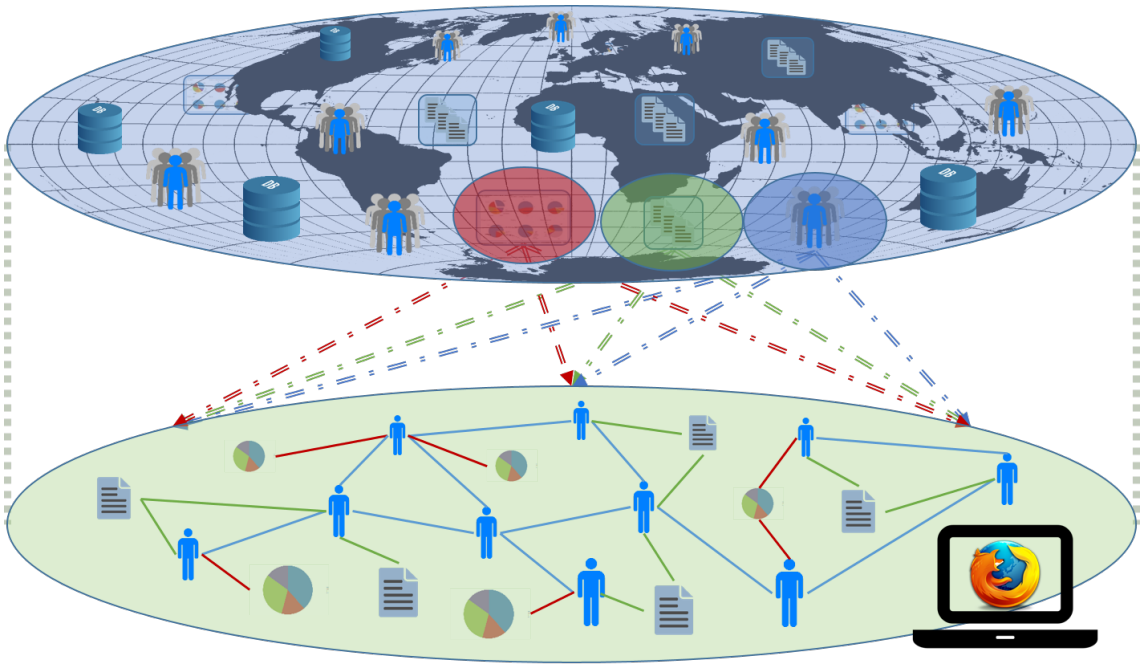


Figure 3.2: Local sphere projected from the global sphere

Only a very small portion of the global graph is actually visible from any connected client. The sum of all viewable items however, if properly conveyed to the client (i.e. with their connection information preserved), could form a subgraph of the whole network called the 'local sphere', which would allow the browser to utilize the underlying graph structure to extract hidden knowledge and perform graph computations on its own.

Needless to say, GraphiniusJS would be an ideal candidate to explore this concept further and could, if used appropriately on carefully modeled local spheres, enable start-ups to compete with much larger companies employing complex and very expensive machine learning infrastructures.

3.2 Graph based image processing

The overall goal of working with images using graph theory contains 3 different aspects: 1) Extracting graphs out of images, thereby laying the groundwork for applying the wealth of graph theory to a hitherto different problem domain, 2) Actually processing the resulting graph with appropriate methods in order to achieve

usable (interpretable) results, and 3) visualizing every step of the pipeline, thereby enabling researchers and domain experts to learn valuable lessons from new methods applied. Figure 3.3 illustrates how a prototypical workflow could look like.

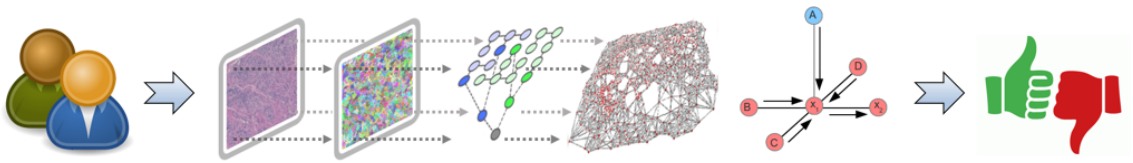


Figure 3.3: Graph based image classification example

1) a laser scan image of a nevus is oversegmented and 2) a graph extracted by interpreting region centroids as nodes and region adjacency as edges. 3) A belief propagation algorithm is applied to the resulting graph yielding 2) a converged state representing the nevus classification as benign or malignant.

3.2.1 Graph extraction

In the attempt to extract graphs out of images, aside from traditional image segmentation approaches (Felzenszwalb and Huttenlocher, 2004), there have been methods proposed for constructing object graphs from images, e.g. (Lee and Grauman, 2012). As for the task of merging several extracted graphs into one, we might build upon the work of (Schneevoigt, Schroers, and Weickert, 2014), in which they propose a 3-step pipeline for the reconstruction of 3D objects from 2D image structures by solely utilizing dense methods of correlation analysis (global energy features instead of sparse local feature vectors. For purposes of the latter, (Demetz, Hafner, and Weickert, 2013) proposes a new local descriptor called Complete Rank Transform which is morphologically as well as illumination invariant, while containing a maximally possible amount of information (Bobylev and Rjasanow, 2014). It would be interesting to see if those methods can also be applied to graphs while still retaining sufficient feature information. Moreover, any point cloud data (Holzinger et al., 2014) could be interpreted as graphs by enriching them with connections in order to form networks.

3.2.2 Graph processing

Lee and Grauman, 2012 propose to conduct object recognition based on pre-existing models of known object primitives and to construct object graphs in order to infer global scene understanding from local information.

Another way is to see graphs extracted from image sources as topographic maps. On such "landscapes" autonomous multi-agents (Kasaiezadeh and Khajepour, 2013) (Olfati-Saber, Fax, and Murray, 2007), e.g. ant-robots (Wagner and Bruckstein, 2001) could explore the terrain and leave markings on interesting spots.

From the field of topology, (Cerri, Fabio, and Medri, 2012) describe a method of shape comparison based on Topological Persistence utilizing Persistence Diagrams - collections of shape descriptors - and computing a distance function between them, while (Di Fabio and Landi, 2012) applies this idea to shape retrieval by showing partial similarity of such descriptors.

3.2.3 Graph visualization

As far as visualization is concerned, Graph layouts have been often applied, but because of the scale and complexity of real world data, these layouts tend to be dense and often contain difficult to read edge configurations (Herman, Melançon, and Marshall, 2000). Much previous work on graph layouts has focused on algorithmic approaches for the creation of readable layouts and on issues such as edge crossings and bends in layout aesthetics (Purchase, 1997). As an algorithm designer, the decision whether or not to preserve the mental map is more dependent on the tasks likely to be performed by users than previously assumed, however, much further experimentation is needed (Archambault and Purchase, 2013; Stahl et al., 2013).

In the context of Graphinius (VIS), an additional question presents itself in the form of 3D visualization of large data structures, especially since most layout algorithms to date are specifically limited to 2D projections.

3.3 Anonymization

The amount of patient-related data produced in today's clinical setting poses many challenges with respect to collection, storage and responsible use. For example, in research and public health care analysis, data must be anonymized before transfer, for which the k-anonymity measure was introduced and successively enhanced by further criteria like L-diversity, T-closeness as well as delta-presence (the latter of which is used to model the background knowledge of potential attackers).

Taking a look at Figure 3.4 will help the reader in understanding the original (tabular) concept of anonymization: Given an input table with several columns, we will probably encounter three different categories of data:

- **Personal identifiers** are data items which directly identify a person without having to cross-reference or further analyze them. Examples are first and last names, but even more so an (email) address or social security number (SSN). As personal identifiers are dangerous and cannot be generalized (see Figure 3.5) in a meaningful way (e.g. one could generalize the *address* field, which would only result in some kind of Zip code), this category of data is usually removed. The table shows this column in a red background color.

- **Sensitive data,** also called 'payload', which is the kind of data we want to convey for statistics or research purposes. Examples for this category would be disease classification, drug intake or personal income level. This data shall be preserved in the anonymized dataset and can therefore not be deleted or generalized. The table shows this column in a green background color.

- **Quasi identifiers**, colored in the table with an orange background, are data that in themselves do not directly reveal the identity of a person, but might be used in aggregate to reconstruct it. For instance, (Sweeney, 2002) mentioned that 87% of U.S. citizens in 2002 had reported characteristics that made them vulnerable to identification based on just the 3 attributes *zip code*, *gender* and *date of birth*. But although this data can be harmful in that respect, it might also hold vital information for the purpose of research (e.g. zip code could be of high value in a study on disease spread). The solution - and this

is the actual point of all anonymization efforts - is to generalize this kind of information, which means to lower its level of granularity. As an example, one could generalize the ZIP codes 41074, 41075 and 41099 to a generalized version 410**, as shown in Figure 3.6.

Name	Age	Zip	Gender	Disease
Alex	25	41076	Male	Allergies
...	...	...	...	...

Figure 3.4: The three types of data considered in (k-)anonymization

As described in (Ciriani et al., 2007), k-anonymization requires that in each data release every combination of values of quasi-identifiers must be identical to at least $k-1$ other entries in that release, which can be seen as a clustering problem with each cluster's (in the context of anonymization also called an 'equivalence class') internal quasi-identifier state being identical for every data point. This can be achieved via suppression and generalization, where suppression means simply deletion, whereas in generalization we try to retain some usable value.

The process of generalization works through a concept called *generalization hierarchies*, which form a tree whose root denotes the most general value available for a data category (usually the 'all' value) and then branches to more and more specific occurrences, with its leafs representing the set of exact, original values (see Figure 3.5). In generalizing some original input value, one traverses the tree from the leaf level upwards until a certain prerequisite is fulfilled. Usually, this prerequisite comes in the form of the k-anonymity requirement, so that we want to find a group of other data rows (=vectors) whose (generalized) quasi identifiers match the data point being processed.

Each level of generalization involves a certain cost in information loss though, which means we do not just want to construct our clusters in any sequence possible, but minimize the overall information loss. This makes k-anonymization an NP-hard optimization problem (because of an exponential number of possible generalized quasi-identifier combinations), leaving us to conclude that the k-Anonymity problem is to lose as little information as possible in a dataset while ensuring that the release

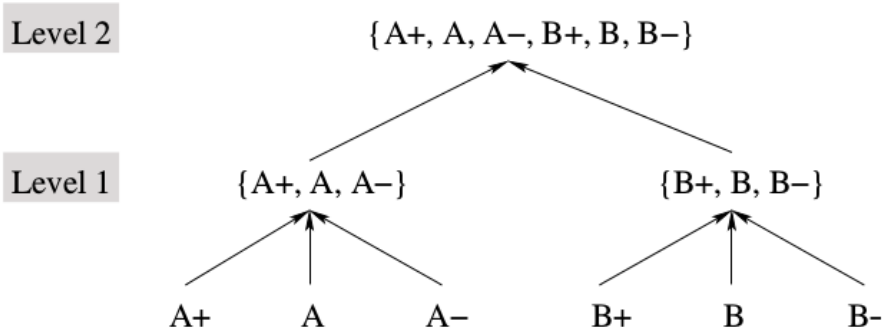


Figure 1: A possible generalization hierarchy for the attribute "Quality".

Figure 3.5: Example of a typical generalization hierarchy taken from (Aggarwal et al., 2005)

(the anonymized, publishable version of the dataset) satisfies the k-anonymization criterion (Aggarwal et al., 2005).

Node	Name	Age	Zip	Gender	Disease
X1	Alex	25	41076	Male	Allergies
X2	Bob	25	41075	Male	Allergies
X3	Charlie	27	41076	Male	Allergies
X4	Dave	32	41099	Male	Diabetes
X5	Eva	27	41074	Female	Flu
X6	Dana	36	41099	Female	Gastritis
X7	George	30	41099	Male	Brain Tumor
X8	Lucas	28	41099	Male	Lung Cancer
X9	Laura	33	41075	Female	Alzheimer

Node	Age	Zip	Gender	Disease
X1	25-27	4107*	Male	Allergies
X2	25-27	4107*	Male	Allergies
X3	25-27	4107*	Male	Allergies
X4	30-36	41099	*	Diabetes
X5	27-33	410**	*	Flu
X6	30-36	41099	*	Gastritis
X7	30-36	41099	*	Brain Tumor
X8	27-33	410**	*	Lung Cancer
X9	27-33	410**	*	Alzheimer

Figure 3.6: Tabular anonymization: input table and anonymization result

3.4 Graph (social network) anonymization

The whole last sections would have been out of place in a thesis regarding a graph platform and library, as it holds no direct references to graph computations. However, as social networks have gained huge popularity over the previous decade, and even modern medical databases come in the form of graph structures, the question of how to efficiently anonymize networks has gained ever more significance over the years.

As a start, one could see a graph just as a collection of nodes, where each node contains some kind of feature vector, akin to the row in a data table. Adopting that

view, we could be tempted to simply ignore the existence of edges and apply some kind of algorithm suitable to the anonymization of tabular data. The main problem with this however lies in the fact that the structural environment of a node (the constellation of its neighbors within the greater network) provides some additional information. That is, even if we successfully (k-)anonymize the feature vectors of a graph according to the methods found in the previous chapter, we still run the risk of leaving to much information in the form of a known local subgraph structure.

Consider Figure 3.7 for example, in which the nodes of a graph have already been k-anonymized into groups of size 3 and 7, respectively. In this figure, local subgraphs b) and c) are actually (3)-anonymized, because as each node has the exact same local neighborhood structure, the additional information of a node possessing a degree of 0 (or 2) is of no additional value. For local subgraphs a) and d) on the other hand, the additional information of a node being of degree (x) has the potential to reveal its identity, as it is not indistinguishable from its neighbors within the equivalence class any more.

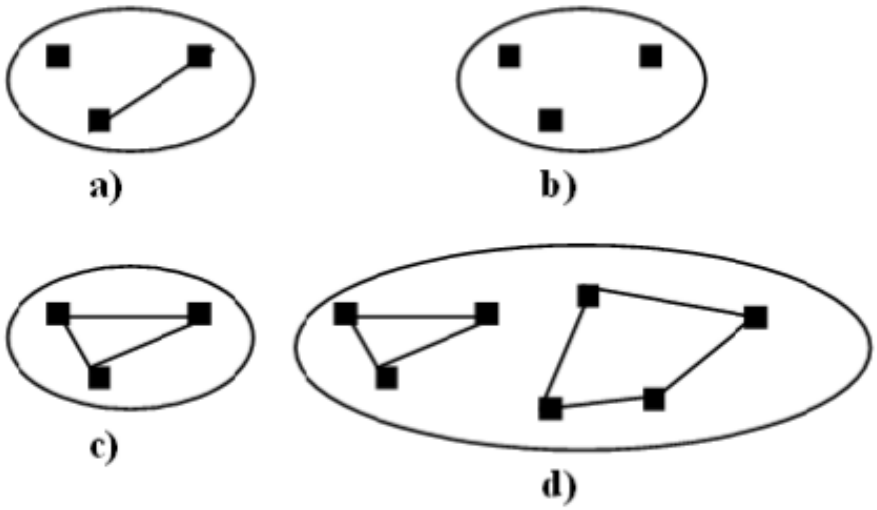


Figure 3.7: Local subgraph neighborhoods as additional anonymization obstacle.

(Example taken from Campan and Truta, 2009.)

Several methods have been proposed to make re-identification of nodes in anonymized social graphs harder. Chester et al., 2011 for example introduce the idea of vertex addition to labeled and unlabeled datasets. While an algorithm on the former remains NP complete, they provide an efficient ($O(nk)$) algorithm for unlabeled data.

Experimenting on several well known datasets, they show that commonly-studied structural properties of the network, such as clustering coefficient, are only minorly distorted by their anonymization procedure.

The authors of (Kapron, Srivastava, and Venkatesh, 2011) take the approach of adding edges to an edge-labeled graph like the Netflix movie database (with users and movies being nodes and edge weights representing movie ratings). They define tables as bipartite graphs and prove NP-hardness for the problems of neighborhood anonymity, i-hop anonymity and k-symmetry anonymity.

Campan and Truta, 2009, whose local subgraph problem we already encountered, proposed a solution in the form of a greedy clustering algorithm which takes into account not only the information loss incurred by generalizing features of nodes, but also introducing a structural loss function based on the local neighborhood within an equivalence class (and between them). The author of this thesis implemented that approach utilizing GraphiniusJS and will demonstrate the algorithm in Section 8.3 as well as the anonymized results in Appendix Section **??**.

3.5 Fraud detection

Anomaly detection via belief propagation in a Markov random field (BP-MRF) has been shown to work on large networks with high accuracy and realistic performance (Pandit et al., 2007), and can also be adapted to scale to hundreds of parallel machines as described in (Kang and Horng, 2010).

The idea behind this approach is that each node in a graph holds some belief about itself and is capable of forming opinions of its neighbors. Let's assume a group of people represented by nodes and edges denoting friendships between those nodes; let's further assume that a small subgroup of people are criminals and that criminals usually engage in friendship with other criminals, while non-criminals prefer to stay amongst themselves as well.

Given this model we can infer that if a node has a lot of edges to criminals, there is a higher chance for the underlying person to be a criminal as well. We first initialize this system with a certain belief structure (some nodes believe they are criminals, others don't) and we let nodes form opinions about each other ("you are

connected to me and I am a criminal, therefore you must be a criminal also"). If we subsequently initiate a turn-based exchange of those opinions with a slight chance per turn of "convincing" a neighboring node, that system might (but not necessarily has to) converge to a stable state - a final belief distribution about who is a criminal and who is not.

Although the example given might sound exotic at first, Belief Propagation has already successfully been applied to email spam classification and fraud detection in auction systems in 2007, improving the methods previously employed by significant margins.

4. The business case for Graphinius

An exciting new software project would only be half exciting if there was not the promise of economic success as well. In the case of an online platform, we are already somewhat limited in our options here. Basically, there are three business models that companies building on large quantities of individual users (in contrast to highly-specialized pro software or business services) can employ.

4.1 Potential business models

1. **The Facebook model.** Giving away the platform services completely for free, attracting potentially extremely large audiences, then trying to cash-in on services offered to external businesses - like advertisement or data analysis. The downside of this model is that it only works in areas where potential applications are so generic as to be of interest to many millions, even billions of people. A graph-theoretical research platform does not fall into that category...

2. **The pay-as-you-go model.** Many services, especially in the realm of cloud data centers, embrace this model as it gives them the opportunity to charge for their services in a very fine-grained manner, as in CPU hours or units of data traffic consumed. This model however requires the offering company to have very precise measuring capabilities in place, which makes it attractive for Amazon but rather uninteresting to small startups.

3. **The Premium membership model.** Giving away a base service for free, thereby trying to attract larger audiences, then cashing in on premium-offerings is another way of conducting business. It is suitable especially for smaller companies, as it requires only monthly billings via credit card payment which can be easily and cheaply performed today. In case the Graphinius platform should ever develop into a commercial offering, this model would certainly be the preferred choice.

4.2 Potential business sectors

Another important consideration in taking a product to the market is some form of potential analysis. Although we are not going to calculate possible user bases here, I want to at least mention some general market segments that might profit from a platform like Graphinius.

4.2.1 Education

Graph theory is an integral part of every Computer Science / Software Engineering degree and will extend more and more into Biology, BioMed, Medicine and the Social Sciences in the future. This means that millions of university students will at some point come into contact with graph-theoretical assignments, which on the other hand have to be designed, deployed, collected and assessed by university employees. Graphinius as an online, Web based platform would not only alleviate the hassle for students to setup their own development libraries and environments, but could also function as a central point of assignment submission & correction / grading.

4.2.2 Algorithm prototyping

Many companies have to apply their algorithms to graphs of enormous size (e.g. Facebook's going into the many billions, but biological networks are growing exponentially as well, if only for the progress in detection technology). However, designing new algorithms on a production graph of that magnitude is hardly practical, which means that much smaller test setups are usually used for algorithmic prototyping. Graphinius could offer such an environment either for open source development or as a premium service for closed source projects.

4.2.3 Community research platform

Representing essentially our base case, we already discussed in detail how a Web based approach holds many advantages over closed-source, isolated Machine Learning islands (see Section 2.5). In addition to this, let us just mention the success and influence Kaggle has gained within the data science community over the previous

years. If a platform only concerned with distributing interesting competitions can gain such a widespread reach, how great a potential would a platform hold which promotes not only communication and awareness, but enables code sharing and easy reproduction as well?

4.3 Remarks on potential competitor platforms

There are several cloud based services in the world offering machine learning APIs connected to elaborate computing / data center infrastructures. A recent article Analytics, 2016 discussed no fewer than a dozen of these, which differ in their service offerings by either providing just computational resources, community aspects (algorithm DB, user experience sharing, communication), predictive services including pre-learned models or by focusing on specialized ML tasks (deep learning applied to videos etc.).

While all of those platforms may have their pros and cons, they all rely on *server-side* computation of the actual experiments, which is not surprising given the extensive availability of supporting software in that area (Hadoop, Spark, external backend cloud services like Tensor flow etc.). Graphinius on the other hand uses the server only as an access point to an algorithmic DB delivering snippets of code to the connecting JSVM. This is of great advantage to the delivering company, as it allows for almost effortless scaling, but hold the disadvantage of being dependent on the processing power of the connecting machine (mobile devices?). Experiments have yet to be conducted concerning the feasibility of such a platform; the author will be excited to setup such tests in the future.

5. Graphinius as a platform

This section contains an overview description of the proposed Graphinius platform and some of its core features envisioned, as well as basic graph-theoretical properties it will support.

5.1 General Properties

5.1.1 Online editor

Front-end web-developers know the great joys and advantages tools like *JSFiddle* or *CodePen* provide: Simply by navigating to their site or - more commonly - being referred to an 'experiment' via a link somebody has posted, one is presented with an online code editor supporting HTML5, CSS and JS on one side of the browser, and a live result window on the other. The code gets executed as soon as it has finished loading, and the live result is updated on every save (and sometimes every few seconds). In modern browsers those tools go as far as being able to preview and live-reload complex scenes written in WebGL, which was the author's basic inspiration to come up with a graph-theoretical counterpart.

5.1.2 Build & mutate

As a consequence, one should be able to go to the Graphinius website, be presented with a console that is pre-loaded with all background objects and functionality needed to build, interact with, mutate and visualize graphs. Upon making changes to the underlying graph structure in the Online Code Editor (which would follow the REPL principle: Read-evaluate-print-loop) the live-visualization will update immediately, so a CodePen-like workflow is offered to the user. This feature - although not yet implemented in a particular UI - is partly already available simply by using the debugging console every modern browser provides.

5.1.3 Save and fork experiments

Following the online web-developer coding example platforms mentioned above, Graphinius will enable users to change experiments, automatically forking them in the background. This way, every user can compile their own extendable library of exchangeable graph experiments over time.

5.1.4 Distributable via (mini) URL

Sharing ones insights with colleagues (investors, the public...) or preparing a live demonstration should be as easy as pressing a 'publish' button upon a mini URL would be generated by the platform. Then simply publish or email that URL to somebody and re-create your experiment on any device capable of handling HTML5 / WebGL through a modern browser anywhere!

5.1.5 Example graph datastructures

In order to help users get started with new experiments, we will provide a database of example graph structures, covering graphs from diverse fields of potential interest (traffic infrastructure graphs, protein interaction networks, sample social networks) as well as different graph classes (tree-shaped graphs, disconnected components, spherical graphs etc.).

5.1.6 Extendable algorithm DB

In the process of working on their experiments, users of the platform will undoubtedly come op with their own graph-theoretical algorithms. In order not to squander those pearls, Graphinius will provide a community-based, extendable algorithm DB which people can upload their algorithms to (with a description of necessary pre- and postconditions), so that other users can easily choose from a wealth of graph-theoretical computations.

5.2 Graph Properties

5.2.1 Mixed mode graph

A mixed mode graph is a graph that may contain directed as well as undirected edges at the same time. While many algorithms are defined on just undirected (e.g. Minimum spanning tree) or directed (e.g. percolation) edges exclusively, for many real world applications it is required to consider a combination of both - imagine traffic simulations with one-way streets or social networks in which people can be friends (undirected) and / or follow each other (directed). As Graphinius should be able to cover such applications, its core needs to be designed as a mixed-mode graph; the problem with this is that many algorithms have no standard implementation for a mix of both edge types, and so here and there it was necessary to come up with a logical and pragmatic solution, even if it could not be verified by any textbook.

5.2.2 Node and edge types (filters)

A mixed-mode graph alone however, is not enough for more complex scenarios. Assuming a social network again, we would first think of humans as participating entities. Depending on the particular use case of the network however, other nodes might be resources such as books, movies, or any type of commodity. Moreover, edges in such a network cannot only differ in mode of direction, but might represent a specific type as well (following someone vs. movie recommendation). From this emerges the need for graph filtering - or graph views - which expose only a specified subgraph to an executing algorithm, suitable to the particular situation. To stay with our example, if we need to find all users within three hops of friendship connections, we do not want to traverse all the edges representing recommendations (or messages, as there might be orders of magnitudes more of those present). In this case, we would execute a Breath-first-search against a view of the graph, which would present the BFS's logic with only the connections it is required to 'see'.

5.2.3 Object oriented

One design decision in writing any new (graph) library - as far as the author can judge from his personal research - lies in speed and memory vs usability. This concerns not so much the handling of nodes and edges themselves (many libraries have very good wrapper functions for dealing with basic primitives), but requirements like additional payload - e.g. the k-gram vector of a node representing a text document - or node & edge types themselves. In order to speed up execution of graph algorithms, advanced libraries use specialized data types like sparse matrices or fixed-length arrays; this on the other hand forces a programmer to hold additional data structures at hand for whenever more complex computations are needed. A good example for this would be computing the 'distance' of two nodes when defined not as the length of the shortest path between them, but as the cosine distance between their feature vectors. Taking into account the special language properties of JavaScript with its great emphasis on first-order functions and closures which makes it ideal for a natural callback-driven algorithmic approach (see Listings 7.2 and 7.3), the author believes that an object oriented approach realized in JavaScript idiomatic callback style is the suitable one for Graphinius, which is backed by 3 different properties of the language:

- Despite not being 'traditionally' object oriented like Java or C++ for its absence of constructs like classes etc., Javascript is firmly OO - in fact, everything except primitives is always an object, including and especially functions.
- Accessing objects instead of flat memory should not incur too much runtime overhead anymore, since modern JSVMs have abandoned the flat model in favor of an object memory model themselves.
- As Graphinius is intended to be part of a learning, teaching and research platform, and not designed to handle large graphs of many millions of nodes and edges (upwards), the OO approach seems a natural fit since it allows implementing algorithms in a very intuitive way - Meaning that the programmer can access all properties of graph objects directly via their methods instead of having to handle diverse, different datastructures (sparce matrices, lists, hashmaps) in coordination.

6. Software Requirements & Survey

In order to assess which technologies to use for a new project, one first has to take into account the kind of software product to build, the sector of the economy it will be used in as well as the specifics and constraints of the environment it's going to operate and interconnect in. Let us first take a closer look at those points:

- **The kind of product** to construct will often determine some core technologies: Building a messenger app requires real-time behavior some statistical product suite would never make use of. Likewise, an autonomous control system of a space probe will also depend on time-critical components, but in a different way than a messenger app, relying strongly on a constant rate of throughput, whereas a message flow will not be critically disturbed by a lag or latency disruption every now and then.

- **The industry sector** largely determines requirements in the form of compliance or industry certification. For instance, whereas the security concerns in a normal end-user centered application might be dealt with with relatively moderate levels of effort, applications employed in the financial or even medical sectors will in all probability have to satisfy additional security demands such as audit trail systems or compliance to specific data formats and standards.

- **The technical environment** a system is operated in will influence its shape and behavior as well. A relatively disconnected and isolated system like a statistical module (which e.g. outputs some results on a nightly basis) will be modeled differently than a web-based, cloud-oriented service incorporating many interfaces and API calls to dependent background or partner services.

In the following sections, we are not considering the entire SW development workflow from an economic / managerial point of view, but just the technological aspects of it. Let us first realize the differences between software development today and the way it was routinely conducted as short as 2 decades ago. The traditional

software development process has been relatively simple: upon setting a goal for functionality or any other measurable entity (code module, UI section), a continuous iterative process of writing some lines of code, re-compiling, testing (automated or manual) and bug fixing was all, or most, that was necessary to arrive at some usable product. Modern applications, however, especially web-based ones (and that includes all kinds of mobile apps that have seen their rise over last decade) operate on many different moving parts:

- Some server-side backend which coordinates incoming requests and provides consistency across business logic and database layers. This is probably the part with the greatest similarity to traditional, client-only or centralized software (development). I would also include old-fashioned web 'applications' (and certainly websites) in this environment, as a browser-based GUI alone without much processing or business logic going on, does not really fall into the category of a distributed application.

- A client part in the form of a modern in-browser based app (like GMail, Google Docs or Office365) or any mobile app executing on a contemporary mobile device.

- Some background-services, mostly in separated modules distributed over one or many servers worldwide, including interfaces to dependent services, isolated REST services (like a search portal for medical professionals inside a larger healthcare application) or microservices: sub-components of the business logic implemented directly on a database level, as implemented in the Foxx application micro-framework inside the multi-model database ArangoDB (Dohmen and Hackstein, 2014).

- Where needed, a visualization module will have to be provided which resides on the client side but is logically separated from the 'normal' business and communication logic of that module. In browsers, this can either be written in normal DOM code, SVG, Canvas, or WebGL (we don't want to mention earlier technologies that are fortunately falling from grace rapidly...).

In addition to this generic complexity, we have to deal with a different workflow cycle even on the level of individual developers: Whereas 15 years ago somebody

could set up some HTML files, include some JavaScript files, iteratively add new snippets of code and check the results by reloading the browser, even this small part of the development cycle has changed dramatically over the past 10 years - new Meta-languages like Typescript or Coffeescript on the language side, HTML-meta-markups like HAML, CSS preprocessors like Sass/Scss/Less as well as the integration of modern testing libraries makes a simple browser reload a technique of the past.

Those new methods provide great opportunities (but also challenges) even for the single programmer, which require a whole execution and deployment infrastructure, as depicted in Figure 6.1 and described in the following sections.

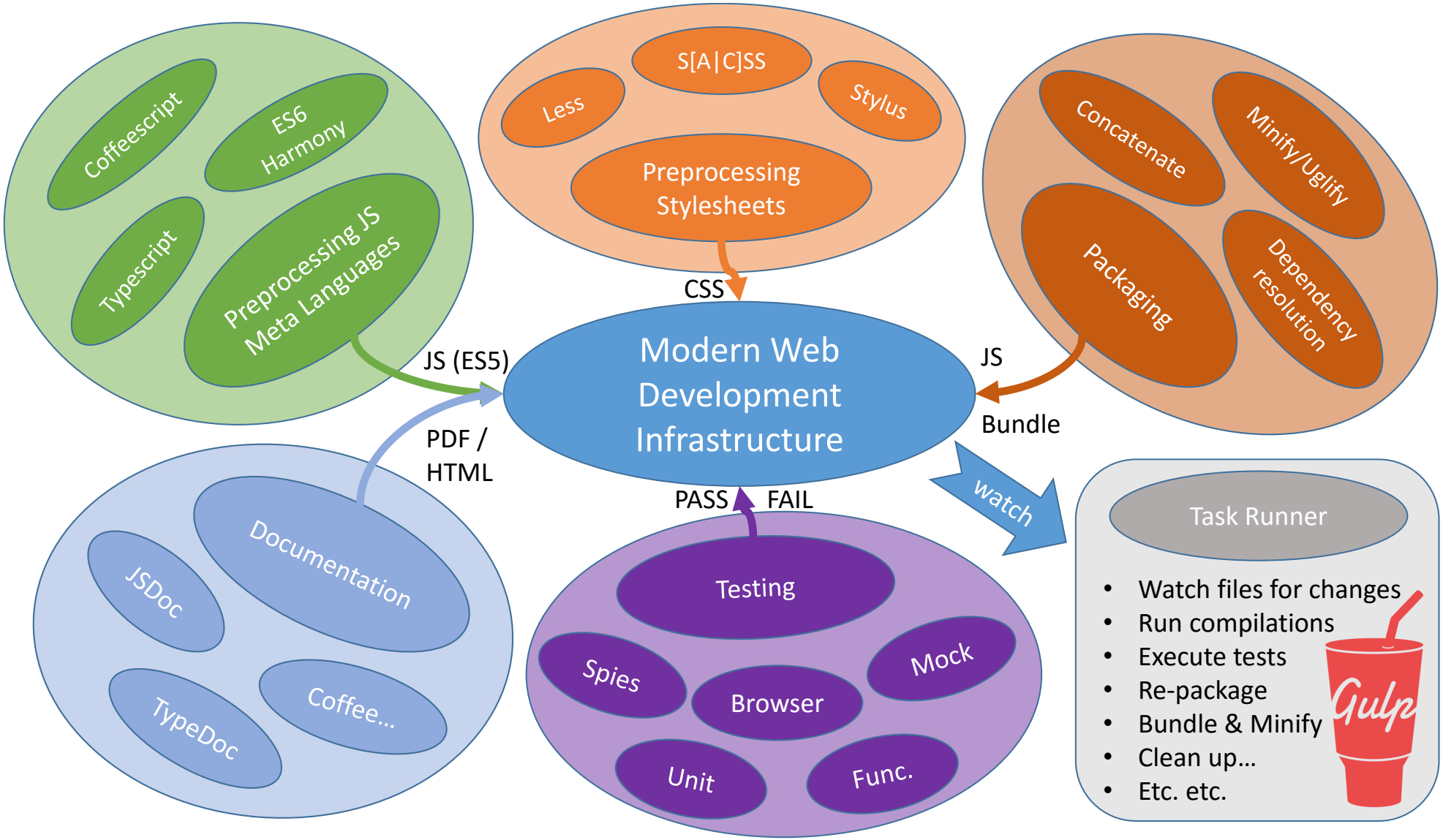


Figure 6.1: Modern Web Development Component Diagram

6.1 Preprocessing (compiling) JS Meta Languages

6.1.1 Javascript / ES6

Although the JavaScript language was put together by Brendan Eich of the Netscape company within 3 weeks in 1994 (when it was initially called LiveScript, and in 1995 renamed to JavaScript in a marketing attempt to jump on the Java-hype bandwagon), it turned out to be a nifty little language for general in-browser development and computations. JavaScript is a prototype based language, which means that it uses pointers to parent objects instead of class instantiations, features first-order functions (functions which can generate functions) and function-as-object passing, which is ideal for callback-based implementations of the visitor pattern, even more so as JS functions are automatically closures (functions or lambdas that have, regardless of their execution context, full access to their original definition scope).

After almost two decades however, the JS development community felt that requirements on modern web-based products had increased so drastically that traditional JavaScript could only partially serve them anymore. Problems are, amongst others: 1) the lack of an explicit type system (which is crucial for larger software projects), 2) the lack of an internal module system allowing for requiring other files or packages (except in the form of browser script tags), 3) the lack of a chaining system for async callbacks (which resulted in the notorious 'pyramid of doom'), 4) the somewhat peculiar functional scoping, which is mostly an entrance barrier to developers of other languages, as well as 5) a general lack of elegant language constructs (like deconstruction of objects into variables etc.).

While some of those problems have been addressed even in the context of ECMAScript 5 (like the introduction of promises to replace nested callback functions), many shortcomings could only be addressed by external libraries, which polluted the workflow with additional dependencies not needed in more sophisticated languages and often increased the JS download size by hundreds of kilobytes, which routinely becomes a problem on mobile devices.

ECMAScript 6 (codename harmony) is an overhaul of the JavaScript language featuring classes, a new keyword for the familiar block scoping (*let*), as well as an integrated module system allowing to require external files even inside the browser.

The greatest obstacle to using ES6 today is the lack of complete support across all browser vendors - this is where ES6-to-ES5 compilers like *Traceur* or the more popular *Babel* come in. As far as syntax goes, ES 6 cleans up some keyword usages in order to make code more readable.

```
1 odds  = evens.map(function (v) { return v + 1; });
2 pairs = evens.map(function (v) { return { even: v, odd: v +
      1 }; });
3 nums  = evens.map(function (v, i) { return v + i; });
```

Listing 6.1: ECMAScript 5 (usually referred to as 'JavaScript') version of functional programming using the natively built-in mapping function.

```
1 odds  = evens.map(v => v + 1)
2 pairs = evens.map(v => ({ even: v, odd: v + 1 }))
3 nums  = evens.map((v, i) => v + i)
```

Listing 6.2: ECMAScript 6 equivalent to the above code.

(Examples taken from (Engelschall, 2016))

Although ES6 is a great improvement over ES5 in many respects, it still lacks an explicit type system and interfaces (which can guide an IDE in it's analysis regarding Code Completion and IntelliSense). It was therefore not considered the best option for the development of a potentially large library as GraphiniusJS.

6.1.2 Coffeescript (CS)

Coffeescript was an attempt to make JavaScript code more readable as well as writable. It was apparently inspired by the clean syntax used in modern scripting languages like Ruby or Python and adopted the use of whitespace as control characters like Python (but not Ruby). Most of the language was based on using 'syntactic sugar' to abbreviate otherwise verbose JS code. For instance, the *this* variable was replaced by the @ sign, the return statement at the end of a function became superfluous, and the lambda operator -> was introduced as a shortcut for the *function* keyword in normal JS.

```
1 [1..10].map (i) -> i*2
2 i * 2 for i in [1..10]
```

Listing 6.3: Two versions of the same mapping functionality in CoffeeScript

(Example taken from CoffeeScript)

Like ES6, Coffeescript is compiled down to ES5 through it's own coffee compiler. As much as the idea of CS is neat and very justified for individual developers, the use of whitespace as control characters can add some additional hassles if working in a team; only slight deviations in the individual setup can cause serious problems, like the use of editors that use different representations for tabs (tab vs. 2 spaces vs. 4 spaces) or different line ending symbols (Windows vs. Mac vs. Linux). Furthermore, CS did not resolve the lack of an internal module system or the lack of an explicit type system, and was therefore not considered for Graphinius JS.

6.1.3 Typescript (TS)

The greatest obstacle to large-scale web-development in ES5 was the lack of an explicit type system and built-in module system, which would allow an IDE to scan dependent files and constructs and infer a valid type-flow, enabling the development environment to assist the programmer with code completion and type error hints. This was realized by Microsoft when they ported part of their Office suite to the web (Office365), which consumed not only years of manpower but took hundreds of thousands of lines of code before a stable product could be released. As a consequence, they formed a group around their lead C# / .NET developer Anders Hejlsberg (the former creator of TurboPascal and chief software engineer at Borland until 1996) with the mission to develop a JS meta language incorporating those characteristics for large scale web development in 2012. The result of those efforts is Typescript, which of 2016 is a superset of ES6 with the additional benefits of allowing for the declaration of interfaces, enums, and in-place type annotations (amongst many others features like default and optional function parameters):

```
1 import * as $N from "./Nodes";
2
3 export interface IConnectedNodes {
4     a: $N.IBaseNode;
```

```
 5     b: $N.IBaseNode;
 6 }
 7
 8 class BaseEdge implements IBaseEdge {
 9     protected _directed     : boolean;
10     protected _weighted     : boolean;
11     protected _weight       : number;
12     protected _label        : string;
13
14     constructor (protected _id,
15         protected _node_a:$N.IBaseNode,
16         protected _node_b:$N.IBaseNode,
17         options?: EdgeConstructorOptions) {...}
18 }
```

Listing 6.4: Typescript sample featuring import of an external module.

..., an exported interface definition, type annotations, instance variable setting via constructor specifiers (protected) as well as optional parameters. (Example taken from the Graphinius JS Edge Class.)

Furthermore, the Typescript compiler can output TS code to ES6, ES5 or even ES3, and already incorporates some features currently only found in the ES7 draft specification (which will take several years to become the next standard). With all the benefits offered by ES6, no additional compile step (over ES6 with traceur/babel) and additional typings support (which comes in the form of separate typing files that even exist for most external libraries like underscore, lodash or jquery), Typescript made an ideal candidate as the base language for Graphinius JS.

6.2 CSS preprocessing

Just like the requirements on JavaScript underwent a steady evolution as project sizes went from very small-scale Webpages to full-blown in-browser application suites, so did the demands on Cascading Style Sheets (CSS) increase over time. The original CSS specification did not offer any possibilities of abstraction or code-reuse, for example. There were no modules that could be included and shared, no parameters one could pass to

instructions (like the pixel width a border-radius should exhibit) nor any variables one could set as defaults (for the purpose of color-scheme definitions, for instance). Several projects have sprung up in order to amend that situation. Let's briefly take a look at 2 options, namely SCSS (SASS, which is just SCSS without braces and semicolons) and LESS, which runs the pre-processing step on the client rather than the server:

```
1 $font-stack:    Helvetica, sans-serif;
2 $primary-color: #333;
3 @mixin border-radius($radius) {
4     -webkit-border-radius: $radius;
5     -moz-border-radius: $radius;
6     -ms-border-radius: $radius;
7     border-radius: $radius;
8 }
9 body {
10     font: 100% $font-stack;
11     color: $primary-color;
12 }
13 .box { @include border-radius(10px); }
```

Listing 6.5: SCSS example demonstrating the use of variables and mixings

(Example taken from (Hampton, Natalie, and Eppstein., 2016))

```
1 @base: #f938ab;
2 .box-shadow(@style, @c) when (iscolor(@c)) {
3 -webkit-box-shadow: @style @c;
4     box-shadow:         @style @c;
5 }
6 .box-shadow(@style, @alpha: 50\%) when (isnumber(@alpha)) {
7         .box-shadow(@style, rgba(0, 0, 0, @alpha));
8 }
```

Listing 6.6: LESS example demonstrating the use of variables and default parameters

(Example taken from (Page and Mikhailov, 2016))

While CSS preprocessors are important in every contemporary web project, the work described in this Master Thesis is focused on the underlying graph library, which features no (built-in) graphical component and therefore did not require any such module.

6.3 Testing

With the increase in size and complexity of JavaScript codebases, testing became essential in the modern web development cycle. While there are several other JS (unit) test frameworks available (like JSUnit, QUnit, YUI-Test), we will mention only two contemporary libraries in detail, which both allow for behavior driven development (BDD) style testing as well as *mocking*, *stubbing*, and *spying*. Both have very similar syntax and are intermixable with different assertion libraries; it seems to the author that a choice between those libraries is more a question of taste than necessity, and for GraphiniusJS, a combination of Mocha with Chai has been chosen.

6.3.1 Jasmine

Jasmine (Hahn, 2013) is modeled after the Ruby RSpec Gem and refers to its tests as *specs*. It allows for nesting of *describe* blocks in order to distinguish different levels of test suites, follows the *should* approach by providing *it* functions taking a description of a test as well as it's test body in the form of a callback. An assertion library is built right into Jasmine and uses the *expect* style of writing assertions. Let's have a look at a small example:

```
1 describe('calculator', function() {
2     describe('add()', function() {
3         it('should add 2 numbers togoether', function() {
4             expect(calculator.add(1, 4)).toEqual(5);
5         });
6     });
7 });
8 // now for spying...
9 var userSaveSpy = spyOn(User.prototype, 'save');
10 // and stubbing...
11 spyOn(user, 'isValid').andReturns(true);
```

Listing 6.7: Jasmine example of a nested test suite containing one simple assertion in expect style as well as a spy and a stub

(Example taken from (Tang, 2015))

6.3.2 Mocha / Chai

The Mocha test runner basically provides the same functionality as Jasmine in that it provides *describe* blocks, should-style *it* functions and several ways to output reports (spec-style, list, dots, the nyan cat...) as well as handle pending tests (via not handing a callback to the *it* function) or skipping tests (via *it.skip();*). The greatest difference to Jasmine is the fact that Mocha does not ship with a built-in assertion library. There are several compatible alternatives, like *should.js*, *expect.js* or *unexpected*. The most popular and widely used, however, clearly seems to be *Chai*, which provides the *should*, *assert* as well as *expect* styles of writing assertions. In addition to that, we also need a library called sinon.js that allows for spying and stubbing. We will be seeing some in-depth examples of Mocha, Chai (expect style) and Sinon in the Chapter 7, so let's limit ourselves to the same use case as above:

```
describe('calculator', function() {
    describe('add()', function() {
        it('should add 2 numbers togoether', function() {
            // expect style
            expect(calculator.add(1, 4)).to.equal(5);
            // should style
            calculator.add(1, 4).should.equal(5);
            // assert style
            assert.equal(calculator.add(1, 4), 5);
        });
    });
});
// now for spying...
sinon.spy(user, 'isValid');
// and stubbing...
sinon.stub(user, 'isValid').returns(true);
```

Listing 6.8: Mocha example of a nested test suite containing one simple assertion in expect style as well as a spy and a stub

(Example taken from (Tang, 2015))

6.3.3 Cucumber

Cucumber describes itself as 'An open-source tool for executable specifications', which means it wants to be understood less as a testing tool but rather as a group communication tool. The idea behind cucumber is the 'single source of truth' concept, which means that both the specifications handed down to the developers by their customers or management as well as the technical test definitions that execute test suites and determine the project's progress from the programmers' viewpoint, should be one and the same document.

The first and foremost implication of this approach is that the specifications must be formulated in plain English (or any other common-language), so that collaborators from outside the immediate project team can read, but ideally also write and modify them. It is then the job of developers to transfer those common-language statements via so-called *step definitions* into executable test code. In invoking this test code, the cucumber test runner receives feedback on the internal state of test case completion, which is translated back up to the semantic common-language level and displayed as progress of the original specification. This way, both programmers as well as customers / managers can monitor the progress of the project, and tests become both functional as well as acceptance tests (with unit tests usually not being expressed via cucumber).

A cucumber feature file is written using the so-called Gherkin syntax (a business readable, domain specific language) consisting of Feature and Scenario definitions. Features are general statements of intent describing the business value of a certain feature from the perspective of a potential user. Scenarios then describe different aspects of that feature in the form of usage scenarios the user might find herself in.

```
1
2     Feature: Example feature
3         As a user of Cucumber.js
4         I want to have documentation on Cucumber
5         So that I can concentrate on building awesome
              applications
```

```
6
7         Scenario: Reading documentation
8             Given I am on the Cucumber.js GitHub repository
9             When I go to the README file
10            Then I should see "Usage" as the page title
```

Listing 6.9: Cucumber example describing a Feature containing a simple Scenario

The step definition file implementing the first of the three lines in the scenario could be implemented as follows:

```
1 module.exports = function () {
2     this.Given(/^I am on the Cucumber.js GitHub repository$
          /, function (callback) {
3         this.visit('https://github.com/cucumber/cucumber-js
              ', callback);
4     });
5 };
```

Listing 6.10: Cucumber example describing a Feature containing a simple Scenario

(Examples taken from (Biezemans, 2016))

Cucumber can be utilized from many different languages and has been used by the author as far back as 2009. In the case of Graphinius it will certainly be used once the platform module (browser UI) begins to take shape, as cucumber is very well suited to be used in combination with the browser DOM. For the scope of the underlying Graphinius JS library however, there was no reason to employ such a tool as collaboration with end-users on that level seems neither necessary nor sensible.

6.4 Automatic Documentation

Writing manual documentation on a software library is absolutely necessary if one desires their product to be used by other developers; writing documentation files separately from one's code files however is a burden not many programmers want to accept. This is were automated documentation generators come in - they scan the source code of a project for

comments above a function or class definition and auto-generate the required documents in a convenient output format like HTML or PDF.

6.4.1 JSDoc & alternatives

In the realm of JavaScript there are several candidates for pure, JS-based auto-doc generation; in a review conducted by (Das Modak, 2016), the four libraries JSDoc, DOCCO, doxx and ui were examined for several properties, the most important of which were:

- Does it support structured syntax (like Javadoc)?
- Can the appearance be customized (via CSS or CSS frameworks like Bootstrap)?
- Does the output contain a search feature?
- Does it parse the entire source code or only the comment blocks?

The interested reader may be referred to the resource mentioned above; as we are using Typescript for Graphinius JS, we do not want to limit ourselves to capturing just the information present in the output JS, but also the type information as well as interface specifications only found in the original TS sources.

6.4.2 TypeDoc

Currently, there is only one library available for automatically scanning through Typescript sources and generating an appropriate documentation - TypeDoc. It extracts all type annotations found in interfaces or method specifications and scans Javadoc-like comment blocks. Its output is HTML5 with a convenient search function and customizable appearance as well as a practical legend for different types of entities (class, interface, function etc.); as of yet, it neither supports simple double-slash comments nor output to PDF.

A very nice feature to mention comes to bear in combination with a github project - in this case Typedoc is capable of directly linking an item's documentation to the appropriate place in the github repository, which makes switching from documentation to code a convenient experience. As the whole documentation for Graphinius JS exists in the form of extracted Typedoc and is readily available to the reader in Appendix B, we can forgo the display of a sample at this point.

6.5 Build system for browsers / packaging

Within the server-side JavaScript version of NodeJS (which uses Google's V8 under the hood), one can easily require other files as modules through the CommonJS module specification. This of course is a superior way of building JS applications, as modules are thereby testable in isolation, dependencies can be required without adding some script tag into an HTML file, and dependency resolution is done via the built-in module framework as opposed to having to take care of including script tags in the right order (or otherwise they might overwrite each others' variables within the global namespace, which is a frequent source of attrition amongst web developers).

Unfortunately, there is no native way to require JavaScript files or parts thereof from another JS file in the browser. With the advent of ECMAScript 6 this shortcoming will finally be corrected; however due to the fact that most browsers do not fully support that standard yet, and given the benefits of a 'normal' SW-development process, it has become commonplace to develop one's webapp in NodeJS (including all necessary libraries from the Node Package Manager's repositories), and then utilizing an external tool to package the software for its final release. Those tools have then to collect all the source files including dependencies, resolve any potential conflicts in the dependency graph, wrap each file's content with a require mechanism that works in the browser, and finally concatenate (and maybe minify / uglify) the result in order to save space.

Although there are a few more candidates (RequireJS, SystemJS, JSPM) that could be mentioned in this section, we are only going to take a look at two libraries which characterize the differences in this field pretty well:

6.5.1 Browserify

Browserify is easily the most popular amongst the various packaging libraries available, and probably also the most powerful in its ability of dependency checking and conflict resolution. Throughout the first months of developing GraphiniusJS, the author used browserify to conveniently package all of the code files into a bundle readable by the browser. The library is especially handy since it can deal with NodeJS specific dependencies like the Filesystem module, which provide no additional use in the browser but can be hard to extract out manually from the packaging process. In addition it resolves circular dependencies on its own, which is especially practical for less experienced programmers. The great disadvantage or browserify is the file size of its output: in order to offer all

the benefits it provides, it generates in excess of 30k lines of boilerplate code for even the tiniest of JavaScript programs. This results in a file size easily in excess of 1MB, which renders browserify practically unusable for mobile applications.

6.5.2 Webpack

Webpack, in many respects, is the opposite of Browserify. It does not offer a no-hassle, work-the-first-time experience for today's stressed web developer, requires some (often much) configuration of loaders for different file types, easily capitulates in the event of circular dependencies (and does not even fire an error or warning) and has serious problems including all kinds of NodeJS (server-side) libraries. Despite the greater configuration effort and necessary developer's care however, Webpack's require wrapper around JavaScript files and functions is microscopic compared to browserify. This way, the author was capable of packaging all of Graphinius JS (as of this writing) into 67.8 kilobytes of uncompressed and 25 kilobytes of minified JavaScript, with further down-potential in using a better compression library like Google's Closure compiler in the future (which requires its own delicate handling...).

6.6 Task Runner

Finally, all of the above components have to work in perfect coordination in order to make the web development experience convenient and leave the programmer to their most important job: focusing on the problems to solve rather than manually operating different libraries in the various preparation steps. Therefore, task runners (the JavaScript vocabulary's version of Makefiles) have been invented, which offer several standard tasks to be executed and countless more to be added via their respective plugin systems. The main duty of a task runner is to watch local files for changes (happening every time they are saved to disk) and execute different, predetermined tasks in a given order. In the context of Graphinius JS, this development cycle consisted of the following phases (only 1 and 2 are executed on every save, the rest required a separate 'make' call):

1. **Build:** Compile TypeScript to JavaScript.

2. **Test:** Run all synchronous Mocha tests; for performance reasons, a few asynchronous tests like remote JSON graph structure loading were not periodically executed.

3. **Generate Typedoc** and output the resulting HTML to a subfolder.

4. **Assemble JS output** files and write them to the dist directory for requiring from a NodeJS console.

5. **Package & bundle (compress)** the dist folder's content using webpack and output a minified JavaScript file consumable by any standard web browser.

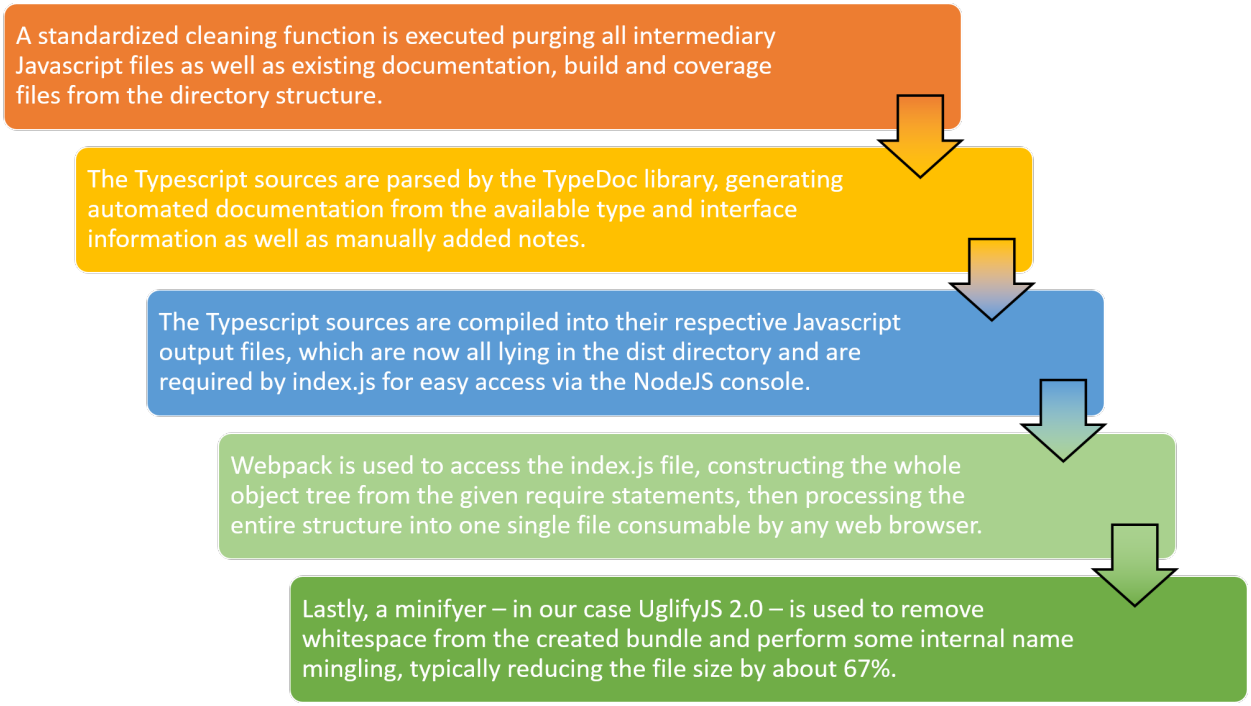


Figure 6.2: The GraphiniusJS Bundling process

In addition to those standard tasks, a library called *Istanbul* was used to capture the progressing test coverage.

In order to execute those tasks, the following two task runners were considered:

6.6.1 Grunt

The Grunt JS task runner has been around since early 2012 and follows the principle of *Configuration over code*. This means that instead of writing executable code, the library takes a configuration file in the form of a *Gruntfile.js*. Although this file is of generic JavaScript format, no real computations or control flows take place inside it; instead, Grunt takes instructions in the form of JSON object blocks, which makes writing a Gruntfile easy and intuitive, but severely limits the developer's possibilities to fine-tune desired functionalities.

6.6.2 Gulp

Gulp JS, as a slightly more recent contender (the first commit logged on github stems from July 2013) represents the opposite approach to Grunt JS - it is based on the idea of *Code over configuration*. This implies that instead of writing an extensive configuration file, a *Gulpfile.js* is written like a normal JavaScript program, making extensive use of the NodeJS-native streaming capabilities to pass results of an earlier processing step on to a later (much like the Unix kernel pipes can be used). This gives developers greater power over their build process, enabling them to use normal language constructs like branches and conditional assignments.

```
// Project configuration.
grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    watch: {
        typescript: {
            files: ['**/*.ts'],
            tasks: ['compileTS']
        },
        tests: {
            files: ['**/*.js'],
            tasks: ['mocha']
        }
    },
    uglify: {
        build: {
            src: 'dist/JSAlgorithms.js',
            dest: 'dist/JSAlgorithms.min.js'
        }
    },
    shell: {
        mocha: {
            options: {
                stdout: true,
                stderr: true
            },
            command: 'mocha'
        },
        compileTS: {
            options: {
                stdout: true,
                stderr: true
            },
            command: 'tsc **/*.ts'
        },
        copyMin: {
            options: {
                stdout: true,
                stderr: true
            },
            command: 'cp dist/JSAlgorithms.js /var/www/html/g
        }
    }
});
grunt.registerTask('mocha', 'shell:mocha');
grunt.registerTask('compileTS', 'shell:compileTS');
grunt.registerTask('build', ['concat', 'uglify', 'shell:copyM
```

```
// Compile Typescript to Javascript
gulp.task('build', ['clean'], function () {
  return gulp.src(paths.typescripts, {base: "."})
             .pipe(ts(tsProject))
             .pipe(gulp.dest('.'));
});

// Generate Documentation (type doc)
gulp.task("tdoc", ['clean'], function() {
  return gulp
    .src(paths.typesources)
    .pipe(tdoc({
      module: "commonjs",
      target: "es5",
      out: "docs/",
      name: "Graphinius"//,
      // theme: "minimal"
    }));
});

// Packaging - Node / Commonjs
gulp.task('dist', ['tdoc'], function () {
  var tsResult = gulp.src(paths.distsources)
                     .pipe(ts(tsProject));
  return merge([
    tsResult.dts
                .pipe(gulp.dest('.')),
    tsResult.js.pipe(gulp.dest('./dist/'))
  ]);
});

// Packaging - Webpack
gulp.task('pack', ['dist'], function() {
  return gulp.src('./index.js')
    .pipe(webpack( require('./webpack.config.js') ))
    .pipe(gulp.dest('build/'));
});

// Uglification...
gulp.task('bundle', ['pack'], function() {
  return gulp.src('build/graphinius.js')
    .pipe(uglify())
    .pipe(rename('graphinius.min.js'))
    .pipe(gulp.dest('build'));
});
```

Figure 6.3: Comparison between Grunt & Gulp build systems

6.7 Overview of technology choices

In summary, the chosen technologies / libraries for developing Graphinius JS came to be:

- **JS Preprocessor:** Typescript
- **CSS Preprocessor:** N/A
- **Testing libraries:** Mocha + Chai (expect style) + Sinon
- **Doc generator:** Typedoc
- **Packaging:** Webpack
- **Task Runner:** Gulp

7. Architecture / Implementation

The Graphinius Platform consists of four main components as depicted in the following diagram. Of those 4, the practical parts of this Master thesis mainly consider Graphinius Base, with all graph construction and analysis functionality residing in GraphiniusJS.

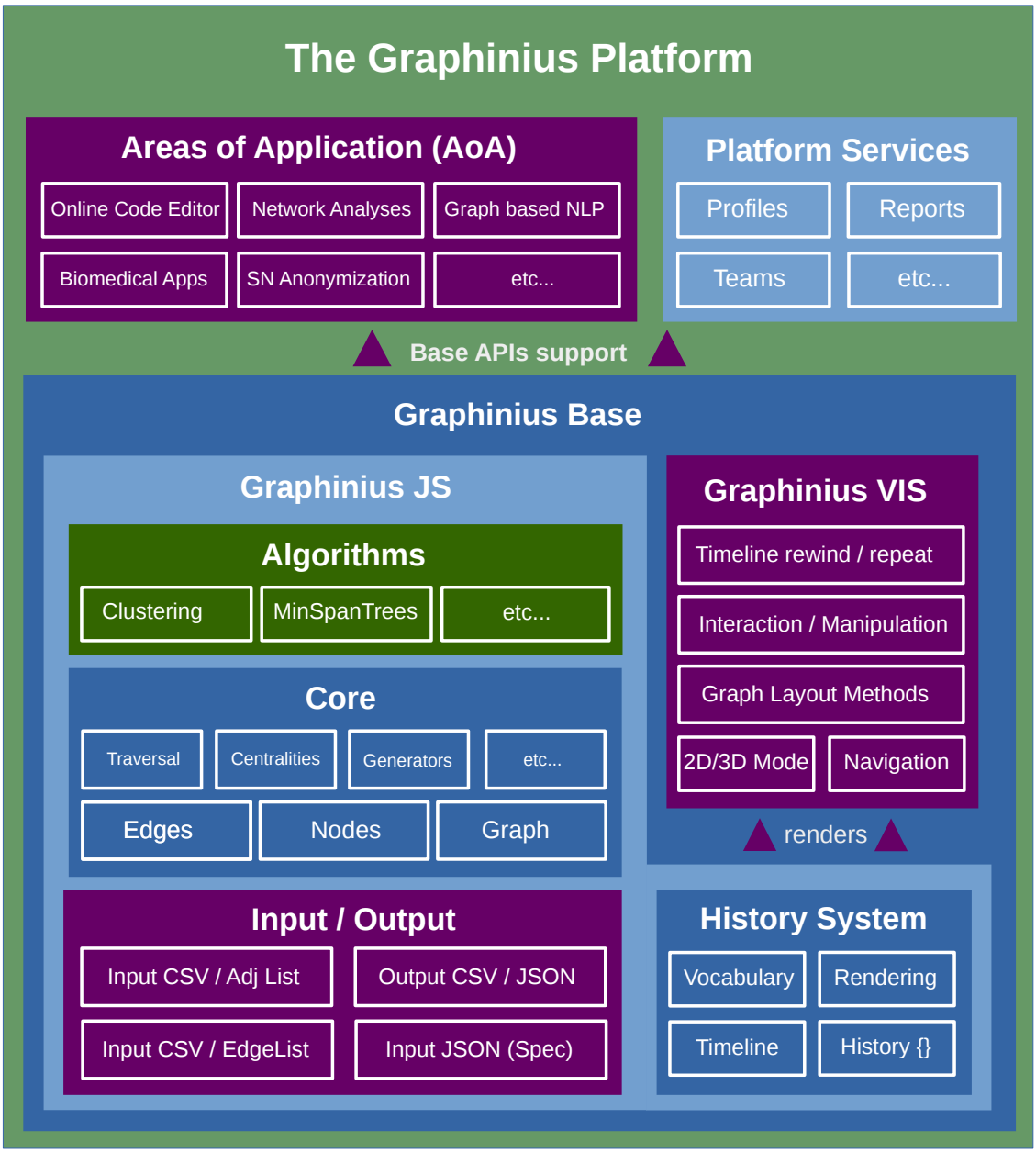


Figure 7.1: Graphinius platform architecture overview

7.1 Graphinius Base

As the name suggests, the Base module offers all the functionality necessary to develop graph-based applications on top of it, including basic graph-computations and algorithms as well as real-time, in-browser visualization. It is therefore logically composed of the two main modules *GraphiniusJS* and *GraphiniusVIS* as well as a mechanism of communication between those two, the *History System*.

7.2 Graphinius JS

Graphinius JS deals with graph loading via Input Readers for CSV and JSON file formats, instantiation as well as mutation of graphs through the graph, nodes and edge classes as well as some basic graph algorithms, including degree distributions, edge generators and traversal via breath-first, depth-first and best-first (priority-first) search algorithms. As not all components depicted in Figure 7.2 are of equal importance (or implemented yet), the following substructure does not fully comply with the organization of that diagram.

7.2.1 Edges

Edges are the most basic class within the GraphiniusJS library. They depend on the Nodes class solely in order to be able to check if their endpoints consist of valid graph nodes. Edges can never instantiate nodes themselves, so in order to add a valid edge to a graph, the connected nodes must already exist in the structure. Edges can be directed or undirected, weighted or unweighted, hold an ID and can be given a label. Apart from that, they contain no other internal logic.

7.2.2 Nodes

Nodes are already a lot more complex in Graphinius JS. They consist of an ID, an optional label, and have datastructures to hold the directed as well as undirected edges connected to them; When adding and removing connected edges they measure and update their own degrees (according to the mixed-graph nature of Graphinius, we always differentiate between in-, out-, and undirected degree) and are the lowest-level objects used in graph traversal, as they contain the necessary functionality to return their neighboring nodes (according to outgoing, incoming, or undirected edges).

7.2.3 Graph

The Graph class is the main component at the core of Graphinius JS as it contains all necessary functionality to create as well as add and remove nodes and edges to a graph. It keeps track of its size in number of nodes and edges, holds datastructures to look up those objects as well (as always it differentiates between directed and undirected edges), can return a random node or edge, clean nodes from all incoming, outgoing or undirected edges and contains helpers to print statistics or the degree distribution over its nodes.

Moreover, it contains the GraphMode setting which has two different meanings: As a metric of the graph, it tells a caller if the graph contains no edges at all (GraphMode.INIT), holds only undirected (GraphMode.UNDIRECTED) or directed (GraphMode.DIRECTED) edges, or a mixture of both (GraphMode.MIXED). As a setting for graph traversal, it enables *graph views* by allowing algorithms operating on it to only see a subset of all the existing edges in the graph.

```
1 if ( dir_mode === $G.GraphMode.MIXED ) {
2     bfsScope.adj_nodes = bfsScope.current.adjNodes();
3 }
4 else if ( dir_mode === $G.GraphMode.UNDIRECTED ) {
5     bfsScope.adj_nodes = bfsScope.current.connNodes();
6 }
7 else if ( dir_mode === $G.GraphMode.DIRECTED ) {
8     bfsScope.adj_nodes = bfsScope.current.nextNodes();
9 }
10 else {
11     bfsScope.adj_nodes = [];
12 }
```

Listing 7.1: Graph traveral dependent on GraphMode.

7.2.4 Edge Generators

There are many graph generators found in more established graph libraries and even specialized generator-only components (like (Bader and Madduri, 2006)), which can be used to generate random graphs or graphs following a specific layout (trees, circles, spheres, arc diagrams etc.). For the emerging GraphiniusJS library, complete graph generators

were not yet considered, but edge generators implemented. This allows us to take generic point cloud data (coming from diverse sources like tabular datasets, range images etc.) and convert them into a graph structure by adding random connections according to two different approaches:

- **Per probability.** This algorithm goes through all node combinations in the graph and adds an edge between them with a certain probability. By its very nature, the algorithm runs in quadratic runtime in the number of nodes.
- **Per node.** Considering every node in sequence and adds a certain number of edges between that node and another, randomly chosen one. This algorithm could take quadratic runtime in the number of nodes (if the random node algorithm chooses so poorly that it tries to add the same edge over and over again) but in practice should perform considerably faster.

7.2.5 Degree distribution

Probably the easiest algorithm performable on a graph is to compute the degree distribution over its nodes. It simply walks over the internal node structure, counting all the ingoing, outgoing, and undirected edges returning some (numerical) histogram. The following figure depicts how to obtain a degree distribution in GraphiniusJS via a browser debugging console.

```
> graph.getStats()
< Object {mode: 1, nr_nodes: 13859, nr_und_edges: 0, nr_dir_edges: 41541}
> graph.degreeDistribution()
< ▼ Object {in: Uint16Array[16], out: Uint16Array[16], dir: Uint16Array[16]
  , und: Uint16Array[16], all: Uint16Array[16]} i
    ▶ all: Uint16Array[16]
    ▶ dir: Uint16Array[16]
    ▶ in: Uint16Array[16]
    ▶ out: Uint16Array[16]
    ▶ und: Uint16Array[16]
    ▶ __proto__: Object
> graph.degreeDistribution().all
< [0, 0, 0, 86, 1221, 3899, 4304, 2760, 1156, 330, 76, 14, 6, 2, 4, 1]
```

Figure 7.2: Degree distribution of a graph with 13859 nodes and 41541 edges

7.2.6 Graph Traversal

Graph traversal is the general term for exploring a graph from a given start node (also called 'root' node in this context). There are three basic forms of traversal differentiating by the order by which new nodes are visited. In GraphiniusJS, graph traversal is implemented using the visitor pattern by defining callback functions for specific 'hooks' in the traversal procedure. The graph traversal algorithm is thus reduced to only running its particular node-expansion scheme on the data structures necessary for its purpose; the actual information 'extraction' and result 'compilation' is then done by executing the injected callbacks at the aforementioned, pre-determined hooks. Let's take a look at an example callback initializing the result set of a Breadth first search algorithm at the start of it's invocation and pushing that callback onto the *init_bfs* callback array inside the algorithm's config.callbacks object:

```
1 // Standard INIT callback
2 var initBFS = function( context : BFS_Scope ) {
3 // initialize all nodes to infinite distance
4 for ( var key in context.nodes ) {
5     config.result[key] = {
6         distance : Number.POSITIVE_INFINITY,
7         parent   : null,
8         counter  : -1
9     };
10 }
11 callbacks.init_bfs.push( initBFS );
```

Listing 7.2: Defining a callback function for traversal.

During the actual BFS run, this callback is then executed as follows:

```
1 // Execute INIT callback
2 if ( callbacks.init_bfs ) {
3     $CB.execCallbacks(callbacks.init_bfs, bfsScope);
4 }
```

Listing 7.3: Executing a callback function during traversal.

7.2.6.1 Breadth first search

BFS traverses a graph by expanding one 'shell' of nodes after the other, which means that it progresses outwards from a starting node like the layers of an onion until all reachable nodes have been visited. It is primarily used for getting a sense of distances in a graph (although it does not compute shortest paths by itself) and uses a queue datastructure to add nodes it's list of future visitations.

In GraphiniusJS, BFS makes use of the following hooks: 1) **init_bfs** at the start of the algorithm, 2) **node_unmarked** when an encountered node has not yet been marked as visited, 3) **node_marked** in case such a node has already been visited, and 4) **sort_nodes** in the case the user wants to imply a certain order by which to add nodes to the queue - this does not change the nature of BFS, however.

7.2.6.2 Depth first search

DFS explores new nodes in a recursive fashion, which makes it advance 'deep' into the far reaches of the graph structure before exploring the immediate vicinity of the start node. For this purpose it uses a stack as its basic datastructure; moreover, if no more nodes are reachable from a given departure node, but the graph structure has not been entirely explored yet, DFS (via an outer loop) will randomly choose one of the remaining nodes, thereby arriving at a (random) segmentation of the entire graph.

In GraphiniusJS, DFS provides the following hooks: 1) **init_dfs** at the start of the outer loop, 2) **init_dfs_visit** at the start of a 'visit' (the inner loop), 3) **node_popped** for any callbacks to execute directly after a node has been taken from the stack, 4) **node_marked** which behaves as in BFS, 5) **node_unmarked** which behaves as in BFS, 6) **sort_nodes** which behaves as in BFS, and 7) **adj_nodes_pushed** which executes directly after a node has been expanded and its neighbors pushed to the stack.

7.2.6.3 Best (priority) first search

PFS always picks from its available nodes the one that evaluates to an optimal value (e.g. the minimal distance given an edge weight) - it therefore uses a (MIN / MAX) heap as its underlying datastructure, which in GraphiniusJS has been implemented with injectable heuristics (via callback functions as described in the Graph Traversal 7.2.6 section).

PFS lets the caller inject the following callbacks: 1) **init_pfs** is invoked at the start of the algorithm, 2) **node_open** in case an encountered node is already contained in

the OPEN set, 3) **node_closed** in case an encountered node is already contained in the CLOSED set, 4) **better_path** in case a more optimal path to a node under test has been discovered, and 5) **goal_reached** if we have encountered a specified end goal causing the algorithm to immediately return.

7.2.7 Traversal-based algorithms

Although as of the time of this writing, no more algorithms have been implemented in Graphinius JS, most of the commonly used techniques are building upon one or the other form of basic graph traversal. This includes shortest paths, cycle testing, topological sorting of nodes, strongly connected component analysis, minimum spanning trees, some centralities (closeness and betweenness for instance) as well as many more than the author (or reader) will be able to recollect.

Building all traversal-based algorithms on top of either BFS, DFS or PFS utilizing their callback & hook structure will enable us to easily add new graph-views defined on node or edge types (and other criteria) to the library in the future, without having to re-adapt each and every single component to the respective improvement.

7.2.8 Input / Output

There are two basic input readers implemented in GraphiniusJS:

The CSV Reader, which takes adjacency lists or edge lists in CSV format and supports the most simple, but widely used, formats as well a a few additional options, and **The JSON Reader**, which operates on a custom, Graphinius-related JSON file format in order to support additional features specific to the use cases of the platform.

As of the time of this writing, standard output filters have not been implemented, but would follow the same principles outlined below for their input counterparts.

7.2.8.1 CSV Reader

The CSV Reader class supports to widely used graph representation formats, namely *adjacency lists* and *edge lists*. An adjacency list is usually composed of lines indicating a StartNode at position one, followed by a series of connected EndNodes at the following positions in the line, where the connections can be interpreted as either directed or undirected edges:

```
1          A, B, C, A, D
2          B, A
3          C, A
4          D, A
```

Listing 7.4: Sample Adjacency list, no edge direction.

In addition to that, the Graphinius CSV Reader can be configured to consider explicitly defined edge directions, as in the following listing, or instructed to interpret all edges as either directed or undirected regardless of the direction specified in a file.

```
1          A, B, u, C, u, A, d, B, d, D, d
2          B, A, u
3          C, A, u, A, d
4          D, A, d
```

Listing 7.5: Sample Adjacency list including edge direction.

CSV Edge Lists work analogously but are even simpler and use the format *(StartNode, EndNode [,directed])*.

7.2.8.2 JSON reader

The Graphinius JSON reader is a more complex class as it uses it's own data format specific to the use cases targeted by the platform. It uses a nested object structure defining an array of node objects potentially containing different arrays of sub-objects:

An **edge array** containing objects specifying a *to* node, a *direction* specifier as well as some *weight*; direction and weight are optional; a **coordinates** array containing the x, y, and z coordinates of a node; the z coordinate is optional; a **feature vector** containing a hashmap of arbitrary length containing objects of arbitrary type including nested structures. The feature vector is solely used by applications building on GraphiniusJS and ignored by the standard suite of algorithms as described in the previous section. A sample of a valid graph in the Graphinius JSON file format is depicted in Figure 7.3.

7.3 The History system

The idea of a history subsystem came from the concept of a real-time in-browser graph exploration platform, in which every action performed via an online editor or some GUI action should result in some immediate, visible change in the graph visualization. Ideally, such real-time changes would also be reversible, so that a user could progress step-by-step forward and backward in time - either to grasp more clearly what some algorithm does to the structure of a graph, or to 'simulate' the behavior of an algorithm on the whole.

In order to guarantee smooth behavior as well as separation of concerns in the software, placing this functionality directly in either the GraphiniusJS or GraphiniusVIS libraries would violate sound architectural principles. Therefor, the author proposes (but has not implemented yet) the following general module:

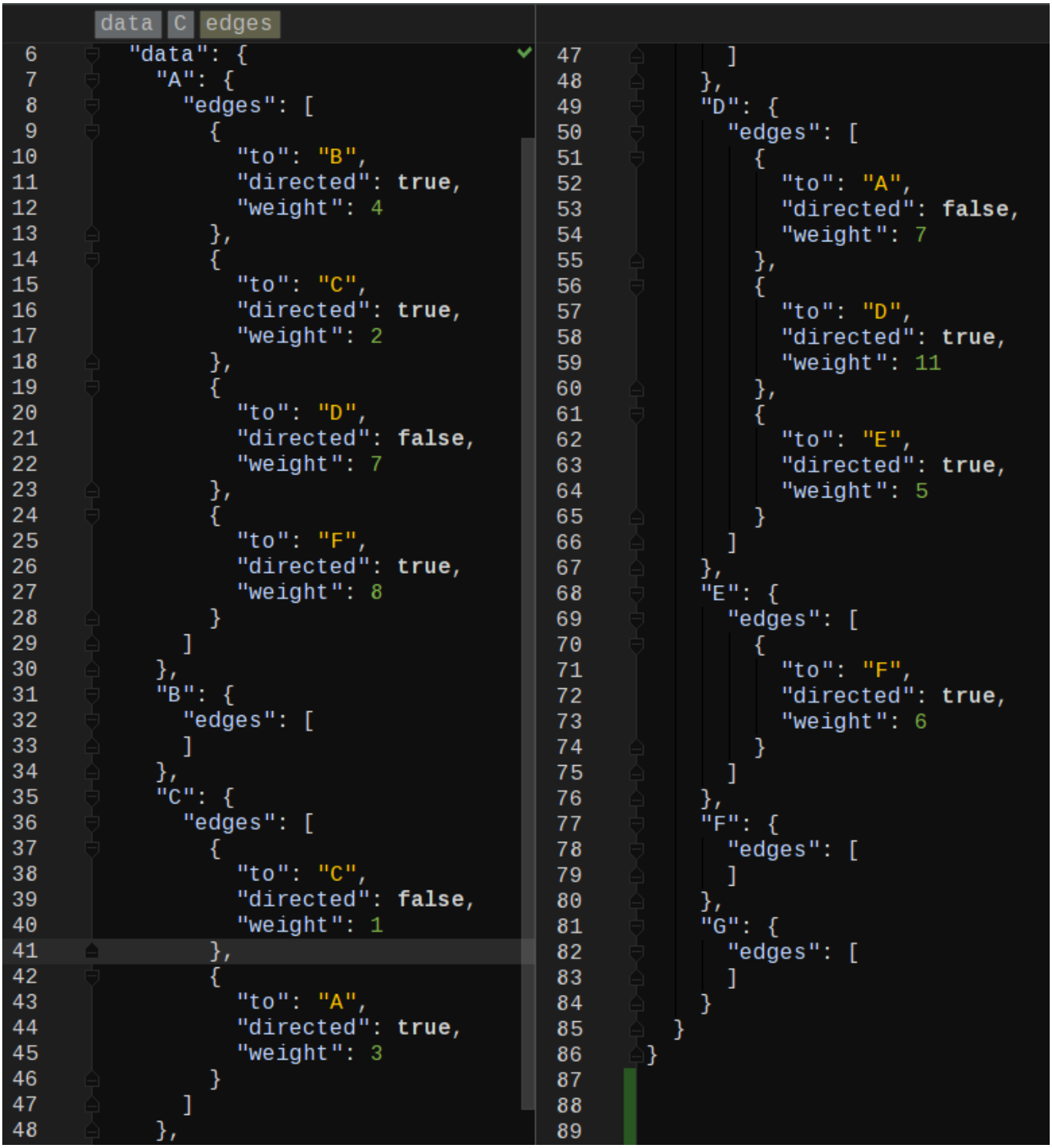


Figure 7.3: Sample graph in the Graphinius JSON format
Apart from the 'to' node, direction and weight, any node can exhibit an arbitrarily large feature vector containing any type of information (like patient data, word vectors, etc.). Another special sub-object which the input reader is looking for is the 'coords' object, which specifies the coordinates used in the constant layout renderer of the GraphiniusVIS library.

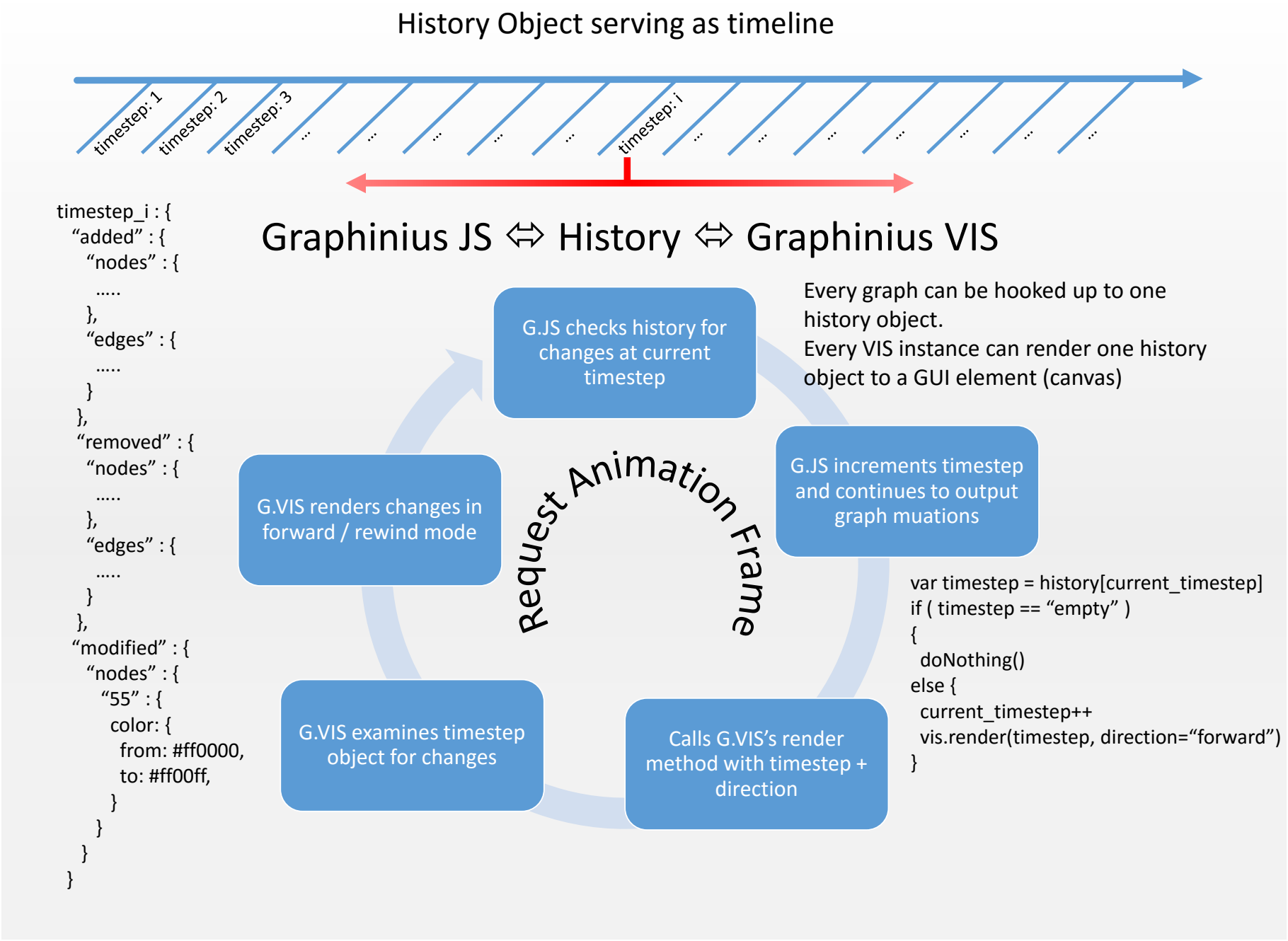


Figure 7.4: Graphinius JS <-> VIS communication via Op-Log

7.3.1 Timeline

The basic idea in implementing the system is to organize all graph mutations along the time axis, if only to imply chronological order. While it is less important to be able to specifically 'audit' some action in the sense of being able to exactly locate its temporal occurrence, we need some mapping of actions to a timestep (object) in order to be able to traverse the timeline back and forth. This is what determines the structure of the history object.

7.3.2 History Object

The history object is a simple JavaScript object (which behaves like a hashmap in other languages' vocabulary) and holds entries in the form of *timestep(timestep:number => actions:JSObject)*. These actions sub-objects then contain all the mutations which took place at a certain *timestep*. It is important to realize that this *timestep* is not an exact moment in time, but can span an arbitrarily long interval. Its value is determined by a global *timestep* variable, which is only incremented when the rendering of the currently 'active' timestep has commenced. The actual procedure take the form of the following loop:

1. Graphinius Platforms runs recursive calls to *window.requestAnimationFrame(renderFunc)* which simulates a main GUI loop
2. At the beginning of each invocation, the algorithm checks for the current value of the global *timestep* variable, and checks if the respective entry in the history object is empty or populated.
3. In case it is empty, the algorithm breaks, laying inactive until the next time-tick from requestAnimationFrame (16.67 milliseconds on a 60Hz monitor).
4. In case it is populated, the algorithm increments the global *timestep* value - from this point onwards, GraphiniusJS will not write to that entry any more ('locking' it so that the rendering process can finish).
5. All items in the current *timestep* object will be evaluated resulting in calls to the GraphiniusVIS library updating the in-browser visualization.

In order to be able to replay / rewind the mechanism, the history subsystem must have a sense of direction in time - which in our case simply translates to possessing a

vocabulary expressing graph mutations in such a way that the respective inverse function is easily obtainable.

7.3.3 Vocabulary

In order to achieve this, the *timestep* entries must be as simple and specific as possible. For most of the primitive graph mutations, this is practically going to happen automatically: For instance, given the command *'addNode(nodeId, arguments)'*, the inverse is logically *'deleteNode(nodeId)'*. For more complex actions like changing coordinates, colors or shape, the original values would have to be stored. In the case of actions regarding whole clusters or changing the structure of the entire graph (*run mincut..*), the *timestep* entries will have to be broken down into atomic units and inverse actions defined in beforehand.

7.4 Graphinius VIS

The author was fortunate to receive the opportunity to guide the Master's Project of Nicole Neuhold in implementing a visualization module for the emerging Graphinius platform. We were working on research, experiments, and the foundation of a future implementation from early January 2016 to the end of March of the same year, and I am proud to be able to say that we surpassed our initial expectations - in brevity and conciseness of implementation as well as performance - by leaps and bounds and are not able to visualize graphs of 15k nodes / 40k edges fluently (25 FPS) even on middle class laptops (22k nodes / 65k edges fluently on the author's desktop machine featuring a low-middle-class Geforce 650TI with 2GB of video RAM).

7.4.1 WebGL rendering

The core component of the GraphiniusVIS module is the WebGL renderer. Although we experimented with SVG and Canvas (2D) as well, we quickly realized that both alternatives were either too slow or didn't provide us with the experience we desired: SVG has the great advantage of working with normal browser (DOM) objects, which enables easy interaction and selection via JS / CSS selectors, but stops rendering fluently at only a few hundred nodes / a few thousand edges.

Canvas, on the other hand, is fine and fast enough for visualizing logical graph structures of thousands of nodes / edges in 2D. Nevertheless, because our project originated

from the need to render structures inherently 3D (like nevi and organs) and there was no easy possibility to project a 3D space onto a 2D canvas (except for computing the projection ourselves), we finally decided against it.

Our Three.js / WebGL based renderer now uses low-level data structures like buffer geometries, statically typed JavaScript arrays and particle systems instead of 3D objects for nodes, which enables us to transfer our (pre-)computed data to the GPU in one single copy operation - this alone increases performance easily 10-fold over earlier attempts at progressively adding new objects to the scene.

7.4.2 2D/3D Mode

GraphiniusVIS supports both 2D and 3D visualization of graph structures. However, since we are using WebGL which is inherently 3D, when switching to 2D mode we are not falling back to SVG or canvas but instead just 'fix' all z-coordinates of nodes / edges to zero, so we end up with a 2D object floating in 3D space.

7.4.3 Navigation

Our module supports panning (via mouse click-and-move), zooming (via mouse-wheel), rotating (via Shift + mouse click-and-move), and also provides any of those actions via keyboard commands.

7.4.4 Graph Layouts

The field of graph drawing has been an active area of research for several decades now, and many graph layouts have been developed for diverse areas of applications. Apart from constant (coordinate-based) layouts and force-directed layouts for physical simulations, there exist circular, spherical, tree-based, and arc diagrams, to name only a tiny fraction. Our basic implementation supports a constant layout per default, and allows switching to a force-directed layout as well, although the latter is currently implemented as a simple mathematical sine function instead of making use of attracting / repulsive forces.

7.4.5 Interaction / Manipulation

There are almost endless possibilities to interact with and manipulate a graph structure, so we were limited to offering just a small selection in order to demonstrate the viability

of our GraphiniusVIS module. We chose to implement node and edge addition as well as deletion, changing the color of nodes and edges, updating their coordinates as well as switching from constant to our (simplified) force-directed layout.

In addition to that, in order to demonstrate seemless interaction (without a history object for now) between GraphiniusJS and GraphiniusVIS, one can visualize distances computed by BFS as well as segments computed by DFS (on a directed graph) from any random or chosen node. This even works during constant re-rendering in force-directed mode, although the algorithm is not yet implemented as a background thread (WebWorker / WebAssembly) and therefore causes the animation to lag for a fraction of a second.

The following diagram summarizes some demo interactions / control flows contained in our base implementation.

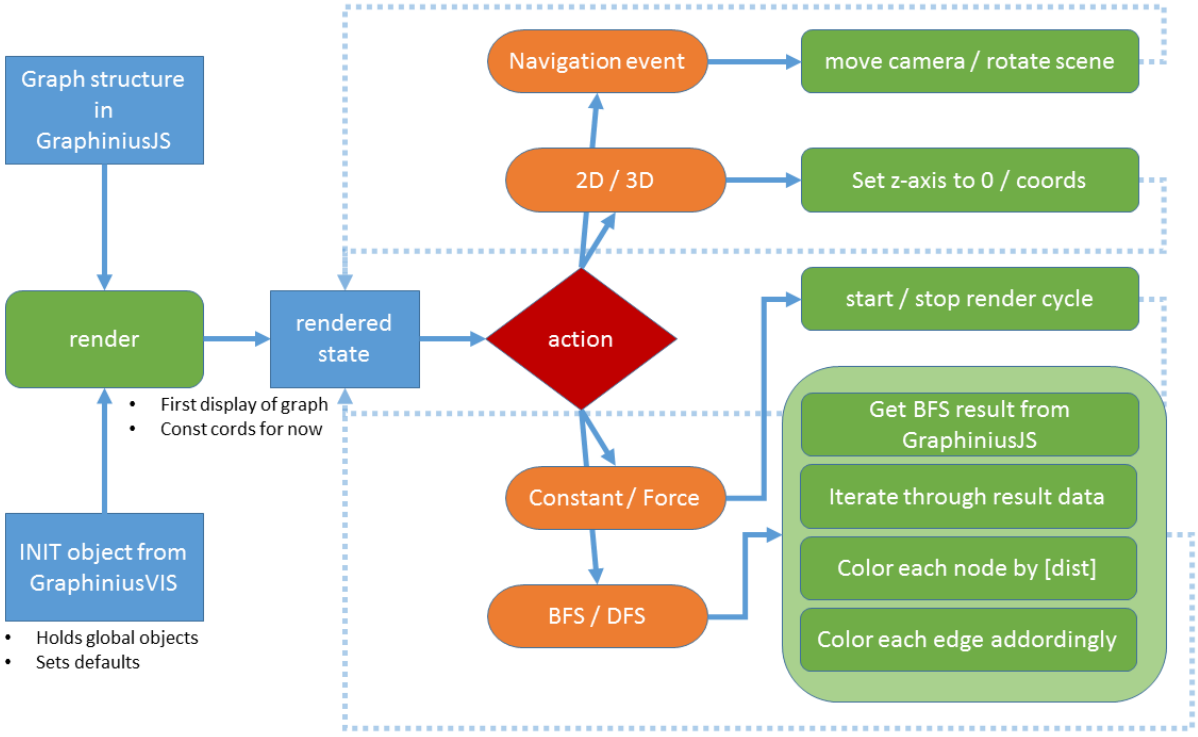


Figure 7.5: Graphinius VIS control flow

7.5 Dependent Libraries

As described in the last chapter 6, modern web development processes utilize a wealth of external modules and helpers to achieve a seamless, convenient development workflow. Whereas GraphiniusJS does not depend on any external libraries at runtime (as can be

seen in the *"dependencies"* section of Figure 7.6), the author has been using a diverse collection of dependencies at development time, which shall be only introduced in all brevity:

The **chai** library handles assertions in mocha tests, **gulp** is the main taskrunner library, **gulp-clean** provides for deletion of compilation- and other results, **gulp-concat** manages file concatenations, **gulp-istanbul** integrates istanbul into the gulp process, **gulp-mocha** integrates mocha into the gulp process, **gulp-rename** handles renaming of one or many sources to a single destination file, **gulp-typedoc** integrates typedoc into the gulp process, **gulp-uglify** integrates uglify into the gulp process, **gulp-watch** provides file watching capabilities, **istanbul** provides test coverage reports, **jsdom** simulates a DOM environment in pure JavaScript (NodeJS), **jsdom-global** provides global variable support for jsdom, **json-loader** is a submodule of webpack allowing for integration of JSON files into a minified JS bundle, **merge2** merges file contents, **mocha** is the main test runner, **sinon** provides spies and stubs for testing, **sinon-chai** integrates sinon with the chai assertion library, **typedoc** automatically generates documentation from Typescript sources, **webpack-stream** is necessary for running a webpack process inside a (stream-based) gulp task, and **xhr-mock** provides a mocking service for simulating browser-based XMLHTTPRequest objects in server-side NodeJS.

```
"name": "graphinius",
"version": "0.2.7",
"description": "Generic graph (analysis) library in Typescript",
"main": "index.js",
"author": "Bernd Malle",
"license": "Apache-2.0",
"bugs": {
  "url": "https://github.com/cassinius/Graphinius/issues"
},
"homepage": "https://github.com/cassinius/Graphinius#readme",
"devDependencies": {
  "chai": "^3.4.1",
  "gulp": "^3.9.0",
  "gulp-clean": "^0.3.1",
  "gulp-concat": "^2.6.0",
  "gulp-istanbul": "^0.10.3",
  "gulp-mocha": "^2.2.0",
  "gulp-rename": "^1.2.2",
  "gulp-typedoc": "^1.2.1",
  "gulp-typescript": "^2.9.2",
  "gulp-uglify": "^1.5.3",
  "gulp-watch": "^4.3.5",
  "istanbul": "^0.4.2",
  "jsdom": "8.2.0",
  "jsdom-global": "1.7.0",
  "json-loader": "^0.5.4",
  "merge2": "^1.0.0",
  "mocha": "^2.3.4",
  "sinon": "^1.17.3",
  "sinon-chai": "^2.8.0",
  "typedoc": "^0.3.12",
  "webpack-stream": "^3.1.0",
  "xhr-mock": "^1.6.0"
},
"dependencies": {}
```

Figure 7.6: GraphiniusJS development and runtime dependencies

As is clearly visible, I focused on managing all complexity during development time, resulting in zero dependencies for the runtime JS bundle.

7.6 Testing approach

Behavior Driven Development (BDD) is a testing philosophy that approaches the development of a new feature from a high level view on the expected end-result and then works it's way down from spec's through functional to unit tests, until the feature is fully implemented, tested and covered. The ideal methodology would consist of the following steps:

1. Define the expected behavior of a feature (=module) in a form that both clients as well as developers understand and formulate them using (executable) specifications. The Cucumber framework introduced in the last chapter is an ideal candidate for this high level of abstraction. Executing the spec immediately will naturally fail because no code was actually written.

2. From the specs defined in step one, we then derive functional tests which span the execution of several functions or methods in sequence in order to produce a certain result. Again, any test execution will expectedly fail as we have still not written the code yet.

3. From the functional tests defined in step two, we finally derive low-level tests covering single units of code. Once more the tests will fail initially, upon which we endeavor to fill in the single pieces of code until all tests at that level are passing.

4. Once the unit-test level is done, we work our way upwards to the next functional test on our todo-list. This process is repeated until all functional tests required for our initially specified features are covered.

5. We then move on to the next feature on our product requirements list.

As already mentioned, because a software library intended to be used by other programmers usually has only technical clients, feature specification (testing) was omitted in programming Graphinius JS. For this reason, we will only take a look at unit and functional tests in this section (as well was mocking and spying). Furthermore, in developing a web application using the Mocha and Chai test libraries, there is no fundamental difference between unit and functional tests. Let us therefore take a look at instances of both categories to see how we would structure our approach:

7.6.1 Unit tests

As stated, unit tests cover the low-level constructs of functions or methods which themselves do not depend on any lower-level functions.

In the example below, we see two test cases: the first calls a binary Heap evalInputPriority function on line 2, whose job it is to cast any input to a (sortable) number, if possible. As the string "55" can be cast to the number 55, this test will be successful. In the second example on lines 5 and 6, we pass boolean values which - employing the JS *parseInt(arg)* function - will evaluate to NaN (which is JS shorthand for *Not a Number*).

```
1 it('should accept String encoded Integers as input and
     evaluate to their Integer value', () => {
2     expect(binHeap.evalInputPriority("55")).to.equal(55);
3 });
4 it('should not accept booleans as input values (makes no
     sense...) ', () => {
5     expect(binHeap.evalInputPriority(true)).to.be.NaN;
6     expect(binHeap.evalInputPriority(false)).to.be.NaN;
7 });
```

Listing 7.6: Unit tests covering the functionality of one simple function.

(Sample taken from the GraphiniusJS binaryHeapTests.ts file.)

7.6.2 Functional tests

In functional testing we invoke some procedure which in turn will call other functions or methods, in any sequence or call depth required in order to fulfill its purpose.

In the example below, we instantiate a new JSON reader, specify some configuration options and hand it a JSON file. We then expect the resulting graph to be of a certain size in number of nodes and edges. The *readFromJSONFile* function will itself call several subordinate functions for reading a file from disk, processing the character sequence it receives, instantiating a GraphiniusJS graph object etc.

```
1 it('should correctly generate our small example graph out
     of a JSON file with direction _mode set to undirected',
     () => {
```

```
2     json = new JSON_IN();
3     json._explicit_direction = false;
4     json._direction = false;
5     graph = json.readFromJSONFile(small_graph);
6     expect(graph.nrNodes()).to.equal(4);
7     expect(graph.nrDirEdges()).to.equal(0);
8     expect(graph.nrUndEdges()).to.equal(4);
9 });
```

Listing 7.7: A functional test covering the whole instantiation process of a graph from a JSON input structure.

(Sample taken from the GraphiniusJS JSONInputTests.ts file.)

7.6.3 Mocks used for browser code testing

Mocks are a useful construct for testing functionality that would involve some non-trivial behavior. For instance, if we would like to test a snippet of code (as in the following code example) which loads a JSON file remotely over the network before instantiating a graph, we are assuming the existence of a server, the existence of the remote file, as well as a working Internet connection. Moreover, in this specific case, the code under test is also meant to be executed inside a browser environment, whereas we would like to invoke our test from the NodeJS console.

The solution to this problem is to mock the XMLHTTPRequest object used by a browser to send AJAX requests over the internet, which - again in this specific case - replaces the object with a NodeJS request object and simulates the original XMLHTTPRequest API through a delegating wrapper around it. The exact sequence in this example is: 1) requiring the mocking library on line 3, 2) initializing a browser environment (providing the window and document objects) on line 4, 3) injecting browser globals into the Node environment on line 13, 4) requiring filesystem capabilities for local file loading on line 16 as well as loading the file on line 17, 5) initiating the mock on line 20 and finally setting up a fake web server responding to a certain URL on lines 23 through 30. The rest of the procedure (not shown in this exmaple) behaves exactly as an equivalent test inside a browser environment would.

```
1 describe('Loading graphs in simulated browser environment',
      () => {
```

```
2     // Mocking the XHR object
3     var mock = require('xhr-mock');
4     var jsDomCleanup = null,
5     mocked = false;
6
7     // URL to replace with path
8     var small_graph_url = REMOTE_HOST + "small_graph.json";
9     var small_graph_path = 'test/input/test_data/
         small_graph.json';
10
11    beforeEach(() => {
12        // Injecting browser globals into our Node
             environment
13        jsDomCleanup = require('jsdom-global')();
14
15        // Access to local filesystem for mocking service
16        var fs = require('fs');
17        var json = fs.readFileSync(small_graph_path).
             toString();
18
19        //replace the real XHR object with the mock XHR
             object
20        mock.setup();
21
22        // Mocking Browser GET request to test server
23            mock.get(small_graph_url, function(req, res) {
24                mocked = true;
25                return res
26                    .status(200)
27                    .header('Content-Type', 'application/
                         json')
28                    .body(json);
29            });
30        });
31
```

```
32      afterEach(() => {
33          mock.teardown();
34          jsDomCleanup();
35          mocked = false;
36      });
37
38      // Rest of tests are the same as in non-mocked, local
            reader based input tests
39  });
```

Listing 7.8: A mocking setup simulating a Web Server GET response.

... to an XML-HTTP GET Request in order to read a graph structure from a remote JSON file from inside a browser environment. (Sample taken from the GraphiniusJS JSONInputAsyncTests.ts file.)

7.6.4 Stubs

There is another concept often confused with mocks called *stubs*. Stubs are used when the response of a function is not supposed to be complex but rather boolean in nature. E.g. an authorization module could internally use a method checking if a user has sufficient permissions to be allowed to access a resource. This functionality could be implemented rather simply, or it could invoke elaborate tests involving diverse software modules throughout the whole system. For testing some function building upon it however, only the distinction between *allowed* and *not allowed* is are really of interest. Therefore, a stub can be instantiated and told to forgo any *real* authority checks but instead simply return a true or false value.

The difference between stubs and mocks is therefore that stubs replace potentially complex implementations with trivial ones, whereas mocks can act as a proxy but do not necessarily reduce complexity - reading a graph from a local file is as complex as reading it from a remote one (not considering the underlying complexity of the network stack, of course).

7.6.5 Spies (Sinon)

Spies are test wrapper objects that observe the original function or method and record any activity regarding it, e.g. how often it was called, which argument values it was called with, which values were returned or if an error was thrown. In the example below, we instantiate a local backup object *original*, then instantiate a spy on line 10, store the original function reference in our backup object on line 12, and replace the original function with our spy on line 14. In calling $DFS.prepareDFSVisitStandardConfig on line 25, we expect to see the original function invoked in the process, which we check on line 28. The *after* block from lines 18 through 21 then restores the original objects.

```
1 describe('testing config preparation functions - ', () => {
2     var prepForDFSVisitSpy,
3     prepForDFSSpy,
4     original = {
5         prepareDFSStandardConfig: null,
6         prepareDFSVisitStandardConfig: null
7     };
8
9     before(() => {
10        prepForDFSSpy = sinon.spy($DFS.
              prepareDFSStandardConfig);
11        prepForDFSVisitSpy = sinon.spy($DFS.
              prepareDFSVisitStandardConfig);
12        original.prepareDFSStandardConfig = $DFS.
              prepareDFSStandardConfig;
13        original.prepareDFSVisitStandardConfig = $DFS.
              prepareDFSVisitStandardConfig;
14        $DFS.prepareDFSStandardConfig = prepForDFSSpy;
15        $DFS.prepareDFSVisitStandardConfig =
              prepForDFSVisitSpy;
16    });
17
18    after(() => {
19        $DFS.prepareDFSStandardConfig = original.
              prepareDFSStandardConfig;
```

```
20          $DFS.prepareDFSVisitStandardConfig = original.
                prepareDFSVisitStandardConfig;
21      });
22
23
24      it('preprareDFSVisitStandardConfig should correctly
            instantiate a DFSConfig object', () => {
25          var config = $DFS.prepareDFSVisitStandardConfig();
26
27          // Here the spy is finally used to check internal
                method invocation
28          expect(prepForDFSVisitSpy).to.have.been.calledOnce;
29      });
30  });
```

Listing 7.9: A functional test making use of a spy.

... to test internal method invocation, restoring the original method afterwards. (Sample taken from the GraphiniusJS DFSTests.ts file.)

7.7 Areas of Application

The implementation of 3 demo applications will be described in the next Chapter, Implementation - Areas of Application) 8.

7.8 Platform Services

As of the time of this writing, the Graphinius Platform has not taken shape yet, so there is no available implementation to discuss.

8. Implementation - Areas of Application

As described in earlier chapters of this thesis, there are countless applications of graph theory in diverse fields of research and engineering; in order to prove the feasibility of the Graphinius platform, it was necessary to implement a few concrete examples. Consequently, in this chapter we are going to take a look at 3 different Areas of Application that Graphinius (JS, VIS, and eventually the platform) is already able to serve. Amongst these, only the first one can be considered a toy application (although interesting for teaching platforms etc. in itself); graph extraction from images as well as social network anonymization however have been hitherto firmly situated in the realm of servers or entire processing infrastructures.

8.1 Manual editing (predefined structures)

The first and foremost use case for Graphinius is simply to be able to interactively build, mutate, and visualize graphs in the browser. Although the final Graphinius Platform will feature a full-blown code editor with code completion and online documentation, the basic functionality can be demonstrated even using a form of REPL every modern browser is automatically equipped with: the debugging console.

8.1.1 Build a graph manually

As depicted in Figure 8.1, the basic case is to create a new graph structure, add some nodes and edges, and then run different computations on it.

8.1.2 Load predefined graph and visualize

Using either the CSV or JSON Reader build into GraphiniusJS, we can also request to instantiate a graph from a remote file. Here we use the JSON Reader to load a graph depicting a nevus and render it using GraphiniusVIS (Figure 8.2). Apart from the visualization, we also compute its degree distribution.

```
> var graph = new $G.core.Graph("newGraph")
< undefined
> graph
< ▶e {_label: "newGraph", _nr_nodes: 0, _nr_dir_edges: 0, _nr_und_edges: 0, _mode: 0…}
> var nodeA = graph.addNode("A")
< undefined
> var nodeB = graph.addNode("B")
< undefined
> var edge_ab = graph.addEdge("a_b", nodeA, nodeB)
< undefined
> var edge_ab = graph.addEdge("a_a", nodeA, nodeA, {directed: true, weighted: true, weight: 55})
< undefined
> var edge_ab = graph.addEdge("b_a", nodeA, nodeB, {directed: true})
< undefined
> graph.getStats()
< Object {mode: 3, nr_nodes: 2, nr_und_edges: 1, nr_dir_edges: 2}
> graph.degreeDistribution().all
< [0, 0, 1, 0, 1]
```

Figure 8.1: Manually building a new graph in the console.

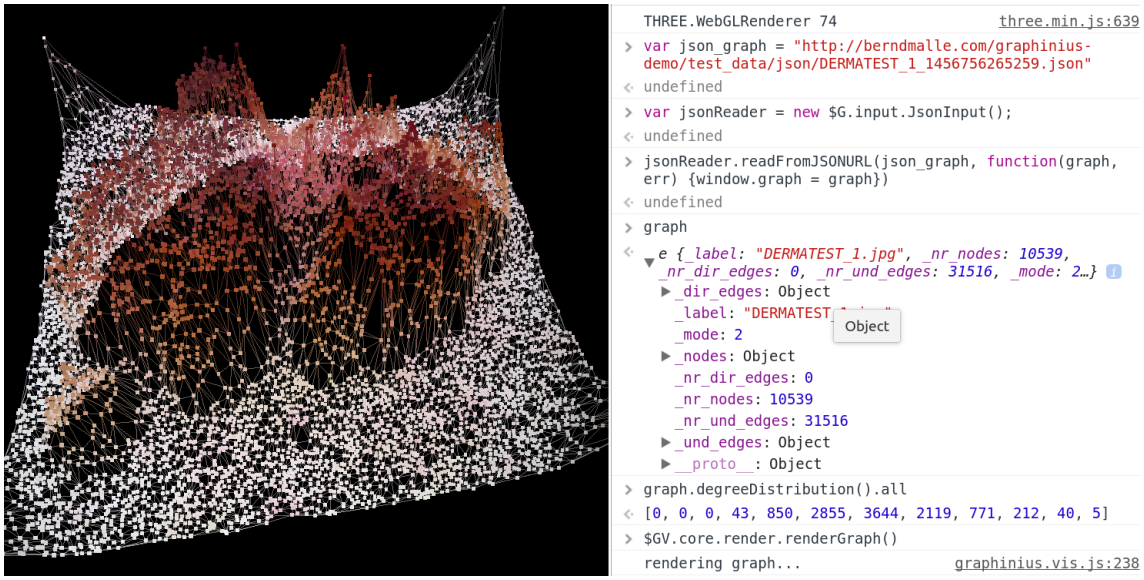


Figure 8.2: Loading a JSON graph and visualizing it via the browser console.

8.1.3 Run a BFS algorithm and visualize

After loading a (undirected) graph according to the previous section, we choose a random start node and invoke a breadth-first-search algorithm resulting in a *distance map* centering around that node. The following lines of code (Figure 8.3) show distances and parents of a selection of nodes (note the parent / distance chain...) while the accompanying visu-

alization colors the graph according to the obtained distances via gradient computations (the start node being colored green and the node with maximum distance being colored red).

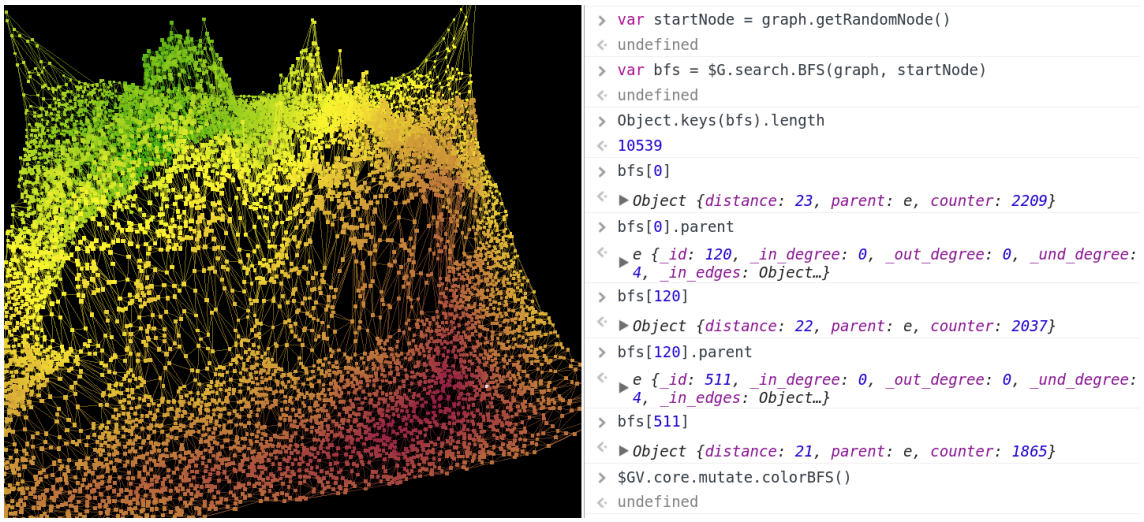


Figure 8.3: Computing a BFS in a live browser REPL & visualizing the result.

8.1.4 Run a DFS algorithm and visualize

Lastly, in Figure 8.4 we load the same graph as before, this time interpreted as a directed graph, choose a random start node again and invoke a DFS algorithm. This returns to us an array of graph segments representing the node sets reachable from each start node of a respective DFS Visit run (had we chosen an undirected graph, there would only be a single segment). We then output the size of each segment and again visualize the result, assigning to each segment a different color.

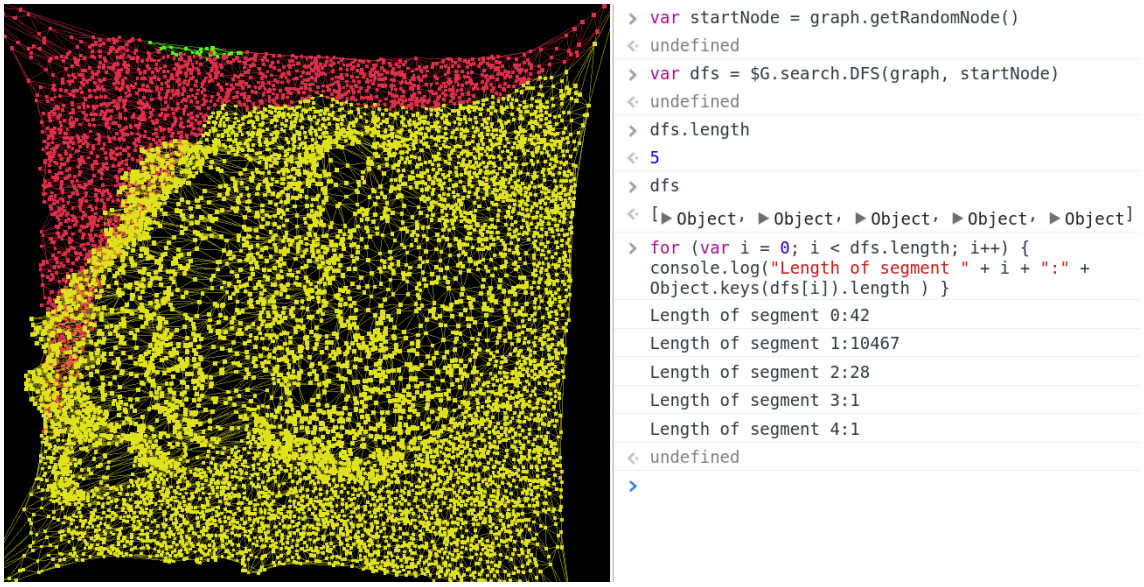


Figure 8.4: Computing a DFS in a live browser REPL & visualizing the result.

8.2 Graph extraction from images

In order to be able to apply graph theory to problems originating in the realm of image processing, first we need to extract a graph structure out of an image. There are potentially many different ways of doing this; as described in our paper (Holzinger, Malle, and Giuliani, 2014), we are executing the following steps:

1. **Image preprocessing.** Many image segmentation algorithms use gray scale values to compute distances between neighboring pixels, gradients etc. (it seems that using 3 color channels per pixel is not much superior over using just one).

2. **Input image as graph.** The resulting image is simply interpreted as a graph, which is possible because every pixel-based image naturally forms a graph structure, in which pixels are represented by nodes and neighborhoods are represented by edges. We then extract an edge list where the entries is sorted according to edge weight (= differences in the intensity values of neighboring nodes).

3. **Graph based (over-)segmentation.** As the next step, a Kruskal MST based algorithm developed by (Felzenszwalb and Huttenlocher, 2004) is applied. It takes the approach of merging regions from pixel level (= inital nodes) 'upwards' instead of recursively partitioning the whole image 'downwards' - essentially, at every step it compares an intra-region coherence measure to an inter-region similarity measure:

$$\mathrm{Int}(C) = \max_{e \in \mathrm{MST}(C,E)} \omega(e)$$

is the intra-region coherence value, given by the maximum edge weight of the region's MST.

$$\mathrm{Dif}(C1, C2) = \min_{v_i \in C1, v_j \in C2, (v_i, v_j) \in E} \omega(v_i, v_j)$$

denotes the intra-region similarity measure, given by the minimum edge weight connecting any two nodes between them.

Finally,

$$\mathrm{D}(C1, C2) = \begin{cases} \mathit{true} & \text{if} \quad \mathrm{Dif}(C1, C2) > \mathrm{MInt}(C1, C2) \\ \mathit{false} & \text{otherwise} \end{cases}$$

determines if two regions should be merged, based on the relation of their intra-region coherence and inter-region similarity measures.

Once all edges have been considered, the final graph partition represents the segmentation (merging) result

4. **A Delauney triangulation** is computed once the final region map has been established, taking each region's centroid to become a node in the new graph, as well as the (non-overlapping, as the algorithm is producing a tessellation of the region map) connections between those centroids to be their respective edges. The edge weight is computed from the difference in average intensity value of two adjacent regions.

5. **Graph output.** Those data are finally transferred into a JSON structure consumable by Graphinius JS, as depicted by the sample JSON graph in Figure 7.3.

With the exception of manually written micro-graphs for the sake of rapid unit testing, this procedure was used for practically all JSON graphs employed in the development and testing of GraphiniusVIS. Due to the flexibility of the algorithm - it features 3 separate parameters which control the granularity of the partition, and therefore the granularity of the graph structure - it was possible to extract graphs from a few hundred nodes and edges to up to 22k nodes / 66k edges. The average execution time on the author's quad-core i5 machine lay in the range of 3-5 seconds.

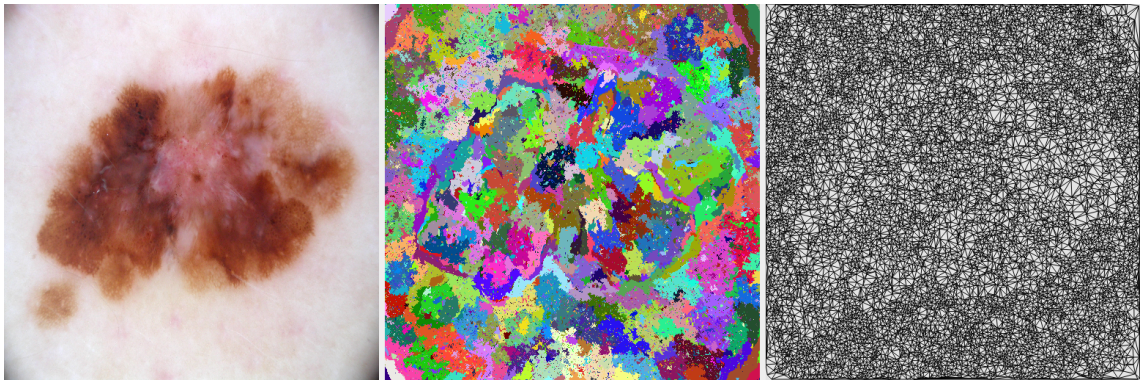


Figure 8.5: Kruskal MST based region merging & graph extraction.

Result of applying a Kruskal based region merging algorithm to an image of numerous small scale regular structures. (1) Input image, (2)

8.3 Anonymity: SaNGreeA

SaNGreeA stands for *Social network greedy clustering* and was introduced by Campan and Truta, 2009. In addition to 'clustering' nodes of a graph according to the minimum general information loss (GIL) incurred as described in Section 3.4, this algorithm also considers the structural information loss (SIL) by assigning a node to a certain cluster. The SIL quantifies the probability of error when trying to reconstruct the structure of the initial graph from its anonymized version.

The SIL is composed of two different components: 1) the intra-cluster structural loss, signifying the error probability in trying to reconstruct the original edge distribution within an equivalence class (= anonymized cluster), and 2) the inter-cluster structural loss which represents the error probability in trying to reconstruct the original configuration of edges between two equivalence classes.

In implementing and demonstrating this algorithm, I recreated the paper's original experiment:

1. **Process input data into suitable structure.** The adults dataset was selected and all but six columns deleted - only Age, Workclass, Country of origin, Gender, Race and Marital status remained (a sample containing the first 19 rows can be seen in Figure 8.6). Furthermore, in order to obtain repeatable results, the first 300 'pure' rows (no missing or mis-formatted values) in the dataset were chosen as input set.

2. **Enhance structure with graph information (random edges).** Using GraphiniusJS's capability of randomly adding edges to nodes, a connected graph was created out of the assortment of nodes (using between 1 and 10 outgoing edges per node).

3. **Compute GIL & NGIL.** The general information loss with respect to a cluster is given by the following formula (repeating from the original paper):

$$\mathrm{GIL}(cl) = |cl| \cdot (\sum_{j=1}^{s} \frac{size(gen(cl)[N_j])}{size(min_{x \epsilon N}(X[N_j]), max_{x \epsilon N}(X[N_j]))} + \sum_{j=1}^{t} \frac{height(\Lambda(gen(cl)[C_j]))}{height(H_{C_j})})$$

where:
- $|cl|$ denotes the cluster cl's cardinality;
- $size([i1, i2])$ is the size of the interval $[i1, i2]$, i.e., $(i2 - i1)$;
- $\Lambda(w), w \epsilon H_{C_j}$ is the subhierarchy of H_{C_j} rooted in w;
- $height(H_{C_j})$ denotes the height of the tree hierarchy H_{C_j};

The total generalization information loss is then given by:

$$\mathrm{GIL}(G, S) = \sum_{j=1}^{v} \mathrm{GIL}(cl_j)$$

And the normalized generalization information loss by:

$$\mathrm{NGIL}(G, S) = \frac{\mathrm{GIL}(G, S)}{n \cdot (s + t)}$$

4. **Compute SIL & NSIL.** For the exact mathematical definitions of SIL & NSIL the reader is kindly referred to the original paper. Because the structural information loss cannot be computed exactly before the final construction of clusters, the exact computations were replaced by the following distance measures:
Distance between two nodes:

$$\mathrm{dist}(X^i, X^j) = \frac{|\{l | l = 1..n \wedge l \neq i, j; b_l^i \neq b_l^j|}{n - 2}$$

Distance between a node and a cluster:

$$\mathrm{dist}(X, cl) = \frac{\sum_{X^j \epsilon cl} \mathrm{dist}(X, X^j)}{|cl|}$$

The algorithm starts with initializing a first cluster by simply adding a randomly chosen

node to it. Then, for every new node encountered, the weighted sum of the above two information loss metrics will yield a certain overall information loss value if the node was added to that cluster - the node with the minimal cost is then chosen as the candidate and expands the cluster. This is repeated until the first cluster reaches a certain requirement (e.g. size == k-factor) upon which another random node is chosen to constitute a next cluster. This procedure is repeated until all nodes have been assigned (if a cluster of size < k-factor remains, its member nodes are dispersed amongst the others).

Since the algorithm does not take ALL possible node combinations into account, but simply start with a node and compares all the candidates in a loop, the algorithm runs in quadratic time w.r.t. the input size in number of nodes. This worked well within milliseconds for an input problem size of a few hundred nodes. An example output of the implemented algorithm can be found in Appendix A.

Age	Workclass	Country	Gender	Race	Marital status
39	State-gov	United-States	Male	White	Never-married
50	Self-emp-not-inc	United-States	Male	White	Married-civ-spouse
38	Private	United-States	Male	White	Divorced
53	Private	United-States	Male	Black	Married-civ-spouse
28	Private	Cuba	Female	Black	Married-civ-spouse
37	Private	United-States	Female	White	Married-civ-spouse
49	Private	Jamaica	Female	Black	Married-spouse-absent
52	Self-emp-not-inc	United-States	Male	White	Married-civ-spouse
31	Private	United-States	Female	White	Never-married
42	Private	United-States	Male	White	Married-civ-spouse
37	Private	United-States	Male	Black	Married-civ-spouse
30	State-gov	India	Male	Asian-Pac-Islander	Married-civ-spouse
23	Private	United-States	Female	White	Never-married
32	Private	United-States	Male	Black	Never-married
40	Private	?	Male	Asian-Pac-Islander	Married-civ-spouse
34	Private	Mexico	Male	Amer-Indian-Eskimo	Married-civ-spouse
25	Self-emp-not-inc	United-States	Male	White	Never-married
32	Private	United-States	Male	White	Never-married
38	Private	United-States	Male	White	Married-civ-spouse

Figure 8.6: Excerpt: the first 25 rows of the Adult census data set

9. Results

Testing the software is one part, testing its performance on real-world scenarios an other. The following sections give a few metrics of Graphinius as of the time of this writing.

9.1 Size of the codebase

Lines of codes never accurately measure the complexity involved in a software project, as different languages and programming styles produce very diverging measures in that area (Java e.g. being much more ceremonial than Ruby or JavaScript). Nevertheless, here are the metrics in LOCs for the different parts of Graphinius as of May, 2nd, 2016, amounting to a total of 11,858 lines of code:

9.1.1 GraphiniusJS

The core JS library was split into source code as well as testing code, with the testing code again split into 3 different categories: 1) Synchronous tests are the ones that run fast enough so they can be continually re-executed on every file save, 2) Asynchronous test code for testing graph instantiation via remote file loading, and 3) Performance test code loading different graphs and executing some basic algorithms on them, measuring the time at every step.
Source code: 2,557 LOC
Test code synchronous: 5,160 LOC
Test code, asynchronous: 321 LOC
Test code, performance: 76 LOC
Total: 8,114 LOC.

9.1.2 Graph extraction demo code

The Anonymization demo library building on top of GraphiniusJS is split into source as well as synchronous test code, there was no need to introduce asynchronous test code in its case.

Source code: 602 LOC
Test code synchronous: 448 LOC
Total: 1,050 LOC

9.1.3 Social network anonymization demo code

The Anonymization demo library building on top of GraphiniusJS is split into source as well as synchronous test code, there was no need to introduce asynchronous test code in its case.
Source code: 778 LOC
Test code synchronous: 455 LOC
Total: 1,233 LOC

9.1.4 GraphiniusVIS

The visualization library was written in pure JavaScript instead of TypeScript and due to its optical nature does not feature any automated test suite.
At the time of this writing the code base comprises 1,461 lines of code.

9.2 Test coverage (just Graphinius JS)

Testing is a great practice to guide the development and programming effort while conducting a software project, but it is of equal importance as a documentation tool allowing the technical staff to demonstrate to their clients (customers and managers alike) the care they exercised in constructing their codebase. Coverage testing detects if parts of the codebase were either just partly tested or not tested at all, taking into account LOC coverage, percentage of functions / methods invoked as well as branch or general statement coverage.

As can be seen in Figure 9.1, Graphinius JS has been exhaustively tested, reaching 100% coverage for all but the branches department. The lower value in this section is a result of *else-branches* not taken, in cases when there was no *else-branch* to take. This can be remedied by re-writing all *if-statements* in the form of *(boolean condition && consequent)*, so that the code coverage tool does not recognize the *if* keyword. The author has successfully tested this practice on the PFS.ts file (as can be seen in Figure 9.1) but at

the time of this writing considers it inappropriate to alter perfectly good code throughout the library just for the sake of artificially achieving 100% coverage.

9.3 Execution speed in various scenarios

All tests were executed on the author's Sandy Bride Quad Core i5, 8GB RAM, Ubuntu 16.04LTS, NodeJS version 5.5.0, bash console. All graphs were given as a simple CSV edge list. Experiments were run 10 times and average times (in milliseconds) taken; the measurements were conducted via JavaScript which translates to system real time (including all other processes running). The respective system time (only this process) was measured by the Unix *time* utility and refers to a whole 'run', that means all of computations listed above it in sequence:

9.3.1 Sample graph 1

The first sample graph consisted of 19,878 nodes and 49,062 edges:
Time to instantiate: 450ms
Calculating degree distribution: 28ms
Calculating BFS: 265ms
Calculating degree distribution: 260ms
Average system time: 48ms

9.3.2 Sample graph 2

The first sample graph consisted of 52,537 nodes and 140,868 edges:
Time to instantiate: 1,260ms
Calculating degree distribution: 51ms
Calculating BFS: 680ms
Calculating degree distribution: 790ms
Average system time: 170ms

9.3.3 Sample graph 3

The first sample graph consisted of 238,986 nodes and 797,115 edges:
Time to instantiate: 7,703ms

Calculating degree distribution: 208ms
Calculating BFS: 3568ms
Calculating degree distribution: 4993ms
Average system time: 454ms

9.4 Closing remarks about competitor libraries

The author contemplated writing a whole chapter about performance comparisons with other graph libraries as there are many of them out there (BGL, iGraph, GraphTool, NetworkX, Loom, some old Java based libraries, etc.). However, given the fact that **none** of those libraries were designed to run inside a browser including tight integration with a real-time 3D, Open/WebGL based visualization library, such comparisons would have been meaningless in any situation in which a user would require a non-effort, easily-configurable platform as ours.

Figure 9.1: GraphiniusJS test coverage

10. Future Work

Considering GRAPHINIUS an arbitrarily extendable computing platform (which uses graphs as underlying, universal data structures), there are many possibilities to build upon this work, ranging from small improvements to the introduction of fundamentally new infrastructure, transcending the use of contemporary graph libraries.

The use of a centralized, Web based graphical workflow system will prove especially useful in exploiting and propagating the experience of individual users, as it bundles not only data, but also the settings and results of all experiments conducted on that platform.

10.1 Parallel processing (CPU)

At the time of this writing (early 2016), real parallel processing inside the browser cannot be achieved without falling back to proprietary technologies like Google's *native client* (Yee et al., 2009). Although it is true that the Web Worker specification has allowed for parallel execution of threads inside the browser for several years now, this model prohibits the use of a shared space of memory. This limitation renders any serious attempt at multiprocessing futile, as not does the main thread have to copy each argument over to a Web Worker, the Workers would also constantly have to communicate their progress back to the main thread. It is obvious that such an approach will simply result in unmanageable overhead.

A new model for SMMT (shared memory multi-threading) revolves around the emerging standard of WebAssembly (WASM), which can be compiled down to ASM and directly passed to the host kernel by the JavaScript Virtual Machine. Instead of writing WASM manually (which will be possible, too) the ideal approach is to rather code in a different language (C++, Typescript etc.) and compile the resulting code to WASM.

10.2 Parallel processing (GPU)

An even faster - and already feasible - alternative is to conduct computations directly on the graphics hardware, which on modern computers (and even mobile devices) is capable

of astonishing performance. The only hindrances to programming directly on the GPU via the web browser are 1) the mandatory usage of the GLSL (OpenGL Shader Language), 2) only complicated copying of data to and from the graphics card (no shared memory with the JSVM) as well as 3) the possible emergence of far superior ways of GPU coding (like WebCL), which would render extensive investments & efforts today relatively useless in the near-future.

10.3 General processing / ML pipelines

As existing stacks indicate, there are several possible levels of pipelines which can be sorted by increasing homogeneity amongst their stages as well as decreasing technical demands on their users:

- Level 0: Writing all of the pipeline manually. As every combination of technologies are usable, this approach gives the most flexibility but is hard to maintain and almost impossible to reproduce. As (Sculley et al., 2014) perfectly states: "Using self-contained solutions often results in a glue code system design pattern, in which a massive amount of supporting code is written to get data into and out of general-purpose packages."
- Level 1: Automated, but self designed and coded. This entails the usage of tools like Unix Make, which is language agnostic and therefore supports any number of technologies as long as they are executable from a Shell. Apart from slightly better decoupling, same problems as Level 0.
- Level 2: Establishing a common understanding of the components and structure of a pipeline while still using individual technologies. Such a common standard exists in the form of PMML - the Predictive Model Markup Language. PMML defines stages of pipelines as well as their inputs, parameter types and ranges, the output format etc. Thus, developers can use their favorite technologies during development while PMML-consuming tools then produce familiar code (Hadoop, Spark etc.).
- Level 3: Language specific libraries exposing an API to conveniently assemble a pipeline. Such libraries have been released by projects like scikit-learn or Apache Spark. While currently becoming popular, APIs still restrict the creation of data applications to experts capable of coding.

- Level 4: The use of a custom DSL would widen the ability to create complex data pipelines to any kind of domain expert. Similar in nature to SQL, it is able to either compile itself into code or an intermediate representation like PMML.
- Level 5: A fully integrated data analysis platform that offers intuitive, visual pipeline assembly. Ideally, tools for reporting, reproduction and collaboration would also be included. In addition, the platform could offer experts the means to write stages themselves via an online code editor.

As many areas of applications for graph theory could make use of integrated processing pipelines (Sparks, 2014), the future of Graphinius could lie in providing a Level 5 experience.

10.4 JSVM based grid computing

In order to process large graphs, single instances of the JSVM will no be sufficient. In order to make this scenario possible, it will be necessary to implement some form of internet-based cluster-approach, which is probably best defined by the term 'grid computing'. Unfortunately, most ML libraries are still not natively designed for distributed computing (Meng, 2015).

A survey of economic models for grid brokers and schedulers was conducted in (Abramson, Buyya, and Giddy, 2002). Although interesting in the long run, we will use a simple queue mechanism to demonstrate our platform.

(Huang et al., 2006) argues that VM's for cloud computing have desirable properties including security, isolation and ease of configuration, but that the overhead of having to start/stop/migrate those instances impacts performance. Graphinius' virtual machines will be completely lightweight and can be managed by opening/closing a browser tab.

A list of top 10 obstacles for cloud computing can be found in (Armbrust et al., 2010). Although an exhaustive comparison to our approach is not possible at this point and matter of our research, most of these arguments stem from the fact that cloud data security as well as scalability are usually in the hands of powerful vendors.

(Youseff, Butrico, and Da Silva, 2008) define an ontology of cloud computing encompassing five layers: hardware (fabric), software kernel (fabric management), cloud infrastructure (communication), cloud software environment (PaaS) as well as cloud application (SaaS, configuration without programming). This view is similar to our proposed pipeline

levels before, where the implementation of Graphinius will start at about level 3.

(Liu et al., 2012) and (Liu et al., 2014) describe a distributed, scientific workflow system for bioinformatics based on a platform called Galaxy and its deployment to Amazons EC2 service via the Globus Provisioning framework; it supports graphical composition of workflows without requiring programming knowledge, which is especially interesting for less tech-savvy researchers. The concept of Galaxy is a very interesting one, although the author believes that client-side, GPU-enabled computations in coordination with one another are the future of personal and institutional grids.

10.5 Heterogeneous data linkage

Many research fields comprise several sub-problems which are amenable to different machine learning approaches and feature their own, distinctive input data sets. Coming from various, different data sets featuring their distinct attribute domains, they probably are - via their time-, space- or other dimensions, interlinkable with one another. Usually, studies are only concerned about using a single one of those data sources and applying different methods to it. However, a more holistic approach would be to fuse those data sets along one or more dimensions (or any other meaningful ruleset) in order to achieve a richer representation of the underlying problem. A resulting data-set might take the form of a graph structure, in which individual entities from the originating sets are linked by meaningful connection rules, which in themselves will have to be learned.

10.6 Meta machine learning

There are many papers in the Machine Learning community, Artificial Intelligence Planning, Operations Research and Hyper Heuristics (mostly for the optimization of search problems). From those papers it is clear that this field has a very rich history and therefore offers a plethora of research venues to be explored. As one of the first papers regarding this topic, (Rice, 1975) defined the "Algorithm Selection Problem", which was first recognized as a meta learning problem by the machine learning community. He describes several spaces in which the Algorithm Selection Problem plays out:

- **The problem space**: This is the set of all possible input problems (datasets + desired result class).

- **The feature space**: Features of a specific problem or family of problems. In dermatological imaging those would be defined by the imaging method (laser-scan vs. stanza), the scale of the objects to be detected (single cells vs nevi) etc.
- **The algorithm space**: The set of all algorithms suitable for the specific problem (features) to be worked on.
- **The performance measures** (the metric space): The set of possible measurements that could describe the quality of a solution (runtime performance, accuracy, ..).
- **The criteria space**: The weighing of different performance measures considered for a particular solution.

Horvitz et al., 2001 proposes a Bayesian learning algorithm that takes as input the generator function (assuming a generative model) of problem instances, the structure of input problems before solving, as well as the runtime behavior of a solver during execution in order to compute a Bayesian (posterior) score. The model was trained to predict runtime, but experiments showed that classification accuracy of the trained models could be significantly higher than that of the respective marginal model.

Hutter et al., 2007 note that manually sifting through parameter spaces of optimization algorithms is tedious work best suited for automated exploration. Interestingly, they see the selection of suitable building blocks in an algorithmic sequence (e.g. a preprocessing phase) as the same problem as setting parameters.

Smith-Miles, 2008 proposes a meta-learning inspired framework for analyzing meta-heuristic algorithm performance by learning the correlation between fitness function evaluations (of a chosen lower-level algorithm) and search space characteristics. The sequence in this approach works as follows: Based on 1) definition of a problem and 2) its features, 3) different problem instances are instantiated (training data). 4) Applying different algorithms to those instances, 5) outcomes are measured by suitable performance metrics, thereby 6) gaining meta-knowledge about algorithmic performance. An experiment to learn those correlations by ANN showed that performance could be predicted rather accurately for different algorithms on instances of the Quadratic Assignment Problem.

In most scenarios concerning Graphinius we will be concerned with the selection of algorithmic components and their parameters based on the nature of the specific area of application, previously tackled problems, their features, available algorithms, preprocessing methods, parameters as well as performance measures.

10.7 Hyper heuristics

Burke et al., 2003 introduce hyper heuristics as being different from meta-heuristics in that they concern themselves with families / classes of problems rather than one specific problem domain. They search a heuristics space instead of a solution space and often the goal is finding general-purpose heuristics that do not need to be optimal, but rather good-enough, soon-enough, cheap-enough. They also describe the 'Domain barrier', a concept which refers to a hyper heuristics approach being oblivious of the underlying problem domain, but simply receiving a collection of low level heuristics and deciding which one(s) to use, based solely on past history of heuristics applied and objective function values returned.

From (Burke et al., 2010), we can take the idea of three 'dimensions' of the hyper heuristics recommendation problem:

Selection vs. generation, where selecting means applying existing heuristics to different (families of) problems, whereas generating new heuristics by assembling them from existing heuristic components (which first need to be decomposed from already known heuristics).

Construction vs. perturbation, where construction means starting with an algorithmic component and developing it iteratively by putting more and more components together. In contrast, perturbation means starting with a whole solution (or sequence of algorithms) and gradually transmuting it into a new solution.

Online vs. offline learning, where online means testing each new combination of algorithmic components on several problem instances to determine their fitness during the learning phase. This seems to be a good approach when immediate recommendation is not required and computing power is abundant. Offline learning, on the other hand, means selecting a ready-to-use (sequence of) heuristics and applying them to a set of training instances in the hope of gaining helpful insights from those experiments.

Although many research fields in themselves are not broad enough to be a suitable proving ground for hyper heuristic research, the Graphinius platform will provide us with meta-data about experiments in many diverse areas. We fully agree with Burke et al., 2013 who concludes that there is still little interaction between research communities, a problem whose solution could lead to the extension of algorithms to both new problem domains and new methodologies through cross-fertilization of ideas.

10.8 Algorithmic recommender

As a natural consequence of the previous sections, a recommender for graph theoretical Machine Learning pipelines would ideally take arbitrary problem instances and output a suitable processing pipeline which has worked well for similar problems in the past. In addition, the pipeline would also be directly configured in code and downloaded into a researcher's browser, making it available to immediate execution.

10.9 Interactive Machine Learning

Last (but certainly not least), the problem of interactive machine learning lies in enabling a human domain-expert to exercise influence over the internal optimization considerations of a ML algorithm. For instance, if we take the SaNGreeA algorithm example of Section 8.3, we can imagine multiplying the cost vector (representing the information loss of a data point's generalized quasi-identifiers) by an arbitrary weight vector, giving either one or the other data attribute precedence of being preserved. Following the approach in Figure 10.1, we could use an appropriate UI to intervene it that process: everytime the algorithm has to make a clustering decision, it would instead (with a certain probability) present the domain expert with that choice - and learn the human's preferences by taking into account their answers, thereby manipulating the weights of the cost vector. It would be interesting to see how anonymization results would change, not only in appearance, but especially in their resilience against background-knowledge attacks as a result of such customizations.

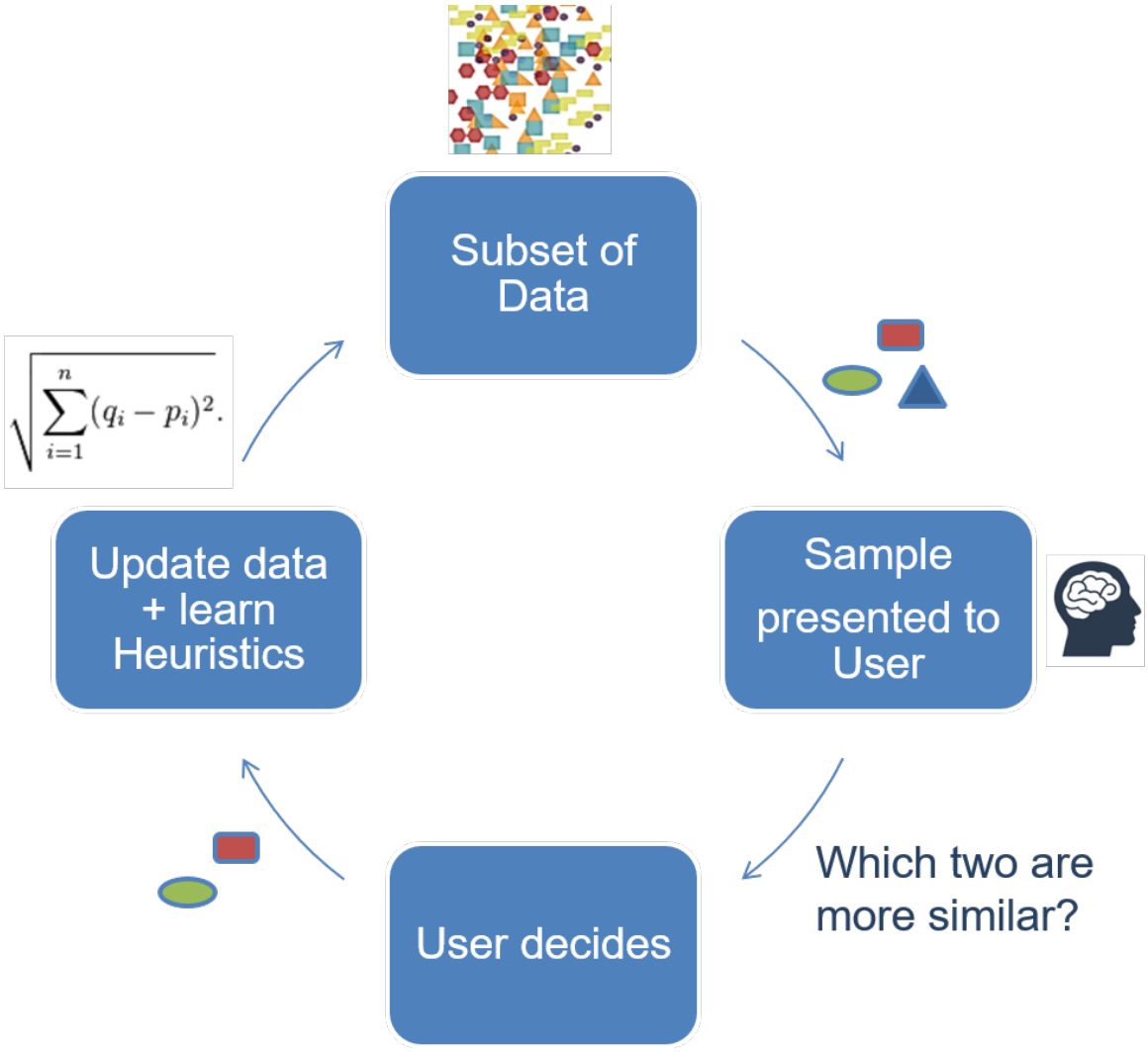


Figure 10.1: Anonymization augmented by IML (human in the loop)

11. Conclusion

In this thesis we have introduced the Graphinius platform for graph-theoretical Machine Learning experiments built on Web technologies, employing in-browser computations as well as visualization, focusing on a community centered approach.

After discussing some theoretical advantages such a platform could offer in the first chapter and delving into more specific descriptions of the theoretical underpinnings of potential application areas, we also took an interest in how such a product could be marketed, what business models would support it, and where the main advantages lie over its foreseeable competitors.

A generic description of the platform characteristics was given, followed by an in-depth survey of the modern web development cycle, its components and supporting infrastructure. Sifting through a wealth of open source alternatives, we finally decided on how to build every component of the Graphinius ecosystem including the base library, visualization, and communication (history) module.

We saw how three different use cases, theoretically tackled in earlier chapters, can be implemented using Graphinius and presented the output of their respective computations.

Following the presentation of some implementation metrics, such as size of the codebase and performance measurements on different graphs, we finally conducted a rather extensive sweep of interesting challenges and the promise emerging technologies hold for future developments of the platform.

The Graphinius development is still in its early phases, with just a groundwork having been laid as of May, 2016. Remembering the first conception of Graphinius only about six months earlier however, the author is remarkably pleased with the progress and dares to take a very optimistic look into the future.

List of Figures

Listings

Bibliography

Abramson, David, Rajkumar Buyya, and Jonathan Giddy (2002). "A computational economy for grid computing and its implementation in the Nimrod-G resource broker". In: *Future Generation Computer Systems* 18.8, pp. 1061–1074. ISSN: 0167739X. DOI: 10.1016/S0167-739X(02)00085-7. arXiv: 0111048 [cs].

Aggarwal, Gagan et al. (2005). "Approximation algorithms for k-anonymity". In: *Journal of Privacy Technology (JOPT)*.

Analytics, Butler (2016). *10+ Machine Learning as a Service Platforms*. Butler Analytics. URL: http://www.butleranalytics.com/10-machine-learning-as-a-service-platforms/.

Archambault, Daniel and Helen C Purchase (2013). "The map in the mental map: Experimental results in dynamic graph drawing". In: *International Journal of Human-Computer Studies* 71.11, pp. 1044–1055.

Armbrust, Michael et al. (2010). "A view of cloud computing". In: *Communications of the ACM* 53.4, p. 50. ISSN: 00010782. DOI: 10.1145/1721654.1721672. arXiv: 0521865719978052186571.

Bader, David A and Kamesh Madduri (2006). "Gtgraph: A synthetic graph generator suite". In: *Atlanta, GA, February*.

Bellard, Fabrice (2015). *Javascript PC Emulator*. Fabrice Bellard. URL: http://bellard.org/jslinux/tech.html.

Biezemans, Julien (2016). *Cucumber for JavaScript*. cucumber.io. URL: https://github.com/cucumber/cucumber-js.

Bobylev, A and S Rjasanow (2014). "Universit at des Saarlandes". In: *math.uni-sb.de*. URL: http://www.math.uni-sb.de/PREPRINTS/preprint03.ps.gz.

Burke, Edmund et al. (2003). *Hyper-Heuristics: An Emerging Direction in Modern Search Technology*. Ed. by Fred Glover and GaryA Kochenberger. New York: Springer, pp. 457–474. DOI: 10.1007/0-306-48056-5_16. URL: http://dx.doi.org/10.1007/0-306-48056-5_16.

Burke, Edmund K et al. (2010). "A Classification of Hyper-heuristics Approaches". In: *Handbook of Metaheuristics* 57, pp. 449–468. ISSN: 0884-8289. DOI: doi:10.1007/978-1-4419-1665-5_15.

Burke, Edmund K et al. (2013). “Hyper-heuristics: a survey of the state of the art”. In: *Journal of the Operational Research Society* 64.12, pp. 1695–1724. ISSN: 0160-5682. DOI: 10.1057/jors.2013.71. URL: http://www.palgrave-journals.com/doifinder/10.1057/jors.2013.71.

Campan, Alina and Traian Marius Truta (2009). “Data and structural k-anonymity in social networks”. In: *Privacy, Security, and Trust in KDD*. Springer, pp. 33–54.

Cerri, Andrea, Barbara Di Fabio, and Filippo Medri (2012). “Multi-scale approximation of the matching distance for shape retrieval”. In: *Computational Topology in Image Context*, pp. 1–10. URL: http://link.springer.com/chapter/10.1007/978-3-642-30238-1_14.

Chester, Sean et al. (2011). “k-Anonymization of Social Networks by Vertex Addition.” In: *ADBIS (2)* 789, pp. 107–116.

Ciriani, Valentina et al. (2007). “κ-anonymity”. In: *Secure data management in decentralized systems*. Springer, pp. 323–353.

Das Modak, Kaustav (2016). *Choosing a JavaScript Documentation Generator – JSDoc vs YUIDoc vs Doxx vs Docco*. FusionBrew. URL: http://www.fusioncharts.com/blog/2013/12/jsdoc-vs-yuidoc-vs-doxx-vs-docco-choosing-a-javascript-documentation-generator/.

Demetz, Oliver, David Hafner, and Joachim Weickert (2013). “The Complete Rank Transform: A Tool for Accurate and Morphologically Invariant Matching of Structures”. In: *Procedings of the British Machine Vision Conference 2013*, pp. 50.1–50.11. DOI: 10.5244/C.27.50. URL: http://www.bmva.org/bmvc/2013/Papers/paper0050/index.html.

Di Fabio, Barbara and Claudia Landi (2012). “Persistent homology and partial similarity of shapes”. In: *Pattern Recognition Letters* 33.11, pp. 1445–1450. ISSN: 01678655. DOI: 10.1016/j.patrec.2011.11.003. URL: http://linkinghub.elsevier.com/retrieve/pii/S0167865511003783.

Dohmen L., Edlich I.S. and M. Hackstein (2014). “A Declarative Web Framework for the Server-side Extension of the Multi Model Database ArangoDB”. In:

Engelschall, Ralf S. (2016). *ECMAScript 6 - New Features: Overview & Comparison*. es6-features.org. URL: http://es6-features.org/.

Felzenszwalb, Pedro F and Daniel P Huttenlocher (2004). “Efficient graph-based image segmentation”. In: *International Journal of Computer Vision* 59.2, pp. 167–181. DOI: 10.1023/B:VISI.0000022288.19776.77.

Hahn, Evan (2013). *JavaScript Testing with Jasmine*. " O’Reilly Media, Inc.".

Hampton, Catlin, Weizenbaum Natalie, and Chris Eppstein. (2016). *Sass Basics*. sass-lang.com. URL: http://sass-lang.com/guide.

Herman, I., G. Melançon, and M.S. Marshall (2000). "Graph visualization and navigation in information visualization: A survey". In: *IEEE Transactions on Visualization and Computer Graphics* 6.1, pp. 24–43.

Holzinger, Andreas, Bernd Malle, and Nicola Giuliani (2014). "On graph extraction from image data". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8609 LNAI, pp. 552–563. ISSN: 16113349. DOI: 10.1007/978-3-319-09891-3_50.

Holzinger, Andreas et al. (2014). "On the Generation of Point Cloud Data Sets: the first step in the Knowledge Discovery Process". In: *Interactive Knowledge Discovery and Data Mining: State-of-the-Art and Future Challenges in Biomedical Informatics, Springer Lecture Notes in Computer Science LNCS 8401*. Berlin, Heidelberg: Springer, pp. 57–80.

Horvitz, Eric et al. (2001). "A Bayesian Approach to Tackling Hard Computational Problems (Preliminary Report)". In: *Electronic Notes in Discrete Mathematics* 9, pp. 376–391. ISSN: 15710653. DOI: 10.1016/S1571-0653(04)00335-X.

Huang, Wei et al. (2006). "A Case for High Performance Computing with Virtual Machines". In: *Proceedings of the 20th annual international conference on supercomputing ICS 06*, pp. 125–134. DOI: 10.1145/1183401.1183421.

Hutter, Frank et al. (2007). "Automatic Algorithm Configuration based on Local Search". In: *BMC Bioinformatics*, pp. 1152–1157.

Kang, U and Duen Horng (2010). "Inference of beliefs on billion-scale graphs". In: URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.188.5276.

Kapron, Bruce, Gautam Srivastava, and S Venkatesh (2011). "Social network anonymization via edge addition". In: *Advances in Social Networks Analysis and Mining (ASONAM), 2011 International Conference on*. IEEE, pp. 155–162.

Kasaiezadeh, A. and A. Khajepour (2013). "Multi-agent stochastic level set method in image segmentation". In: *Computer Vision and Image Understanding* 117.9, pp. 1147–1162.

Lee, Yong Jae and Kristen Grauman (2012). "Object-graphs for context-aware visual category discovery." In: *IEEE transactions on pattern analysis and machine intelligence* 34.2, pp. 346–58. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2011.122. URL: http://www.ncbi.nlm.nih.gov/pubmed/21670480.

Leskovec, Jure, Ajit Singh, and Jon Kleinberg (2006). "Patterns of Influence in a Recommendation Network". In: *Proceedings of the 10th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. PAKDD'06. Singapore: Springer-Verlag, pp. 380–389. ISBN: 3-540-33206-5, 978-3-540-33206-0. DOI: 10.1007/11731139_44. URL: http://dx.doi.org/10.1007/11731139_44.

Liu, Bo et al. (2012). "Deploying Bioinformatics Workflows on clouds with galaxy and globus provision". In: *Proceedings - 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC 2012*, pp. 1087–1095. DOI: 10.1109/SC.Companion.2012.131.

Liu, Bo et al. (2014). "Cloud-based bioinformatics workflow platform for large-scale next-generation sequencing analyses". In: *Journal of Biomedical Informatics* 49, pp. 119–133. ISSN: 15320464. DOI: 10.1016/j.jbi.2014.01.005. URL: http://dx.doi.org/10.1016/j.jbi.2014.01.005.

Lorica, Ben (2013a). *Data Analysis: Just one component of the Data Science workflow*. O'Reilly Media, Inc. URL: http://radar.oreilly.com/2013/09/data-analysis-just-one-component-of-the-data-science-workflow.html.

Lorica, Ben (2013b). *Data Science Tools: Fast, easy to use, and scalable*. O'Reilly Media, Inc. URL: http://radar.oreilly.com/2013/03/fast-easy-to-use-scalable-data-science-tools.html.

Lorica, Ben (2013c). *Data scientists tackle the analytic lifecycle*. O'Reilly Media, Inc. URL: http://radar.oreilly.com/2013/07/data-scientists-and-the-analytic-lifecycle.html.

Meng, Xiangrui (2015). *ML Pipelines: A New High-Level API for MLlib*. Databricks Inc. URL: https://databricks.com/blog/2015/01/07/ml-pipelines-a-new-high-level-api-for-mllib.html.

Olfati-Saber, R., J. A. Fax, and R. M. Murray (2007). "Consensus and cooperation in networked multi-agent systems". In: *Proceedings of the IEEE* 95.1, pp. 215–233.

Page, Lukas and Max Mikhailov (2016). *An overview of Less, how to download and use, examples and more.* lesscss.org. URL: http://lesscss.org/.

Pandit, Shashank et al. (2007). "Netprobe: a fast and scalable system for fraud detection in online auction networks". In: *Proceedings of the 16th ...* 42, pp. 210, 201. DOI: 10.1145/1242572.1242600. URL: http://dx.doi.org/10.1145/1242572.1242600$\backslash$nhttp://dl.acm.org/citation.cfm?id=1242600.

Purchase, Helen (1997). "Which aesthetic has the greatest effect on human understanding?" In: *Graph Drawing, Lecture Notes in Computer Science LNCS 1353*. Ed. by Giuseppe DiBattista. Berlin Heidelberg: Springer, pp. 248–261.

Rice, John R (1975). "The algorithm selection problem". In: *Advances in Computers* 15, pp. 65–117.

Schneevoigt, Timm, Christopher Schroers, and Joachim Weickert (2014). "A Dense Pipeline for 3D Reconstruction from Image Sequences". In: *Pattern Recognition* 8753, pp. 629–640. URL: http://link.springer.com/chapter/10.1007/978-3-319-11752-2_52.

Sculley, D. et al. (2014). "Machine Learning: The High Interest Credit Card of Technical Debt". In: *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*.

Smith-Miles, Kate a. (2008). "Towards insightful algorithm selection for optimisation using meta-learning concepts". In: *Proceedings of the International Joint Conference on Neural Networks*, pp. 4118–4124. ISSN: 1098-7576. DOI: 10.1109/IJCNN.2008.4634391.

Sparks, Evan (2014). *ML Pipelines*. UC Berkeley. URL: https://amplab.cs.berkeley.edu/ml-pipelines/.

Stahl, F. et al. (2013). "An overview of interactive visual data mining techniques for knowledge discovery". In: *Wiley Interdisciplinary Reviews-Data Mining and Knowledge Discovery* 3.4, pp. 239–256.

Sweeney, Latanya (2002). "k-anonymity: A model for protecting privacy". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10.05, pp. 557–570.

Tang, David (2015). *Jasmine vs. Mocha, Chai, and Sinon*. THE JS GUY. URL: http://thejsguy.com/2015/01/12/jasmine-vs-mocha-chai-and-sinon.html.

Wagner, Israel A and Alfred M Bruckstein (2001). "From ants to a (ge) nts: A special issue on ant-robotics". In: *Annals of Mathematics and Artificial Intelligence* 31.1, pp. 1–5.

Yee, Bennet et al. (2009). "Native client: A sandbox for portable, untrusted x86 native code". In: *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, pp. 79–93.

Youseff, Lamia, Maria Butrico, and Dilma Da Silva (2008). "Toward a unified ontology of cloud computing". In: *Grid Computing Environments Workshop, GCE 2008*. ISSN: 15347362. DOI: 10.1109/GCE.2008.4738443.

A. Anonymization Table

Age range	Workclass	Country	Gender	Race	Marital status
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
39	*	United-States	Male	White	Never-married
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[50 - 53]	Self	United-States	Male	White	Married-civ-spouse
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[38 - 50]	Private	United-States	Male	White	Divorced
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[53 - 60]	Private	United-States	Male	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse

Age range	Workclass	Country	Gender	Race	Marital status
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[28 - 46]	Private	America	Female	*	Married-civ-spouse
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[49 - 54]	Private	America	Female	*	*
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[31 - 36]	Private	United-States	Female	White	Never-married
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[42 - 45]	Private	United-States	Male	White	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse

Age range	Workclass	Country	Gender	Race	Marital status
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[37 - 38]	Private	United-States	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[30 - 49]	*	*	Male	*	Married-civ-spouse
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[23 - 25]	Private	United-States	Female	White	Never-married
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[32 - 41]	*	United-States	Male	Black	*
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[25 - 27]	Self	United-States	Male	White	Never-married
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced

Age range	Workclass	Country	Gender	Race	Marital status
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced
[43 - 44]	Self	United-States	Female	White	Divorced
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[59 - 66]	*	United-States	Female	White	*
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[56 - 64]	*	United-States	Male	White	Married-civ-spouse
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[19 - 22]	Private	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[23 - 28]	*	United-States	Male	White	Never-married
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse

Age range	Workclass	Country	Gender	Race	Marital status
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[22 - 41]	Gov	United-States	Male	*	Married-civ-spouse
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[48 - 70]	Private	America	Male	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[19 - 36]	Private	United-States	Female	White	*
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[31 - 35]	Private	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse

Age range	Workclass	Country	Gender	Race	Marital status
[24 - 40]	Self	United-States	Male	White	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
[57 - 76]	*	North-America	Male	*	Married-civ-spouse
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
50	*	America	Male	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
46	*	*	*	White	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[29 - 42]	Private	*	*	*	*
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*

Age range	Workclass	Country	Gender	Race	Marital status
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*
[79 - 90]	*	*	*	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
43	*	America	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*
[28 - 34]	Private	*	Female	*	*

B. GraphiniusJS API

The following section provides the API documentation automatically extracted from the source code utilizing the TypeDoc library.

Graphinius

Generic graph (analysis) library in Typescript

Index

External modules

External modules

"core/Edges"

```
"core/Edges":
```

Defined in core/Edges.ts:1

BaseEdge

```
BaseEdge:
```

Defined in core/Edges.ts:47

constructor

```
new BaseEdge(_id: any, _node_a: IBaseNode, _node_b: IBaseNode, options?: EdgeConstructorOptions): BaseEdge
```

Defined in core/Edges.ts:51

Parameters

- **_id:** *any*
- **_node_a:** *IBaseNode*
- **_node_b:** *IBaseNode*
- Optional **options:** *EdgeConstructorOptions*

Returns *BaseEdge*

_directed

```
_directed: boolean
```

Defined in core/Edges.ts:48

_id

```
_id: any
```

Defined in core/Edges.ts:53

_label

```
_label: string
```

Defined in core/Edges.ts:51

_node_a

```
_node_a: IBaseNode
```

Defined in core/Edges.ts:54

_node_b

```
_node_b: IBaseNode
```

Defined in core/Edges.ts:55

_weight

```
_weight: number
```

Defined in core/Edges.ts:50

_weighted

```
_weighted: boolean
```

Defined in core/Edges.ts:49

getID

```
getID(): string
```

Implementation of IBaseEdge.getID
Defined in core/Edges.ts:66

Returns *string*

getLabel

```
getLabel(): string
```

Implementation of IBaseEdge.getLabel
Defined in core/Edges.ts:70

Returns *string*

getNodes

```
getNodes(): IConnectedNodes
```

Implementation of IBaseEdge.getNodes
Defined in core/Edges.ts:97

Returns *IConnectedNodes*

getWeight

```
getWeight(): number
```

Implementation of IBaseEdge.getWeight
Defined in core/Edges.ts:86

Returns *number*

isDirected

```
isDirected(): boolean
```

Implementation of IBaseEdge.isDirected
Defined in core/Edges.ts:78

Returns *boolean*

isWeighted

```
isWeighted(): boolean
```

Implementation of IBaseEdge.isWeighted
Defined in core/Edges.ts:82

Returns *boolean*

setLabel

```
setLabel(label: string): void
```

Implementation of IBaseEdge.setLabel
Defined in core/Edges.ts:74

Parameters

- **label:** *string*

Returns *void*

setWeight

```
setWeight(w: number): void
```

Implementation of IBaseEdge.setWeight
Defined in core/Edges.ts:90

Parameters

- **w:** *number*

Returns *void*

EdgeConstructorOptions

```
EdgeConstructorOptions:
```

Defined in core/Edges.ts:40

directed

```
directed: boolean
```

Defined in core/Edges.ts:41

label

```
label: string
```

Defined in core/Edges.ts:44

weight

```
weight: number
```

Defined in core/Edges.ts:43

weighted

```
weighted: boolean
```

Defined in core/Edges.ts:42

IBaseEdge

```
IBaseEdge:
```

Defined in core/Edges.ts:15

Edges are the most basic components in graphinius. They control no other elements below them, but hold references to the nodes they are connecting...

param internal id, public

param edge label, public

getID

```
getID(): string
```

Defined in core/Edges.ts:16

Returns *string*

getLabel

```
getLabel(): string
```

Defined in core/Edges.ts:17

Returns *string*

getNodes

```
getNodes(): IConnectedNodes
```

Defined in core/Edges.ts:29

Returns *IConnectedNodes*

getWeight

```
getWeight(): number
```

Defined in core/Edges.ts:25

Returns *number*

isDirected

```
isDirected(): boolean
```

Defined in core/Edges.ts:21

Returns *boolean*

isWeighted

```
isWeighted(): boolean
```

Defined in core/Edges.ts:24

Returns *boolean*

setLabel

```
setLabel(label: string): void
```

Defined in core/Edges.ts:18

Parameters

- **label:** *string*

Returns *void*

setWeight

```
setWeight(w: number): void
```

Defined in core/Edges.ts:26

Parameters

- **w:** *number*

Returns *void*

IConnectedNodes

```
IConnectedNodes:
```

Defined in core/Edges.ts:3

a

```
a: IBaseNode
```

Defined in core/Edges.ts:4

b

```
b: IBaseNode
```

Defined in core/Edges.ts:5

UNDIRECTED

```
UNDIRECTED:
```

Defined in core/Graph.ts:11

"core/Graph"

```
"core/Graph":
```

Defined in core/Graph.ts:1

GraphMode

```
GraphMode:
```

Defined in core/Graph.ts:8

DIRECTED

```
DIRECTED:
```

Defined in core/Graph.ts:10

INIT

```
INIT:
```

Defined in core/Graph.ts:9

MIXED

```
MIXED:
```

Defined in core/Graph.ts:12

BaseGraph

```
BaseGraph:
```

Defined in core/Graph.ts:85

constructor

```
new BaseGraph(_label: any): BaseGraph
```

Defined in core/Graph.ts:92

Parameters

- **_label:** *any*

Returns *BaseGraph*

_dir_edges

```
_dir_edges: object
```

Defined in core/Graph.ts:91

Type declaration

- [**key:** *string*]: *IBaseEdge*

_label

```
_label: any
```

Implementation of IGraph._label
Defined in core/Graph.ts:99

_mode

```
_mode: GraphMode
```

Defined in core/Graph.ts:89

_nodes

```
_nodes: object
```

Defined in core/Graph.ts:90

Type declaration

- [**key:** *string*]: *IBaseNode*

_nr_dir_edges

```
_nr_dir_edges: number
```

Defined in core/Graph.ts:87

_nr_nodes

```
_nr_nodes: number
```

Defined in core/Graph.ts:86

_nr_und_edges

```
_nr_und_edges: number
```

Defined in core/Graph.ts:88

_und_edges

```
_und_edges: object
```

Defined in core/Graph.ts:92

Type declaration

- [**key:** *string*]: *IBaseEdge*

addEdge

```
addEdge(id: string, node_a: IBaseNode, node_b: IBaseNode,
opts?: EdgeConstructorOptions): IBaseEdge
```

Defined in core/Graph.ts:300

Parameters

- **id:** *string*
- **node_a:** *IBaseNode*
- **node_b:** *IBaseNode*
- Optional **opts:** *EdgeConstructorOptions*

Returns *IBaseEdge*

addEdgeByNodeIDs

```
addEdgeByNodeIDs(label: string, node_a_id: string,
node_b_id: string, opts?: object): IBaseEdge
```

Defined in core/Graph.ts:286

Parameters

- **label:** *string*
- **node_a_id:** *string*

- **node_b_id:** *string*
- Optional **opts:** *object*

Returns *IBaseEdge*

addNode

```
addNode(id: string, opts?: object): IBaseNode
```

Defined in core/Graph.ts:162

Parameters

- **id:** *string*
- Optional **opts:** *object*

Returns *IBaseNode*

checkConnectedNodeOrThrow

```
checkConnectedNodeOrThrow(node: IBaseNode): void
```

Defined in core/Graph.ts:532

Parameters

- **node:** *IBaseNode*

Returns *void*

clearAllDirEdges

```
clearAllDirEdges(): void
```

Implementation of IGraph.clearAllDirEdges
Defined in core/Graph.ts:423

Remove all the (un)directed edges in the graph

Returns *void*

clearAllEdges

```
clearAllEdges(): void
```

Implementation of IGraph.clearAllEdges
Defined in core/Graph.ts:435

Returns *void*

clearAllUndEdges

```
clearAllUndEdges(): void
```

Implementation of IGraph.clearAllUndEdges
Defined in core/Graph.ts:429

Returns *void*

createRandomEdgesProb

```
createRandomEdgesProb(probability: number, directed?: boolean): void
```

Implementation of IGraph.createRandomEdgesProb
Defined in core/Graph.ts:448

Simple edge generator: Go through all node combinations, and add an (un)directed edge with

direction — true or false CAUTION: this algorithm takes quadratic runtime in #nodes

Parameters

- **probability:** *number*

  and

- Optional **directed:** *boolean*

Returns *void*

createRandomEdgesSpan

```
createRandomEdgesSpan(min: number, max: number, directed?: boolean): void
```

Implementation of IGraph.createRandomEdgesSpan
Defined in core/Graph.ts:476

Simple edge generator: Go through all nodes, and add [min, max] (un)directed edges to a randomly chosen node CAUTION: this algorithm could take quadratic runtime in #nodes but should be much faster

Parameters

- **min:** *number*
- **max:** *number*
- Optional **directed:** *boolean*

Returns *void*

degreeDistribution

```
degreeDistribution(): DegreeDistribution
```

Implementation of IGraph.degreeDistribution
Defined in core/Graph.ts:118

We assume graphs in which no node has higher total degree than 65536

Returns *DegreeDistribution*

deleteAllEdgesOf

```
deleteAllEdgesOf(node: IBaseNode): void
```

Implementation of IGraph.deleteAllEdgesOf
Defined in core/Graph.ts:415

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteDirEdgesOf

```
deleteDirEdgesOf(node: IBaseNode): void
```

Implementation of IGraph.deleteDirEdgesOf
Defined in core/Graph.ts:388

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteEdge

```
deleteEdge(edge: IBaseEdge): void
```

Implementation of IGraph.deleteEdge
Defined in core/Graph.ts:327

Parameters

- **edge:** *IBaseEdge*

Returns *void*

deleteInEdgesOf

```
deleteInEdgesOf(node: IBaseNode): void
```

Implementation of IGraph.deleteInEdgesOf
Defined in core/Graph.ts:354

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteNode

```
deleteNode(node: any): void
```

Implementation of IGraph.deleteNode
Defined in core/Graph.ts:209

Parameters

- **node:** *any*

Returns *void*

deleteOutEdgesOf

```
deleteOutEdgesOf(node: IBaseNode): void
```

Implementation of IGraph.deleteOutEdgesOf
Defined in core/Graph.ts:371

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteUndEdgesOf

```
deleteUndEdgesOf(node: IBaseNode): void
```

Implementation of IGraph.deleteUndEdgesOf
Defined in core/Graph.ts:394

Parameters

- **node:** *IBaseNode*

Returns *void*

getDirEdges

```
getDirEdges(): object
```

Implementation of IGraph.getDirEdges
Defined in core/Graph.ts:278

Returns *object*

- [**key:** *string*]: *IBaseEdge*

getEdgeById

```
getEdgeById(id: string): IBaseEdge
```

Implementation of IGraph.getEdgeById
Defined in core/Graph.ts:252

Parameters

- **id:** *string*

Returns *IBaseEdge*

getEdgeByLabel

```
getEdgeByLabel(label: string): IBaseEdge
```

Implementation of IGraph.getEdgeByLabel
Defined in core/Graph.ts:264

Use hasEdgeLabel with CAUTION -> it has LINEAR runtime in the graph's #edges

Parameters

- **label:** *string*

Returns *IBaseEdge*

getMode

```
getMode(): GraphMode
```

Implementation of IGraph.getMode
Defined in core/Graph.ts:101

Returns *GraphMode*

getNodeById

```
getNodeById(id: string): IBaseNode
```

Implementation of IGraph.getNodeById
Defined in core/Graph.ts:183

Parameters

- **id:** *string*

Returns *IBaseNode*

getNodeByLabel

```
getNodeByLabel(label: string): IBaseNode
```

Implementation of IGraph.getNodeByLabel
Defined in core/Graph.ts:191

Use getNodeByLabel with CAUTION -> it has LINEAR runtime in the graph's #nodes

Parameters

- **label:** *string*

Returns *IBaseNode*

getNodes

```
getNodes(): object
```

Implementation of IGraph.getNodes
Defined in core/Graph.ts:198

Returns *object*

- [**key:** *string*]: *IBaseNode*

getRandomDirEdge

```
getRandomDirEdge(): IBaseEdge
```

Implementation of IGraph.getRandomDirEdge
Defined in core/Graph.ts:520

CAUTION - This function is linear in # directed edges

Returns *IBaseEdge*

getRandomNode

```
getRandomNode(): IBaseNode
```

Implementation of IGraph.getRandomNode
Defined in core/Graph.ts:205

CAUTION - This function takes linear time in # nodes

Returns *IBaseNode*

getRandomUndEdge

```
getRandomUndEdge(): IBaseEdge
```

Implementation of IGraph.getRandomUndEdge
Defined in core/Graph.ts:527

CAUTION - This function is linear in # undirected edges

Returns *IBaseEdge*

getStats

```
getStats(): GraphStats
```

Implementation of IGraph.getStats
Defined in core/Graph.ts:105

Returns *GraphStats*

getUndEdges

```
getUndEdges(): object
```

Implementation of IGraph.getUndEdges
Defined in core/Graph.ts:282

Returns *object*

- [**key:** *string*]: *IBaseEdge*

hasEdgeID

```
hasEdgeID(id: string): boolean
```

Implementation of IGraph.hasEdgeID
Defined in core/Graph.ts:234

Parameters

- **id:** *string*

Returns *boolean*

hasEdgeLabel

```
hasEdgeLabel(label: string): boolean
```

Implementation of IGraph.hasEdgeLabel
Defined in core/Graph.ts:242

Use hasEdgeLabel with CAUTION -> it has LINEAR runtime in the graph's #edges

Parameters

- **label:** *string*

Returns *boolean*

hasNodeID

```
hasNodeID(id: string): boolean
```

Implementation of IGraph.hasNodeID
Defined in core/Graph.ts:169

Parameters

- **id:** *string*

Returns *boolean*

hasNodeLabel

```
hasNodeLabel(label: string): boolean
```

Implementation of IGraph.hasNodeLabel
Defined in core/Graph.ts:177

Use hasNodeLabel with CAUTION -> it has LINEAR runtime in the graph's #nodes

Parameters

- **label:** *string*

Returns *boolean*

nrDirEdges

```
nrDirEdges(): number
```

Implementation of IGraph.nrDirEdges
Defined in core/Graph.ts:154

Returns *number*

nrNodes

```
nrNodes(): number
```

Implementation of IGraph.nrNodes
Defined in core/Graph.ts:150

Returns *number*

nrUndEdges

```
nrUndEdges(): number
```

Implementation of IGraph.nrUndEdges
Defined in core/Graph.ts:158

Returns *number*

pickRandomProperty

```
pickRandomProperty(obj: any): any
```

Defined in core/Graph.ts:557

Parameters

- **obj:** *any*

Returns *any*

updateGraphMode

```
updateGraphMode(): void
```

Defined in core/Graph.ts:539

Returns *void*

DegreeDistribution

```
DegreeDistribution:
```

Defined in core/Graph.ts:16

all

```
all: Uint16Array
```

Defined in core/Graph.ts:21

dir

```
dir: Uint16Array
```

Defined in core/Graph.ts:19

in

```
in: Uint16Array
```

Defined in core/Graph.ts:17

out

```
out: Uint16Array
```

Defined in core/Graph.ts:18

und

```
und: Uint16Array
```

Defined in core/Graph.ts:20

GraphStats

```
GraphStats:
```

Defined in core/Graph.ts:25

mode

```
mode: GraphMode
```

Defined in core/Graph.ts:26

nr_dir_edges

```
nr_dir_edges: number
```

Defined in core/Graph.ts:29

nr_nodes

```
nr_nodes: number
```

Defined in core/Graph.ts:27

nr_und_edges

```
nr_und_edges: number
```

Defined in core/Graph.ts:28

IGraph

```
IGraph:
```

Defined in core/Graph.ts:34

_label

```
_label: string
```

Defined in core/Graph.ts:35

addEdge

```
addEdge(label: string, node_a: IBaseNode, node_b: IBaseNode, opts?: object): IBaseEdge
```

Defined in core/Graph.ts:53

Parameters

- **label:** *string*
- **node_a:** *IBaseNode*
- **node_b:** *IBaseNode*
- Optional **opts:** *object*

Returns *IBaseEdge*

addEdgeByNodeIDs

```
addEdgeByNodeIDs(label: string, node_a_id: string, node_b_id: string, opts?: object): IBaseEdge
```

Defined in core/Graph.ts:54

Parameters

- **label:** *string*
- **node_a_id:** *string*
- **node_b_id:** *string*
- Optional **opts:** *object*

Returns *IBaseEdge*

addNode

```
addNode(id: string, opts?: object): IBaseNode
```

Defined in core/Graph.ts:42

Parameters

- **id:** *string*
- Optional **opts:** *object*

Returns *IBaseNode*

clearAllDirEdges

```
clearAllDirEdges(): void
```

Defined in core/Graph.ts:75

Returns *void*

clearAllEdges

```
clearAllEdges(): void
```

Defined in core/Graph.ts:77

Returns *void*

clearAllUndEdges

```
clearAllUndEdges(): void
```

Defined in core/Graph.ts:76

Returns *void*

createRandomEdgesProb

```
createRandomEdgesProb(probability: number, directed?: boolean): void
```

Defined in core/Graph.ts:80

Parameters

- **probability:** *number*
- Optional **directed:** *boolean*

Returns *void*

createRandomEdgesSpan

```
createRandomEdgesSpan(min: number, max: number, directed?: boolean): void
```

Defined in core/Graph.ts:81

Parameters

- **min:** *number*
- **max:** *number*
- Optional **directed:** *boolean*

Returns *void*

degreeDistribution

```
degreeDistribution(): DegreeDistribution
```

Defined in core/Graph.ts:39

Returns *DegreeDistribution*

deleteAllEdgesOf

```
deleteAllEdgesOf(node: IBaseNode): void
```

Defined in core/Graph.ts:72

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteDirEdgesOf

```
deleteDirEdgesOf(node: IBaseNode): void
```

Defined in core/Graph.ts:70

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteEdge

```
deleteEdge(edge: IBaseEdge): void
```

Defined in core/Graph.ts:63

Parameters

- **edge:** *IBaseEdge*

Returns *void*

deleteInEdgesOf

```
deleteInEdgesOf(node: IBaseNode): void
```

Defined in core/Graph.ts:68

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteNode

```
deleteNode(node: any): void
```

Defined in core/Graph.ts:50

Parameters

- **node:** *any*

Returns *void*

deleteOutEdgesOf

```
deleteOutEdgesOf(node: IBaseNode): void
```

Defined in core/Graph.ts:69

Parameters

- **node:** *IBaseNode*

Returns *void*

deleteUndEdgesOf

```
deleteUndEdgesOf(node: IBaseNode): void
```

Defined in core/Graph.ts:71

Parameters

- **node:** *IBaseNode*

Returns *void*

getDirEdges

```
getDirEdges(): object
```

Defined in core/Graph.ts:59

Returns *object*

- [**key:** *string*]: *IBaseEdge*

getEdgeById

```
getEdgeById(id: string): IBaseEdge
```

Defined in core/Graph.ts:57

Parameters

- **id:** *string*

Returns *IBaseEdge*

getEdgeByLabel

```
getEdgeByLabel(label: string): IBaseEdge
```

Defined in core/Graph.ts:58

Parameters

- **label:** *string*

Returns *IBaseEdge*

getMode

```
getMode(): GraphMode
```

Defined in core/Graph.ts:37

Returns *GraphMode*

getNodeById

```
getNodeById(id: string): IBaseNode
```

Defined in core/Graph.ts:45

Parameters

- **id:** *string*

Returns *IBaseNode*

getNodeByLabel

```
getNodeByLabel(label: string): IBaseNode
```

Defined in core/Graph.ts:46

Parameters

- **label:** *string*

Returns *IBaseNode*

getNodes

```
getNodes(): object
```

Defined in core/Graph.ts:47

Returns *object*

- [**key:** *string*]: *IBaseNode*

getRandomDirEdge

```
getRandomDirEdge(): IBaseEdge
```

Defined in core/Graph.ts:64

Returns *IBaseEdge*

getRandomNode

```
getRandomNode(): IBaseNode
```

Defined in core/Graph.ts:49

Returns *IBaseNode*

getRandomUndEdge

```
getRandomUndEdge(): IBaseEdge
```

Defined in core/Graph.ts:65

Returns *IBaseEdge*

getStats

```
getStats(): GraphStats
```

Defined in core/Graph.ts:38

Returns *GraphStats*

getUndEdges

```
getUndEdges(): object
```

Defined in core/Graph.ts:60

Returns *object*

- [**key:** *string*]: *IBaseEdge*

hasEdgeID

```
hasEdgeID(id: string): boolean
```

Defined in core/Graph.ts:55

Parameters

- **id:** *string*

Returns *boolean*

hasEdgeLabel

```
hasEdgeLabel(label: string): boolean
```

Defined in core/Graph.ts:56

Parameters

- **label:** *string*

Returns *boolean*

hasNodeID

```
hasNodeID(id: string): boolean
```

Defined in core/Graph.ts:43

Parameters

- **id:** *string*

Returns *boolean*

hasNodeLabel

```
hasNodeLabel(label: string): boolean
```

Defined in core/Graph.ts:44

Parameters

- **label:** *string*

Returns *boolean*

nrDirEdges

```
nrDirEdges(): number
```

Defined in core/Graph.ts:61

Returns *number*

nrNodes

```
nrNodes(): number
```

Defined in core/Graph.ts:48

Returns *number*

nrUndEdges

```
nrUndEdges(): number
```

Defined in core/Graph.ts:62

Returns *number*

"core/Nodes"

```
"core/Nodes":
```

Defined in core/Nodes.ts:1

BaseNode

```
BaseNode:
```

Defined in core/Nodes.ts:59

constructor

```
new BaseNode(_id: any, features?: object): BaseNode
```

Defined in core/Nodes.ts:81

Parameters

- **_id:** *any*
- Optional **features:** *object*
  - [**k:** *string*]: *any*

Returns *BaseNode*

_features

```
_features: object
```

Defined in core/Nodes.ts:68

Type declaration

- [**k:** *string*]: *any*

_id

```
_id: any
```

Defined in core/Nodes.ts:83

_in_degree

```
_in_degree: number
```

Defined in core/Nodes.ts:64

degrees - let's hold them separate in order to avoid Object.keys(...)

_in_edges

```
_in_edges: object
```

Defined in core/Nodes.ts:78

Design decision: Do we only use ONE _edges hash - OR - separate hashes for _in_edges, _out_edges, _und_edges As getting edges based on their type during the execution of graph algorithms is pretty common, it's logical to separate the structures.

Type declaration

- [**k:** *string*]: *IBaseEdge*

_label

```
_label: string
```

Defined in core/Nodes.ts:81

_out_degree

```
_out_degree: number
```

Defined in core/Nodes.ts:65

_out_edges

```
_out_edges: object
```

Defined in core/Nodes.ts:79

Type declaration

- [**k:** *string*]: *IBaseEdge*

_und_degree

```
_und_degree: number
```

Defined in core/Nodes.ts:66

_und_edges

```
_und_edges: object
```

Defined in core/Nodes.ts:80

Type declaration

- [**k:** *string*]: *IBaseEdge*

addEdge

```
addEdge(edge: IBaseEdge): void
```

Implementation of IBaseNode.addEdge
Defined in core/Nodes.ts:167

We have to:

1. throw an error if the edge is already attached
2. add it to the edge array
3. check type of edge (directed / undirected)
4. update our degrees accordingly This is a design decision we can defend by pointing out that querying degrees will occur much more often than modifying the edge structure of a node (??) One further point: do we also check for duplicate edges not in the sense of duplicate ID's but duplicate structure (nodes, direction) ? => Not for now, as we would have to check every edge instead of simply checking the hash id... ALTHOUGH: adding edges will (presumably) not occur often...

Parameters

- **edge:** *IBaseEdge*

Returns *void*

adjNodes

```
adjNodes(): Array<NeighborEntry>
```

Implementation of IBaseNode.adjNodes
Defined in core/Nodes.ts:363

Returns *Array<NeighborEntry>*

allEdges

```
allEdges(): object
```

Implementation of IBaseNode.allEdges
Defined in core/Nodes.ts:234

Returns *object*

clearEdges

```
clearEdges(): void
```

Implementation of IBaseNode.clearEdges
Defined in core/Nodes.ts:296

Returns *void*

clearFeatures

```
clearFeatures(): void
```

Implementation of IBaseNode.clearFeatures
Defined in core/Nodes.ts:134

Returns *void*

clearInEdges

```
clearInEdges(): void
```

Implementation of IBaseNode.clearInEdges
Defined in core/Nodes.ts:286

Returns *void*

clearOutEdges

```
clearOutEdges(): void
```

Implementation of IBaseNode.clearOutEdges
Defined in core/Nodes.ts:281

Returns *void*

clearUndEdges

```
clearUndEdges(): void
```

Implementation of IBaseNode.clearUndEdges
Defined in core/Nodes.ts:291

Returns *void*

connNodes

```
connNodes(): Array<NeighborEntry>
```

Implementation of IBaseNode.connNodes
Defined in core/Nodes.ts:336

Returns *Array<NeighborEntry>*

degree

```
degree(): number
```

Implementation of IBaseNode.degree
Defined in core/Nodes.ts:147

Returns *number*

deleteFeature

```
deleteFeature(key: string): any
```

Implementation of IBaseNode.deleteFeature
Defined in core/Nodes.ts:125

Parameters

- **key:** *string*

Returns *any*

dirEdges

```
dirEdges(): object
```

Implementation of IBaseNode.dirEdges
Defined in core/Nodes.ts:230

Returns *object*

getEdge

```
getEdge(id: string): IBaseEdge
```

Implementation of IBaseNode.getEdge
Defined in core/Nodes.ts:210

Parameters

- **id:** *string*

Returns *IBaseEdge*

getFeature

```
getFeature(key: string): any
```

Implementation of IBaseNode.getFeature
Defined in core/Nodes.ts:109

Parameters

- **key:** *string*

Returns *any*

getFeatures

```
getFeatures(): object
```

Implementation of IBaseNode.getFeatures
Defined in core/Nodes.ts:105

Returns *object*

- [**k:** *string*]: *any*

getID

```
getID(): string
```

Implementation of IBaseNode.getID
Defined in core/Nodes.ts:93

Returns *string*

getLabel

```
getLabel(): string
```

Implementation of IBaseNode.getLabel
Defined in core/Nodes.ts:97

Returns *string*

hasEdge

```
hasEdge(edge: IBaseEdge): boolean
```

Implementation of IBaseNode.hasEdge
Defined in core/Nodes.ts:202

Parameters

- **edge:** *IBaseEdge*

Returns *boolean*

hasEdgeID

```
hasEdgeID(id: string): boolean
```

Implementation of IBaseNode.hasEdgeID
Defined in core/Nodes.ts:206

Parameters

- **id:** *string*

Returns *boolean*

inDegree

```
inDegree(): number
```

Implementation of IBaseNode.inDegree
Defined in core/Nodes.ts:139

Returns *number*

inEdges

```
inEdges(): object
```

Implementation of IBaseNode.inEdges
Defined in core/Nodes.ts:218

Returns *object*

- [**k:** *string*]: *IBaseEdge*

nextNodes

```
nextNodes(): Array<NeighborEntry>
```

Implementation of IBaseNode.nextNodes
Defined in core/Nodes.ts:319

Returns *Array<NeighborEntry>*

outDegree

```
outDegree(): number
```

Implementation of IBaseNode.outDegree
Defined in core/Nodes.ts:143

Returns *number*

outEdges

```
outEdges(): object
```

Implementation of IBaseNode.outEdges
Defined in core/Nodes.ts:222

Returns *object*

- [**k:** *string*]: *IBaseEdge*

prevNodes

```
prevNodes(): Array<NeighborEntry>
```

Implementation of IBaseNode.prevNodes
Defined in core/Nodes.ts:302

Returns *Array<NeighborEntry>*

removeEdge

```
removeEdge(edge: IBaseEdge): void
```

Implementation of IBaseNode.removeEdge
Defined in core/Nodes.ts:238

Parameters

- **edge:** *IBaseEdge*

Returns *void*

removeEdgeID

```
removeEdgeID(id: string): void
```

Implementation of IBaseNode.removeEdgeID
Defined in core/Nodes.ts:260

Parameters

- **id:** *string*

Returns *void*

setFeature

```
setFeature(key: string, value: any): void
```

Implementation of IBaseNode.setFeature
Defined in core/Nodes.ts:121

Parameters

- **key:** *string*
- **value:** *any*

Returns *void*

setFeatures

```
setFeatures(features: object): void
```

Defined in core/Nodes.ts:117

Parameters

- **features:** *object*
  - [**k:** *string*]: *any*

Returns *void*

setLabel

```
setLabel(label: string): void
```

Implementation of IBaseNode.setLabel
Defined in core/Nodes.ts:101

Parameters

- **label:** *string*

Returns *void*

undEdges

```
undEdges(): object
```

Implementation of IBaseNode.undEdges
Defined in core/Nodes.ts:226

Returns *object*

- [**k:** *string*]: *IBaseEdge*

IBaseNode

```
IBaseNode:
```

Defined in core/Nodes.ts:11

addEdge

```
addEdge(edge: IBaseEdge): void
```

Defined in core/Nodes.ts:30

Parameters

- **edge:** *IBaseEdge*

Returns *void*

adjNodes

```
adjNodes(): Array<NeighborEntry>
```

Defined in core/Nodes.ts:55

Returns *Array<NeighborEntry>*

allEdges

```
allEdges(): object
```

Defined in core/Nodes.ts:40

Returns *object*

clearEdges

```
clearEdges(): void
```

Defined in core/Nodes.ts:49

Returns *void*

clearFeatures

```
clearFeatures(): void
```

Defined in core/Nodes.ts:22

Returns *void*

clearInEdges

```
clearInEdges(): void
```

Defined in core/Nodes.ts:47

Returns *void*

clearOutEdges

```
clearOutEdges(): void
```

Defined in core/Nodes.ts:46

Returns *void*

clearUndEdges

```
clearUndEdges(): void
```

Defined in core/Nodes.ts:48

Returns *void*

connNodes

```
connNodes(): Array<NeighborEntry>
```

Defined in core/Nodes.ts:54

Returns *Array<NeighborEntry>*

degree

```
degree(): number
```

Defined in core/Nodes.ts:27

Returns *number*

deleteFeature

```
deleteFeature(key: string): any
```

Defined in core/Nodes.ts:21

Parameters

- **key:** *string*

Returns *any*

dirEdges

```
dirEdges(): object
```

Defined in core/Nodes.ts:39

Returns *object*

getEdge

```
getEdge(id: string): IBaseEdge
```

Defined in core/Nodes.ts:33

Parameters

- **id:** *string*

Returns *IBaseEdge*

getFeature

```
getFeature(key: string): any
```

Defined in core/Nodes.ts:18

Parameters

- **key:** *string*

Returns *any*

getFeatures

```
getFeatures(): object
```

Defined in core/Nodes.ts:17

Returns *object*

- [**k:** *string*]: *any*

getID

```
getID(): string
```

Defined in core/Nodes.ts:12

Returns *string*

getLabel

```
getLabel(): string
```

Defined in core/Nodes.ts:13

Returns *string*

hasEdge

```
hasEdge(edge: IBaseEdge): boolean
```

Defined in core/Nodes.ts:31

Parameters

- **edge:** *IBaseEdge*

Returns *boolean*

hasEdgeID

```
hasEdgeID(id: string): boolean
```

Defined in core/Nodes.ts:32

Parameters

- **id:** *string*

Returns *boolean*

inDegree

```
inDegree(): number
```

Defined in core/Nodes.ts:25

Returns *number*

inEdges

```
inEdges(): object
```

Defined in core/Nodes.ts:35

Returns *object*

- [**k:** *string*]: *IBaseEdge*

nextNodes

```
nextNodes(): Array<NeighborEntry>
```

Defined in core/Nodes.ts:53

Returns *Array<NeighborEntry>*

outDegree

```
outDegree(): number
```

Defined in core/Nodes.ts:26

Returns *number*

outEdges

```
outEdges(): object
```

Defined in core/Nodes.ts:36

Returns *object*

- [**k:** *string*]: *IBaseEdge*

prevNodes

```
prevNodes(): Array<NeighborEntry>
```

Defined in core/Nodes.ts:52

Returns *Array<NeighborEntry>*

removeEdge

```
removeEdge(edge: IBaseEdge): void
```

Defined in core/Nodes.ts:42

Parameters

- **edge:** *IBaseEdge*

Returns *void*

removeEdgeID

```
removeEdgeID(id: string): void
```

Defined in core/Nodes.ts:43

Parameters

- **id:** *string*

Returns *void*

setFeature

```
setFeature(key: string, value: any): void
```

Defined in core/Nodes.ts:20

Parameters

- **key:** *string*
- **value:** *any*

Returns *void*

setFeatures

```
setFeatures(features: object): void
```

Defined in core/Nodes.ts:19

Parameters

- **features:** *object*
  - [**k:** *string*]: *any*

Returns *void*

setLabel

```
setLabel(label: string): void
```

Defined in core/Nodes.ts:14

Parameters

- **label:** *string*

Returns *void*

undEdges

```
undEdges(): object
```

Defined in core/Nodes.ts:37

Returns *object*

- [**k:** *string*]: *IBaseEdge*

NeighborEntry

```
NeighborEntry:
```

Defined in core/Nodes.ts:6

edge

```
edge: IBaseEdge
```

Defined in core/Nodes.ts:8

node

```
node: IBaseNode
```

Defined in core/Nodes.ts:7

"datastructs/binaryHeap"

```
"datastructs/binaryHeap":
```

Defined in datastructs/binaryHeap.ts:1

BinaryHeapMode

```
BinaryHeapMode:
```

Defined in datastructs/binaryHeap.ts:4

MAX

```
MAX:
```

Defined in datastructs/binaryHeap.ts:6

MIN

```
MIN:
```

Defined in datastructs/binaryHeap.ts:5

BinaryHeap

```
BinaryHeap:
```

Defined in datastructs/binaryHeap.ts:28

constructor

```
new BinaryHeap(_mode?: BinaryHeapMode, _evalPriority?: (Anonymous function), _evalObjID?: (Anonymous function)): BinaryHeap
```

Defined in datastructs/binaryHeap.ts:29

Mode of a min heap should only be set upon instantiation and never again afterwards...

Parameters

- Default value **_mode:** *BinaryHeapMode*
  = BinaryHeapMode.MIN

  MIN or MAX heap

- Default value **_evalPriority:** *(Anonymous function)*
  = (obj:any) => {if (typeof obj !== 'number' && typeof obj !== 'string') {return NaN;}return parseInt(obj)}

  the evaluation function applied to all incoming objects to determine it's score

- Default value **_evalObjID:** *(Anonymous function)*
  = (obj:any) => {return obj;}

  function to determine the identity of the object we are looking for at removal etc..

Returns *BinaryHeap*

_array

```
_array: Array<any>
```

Defined in datastructs/binaryHeap.ts:29

_evalObjID

```
_evalObjID: (Anonymous function)
```

Defined in datastructs/binaryHeap.ts:47

function to determine the identity of the object we are looking for at removal etc..

_evalPriority

```
_evalPriority: (Anonymous function)
```

Defined in datastructs/binaryHeap.ts:41

the evaluation function applied to all incoming objects to determine it's score

_mode

```
_mode: BinaryHeapMode
```

Defined in datastructs/binaryHeap.ts:40

MIN or MAX heap

evalInputObjID

```
evalInputObjID(obj: any): any
```

Implementation of IBinaryHeap.evalInputObjID
Defined in datastructs/binaryHeap.ts:76

Parameters

- **obj:** *any*

Returns *any*

evalInputPriority

```
evalInputPriority(obj: any): number
```

Implementation of IBinaryHeap.evalInputPriority
Defined in datastructs/binaryHeap.ts:68

Parameters

- **obj:** *any*

Returns *number*

getArray

```
getArray(): Array<any>
```

Implementation of IBinaryHeap.getArray
Defined in datastructs/binaryHeap.ts:56

Returns *Array<any>*

getEvalObjIDFun

```
getEvalObjIDFun(): Function
```

Implementation of IBinaryHeap.getEvalObjIDFun
Defined in datastructs/binaryHeap.ts:72

Returns *Function*

getEvalPriorityFun

```
getEvalPriorityFun(): Function
```

Implementation of IBinaryHeap.getEvalPriorityFun
Defined in datastructs/binaryHeap.ts:64

Returns *Function*

getMode

```
getMode(): BinaryHeapMode
```

Implementation of IBinaryHeap.getMode
Defined in datastructs/binaryHeap.ts:52

Returns *BinaryHeapMode*

insert

```
insert(obj: any): void
```

Implementation of IBinaryHeap.insert
Defined in datastructs/binaryHeap.ts:93

Insert - Adding an object to the heap

Parameters

- **obj:** *any*
  the obj to add to the heap

Returns *void*

the objects index in the internal array

orderCorrect

```
orderCorrect(obj_a: any, obj_b: any): boolean
```

Defined in datastructs/binaryHeap.ts:177

Parameters

- **obj_a:** *any*
- **obj_b:** *any*

Returns *boolean*

peek

```
peek(): any
```

Implementation of IBinaryHeap.peek
Defined in datastructs/binaryHeap.ts:80

Returns *any*

pop

```
pop(): any
```

Implementation of IBinaryHeap.pop
Defined in datastructs/binaryHeap.ts:84

Returns *any*

remove

```
remove(obj: any): any
```

Implementation of IBinaryHeap.remove
Defined in datastructs/binaryHeap.ts:102

Parameters

- **obj:** *any*

Returns *any*

size

```
size(): number
```

Implementation of IBinaryHeap.size
Defined in datastructs/binaryHeap.ts:60

Returns *number*

trickleDown

```
trickleDown(i: number): void
```

Defined in datastructs/binaryHeap.ts:127

Parameters

- **i:** *number*

Returns *void*

trickleUp

```
trickleUp(i: number): void
```

Defined in datastructs/binaryHeap.ts:159

Parameters

- **i:** *number*

Returns *void*

IBinaryHeap

```
IBinaryHeap:
```

Defined in datastructs/binaryHeap.ts:10

evalInputObjID

```
evalInputObjID(obj: any): any
```

Defined in datastructs/binaryHeap.ts:18

Parameters

- **obj:** *any*

Returns *any*

evalInputPriority

```
evalInputPriority(obj: any): number
```

Defined in datastructs/binaryHeap.ts:16

Parameters

- **obj:** *any*

Returns *number*

getArray

```
getArray(): Array<any>
```

Defined in datastructs/binaryHeap.ts:13

Returns *Array<any>*

getEvalObjIDFun

```
getEvalObjIDFun(): Function
```

Defined in datastructs/binaryHeap.ts:17

Returns *Function*

getEvalPriorityFun

```
getEvalPriorityFun(): Function
```

Defined in datastructs/binaryHeap.ts:15

Returns *Function*

getMode

```
getMode(): BinaryHeapMode
```

Defined in datastructs/binaryHeap.ts:12

Returns *BinaryHeapMode*

insert

```
insert(obj: any): void
```

Defined in datastructs/binaryHeap.ts:21

Parameters

- **obj:** *any*

Returns *void*

peek

```
peek(): any
```

Defined in datastructs/binaryHeap.ts:23

Returns *any*

pop

```
pop(): any
```

Defined in datastructs/binaryHeap.ts:24

Returns *any*

remove

```
remove(obj: any): any
```

Defined in datastructs/binaryHeap.ts:22

Parameters

- **obj:** *any*

Returns *any*

size

```
size(): number
```

Defined in datastructs/binaryHeap.ts:14

Returns *number*

"datastructs/fibonacciHeap"

```
"datastructs/fibonacciHeap":
```

Defined in datastructs/fibonacciHeap.ts:1

"input/CSVInput"

```
"input/CSVInput":
```

Defined in input/CSVInput.ts:1

CSVInput

```
CSVInput:
```

Defined in input/CSVInput.ts:27

constructor

```
new CSVInput(_separator?: string, _explicit_direction?:
boolean, _direction_mode?: boolean): CSVInput
```

Defined in input/CSVInput.ts:27

Parameters

- Default value **_separator:** *string* = ","
- Default value **_explicit_direction:** *boolean* = true
- Default value **_direction_mode:** *boolean* = false

Returns *CSVInput*

_direction_mode

```
_direction_mode: boolean
```

Implementation of ICSVInput._direction_mode
Defined in input/CSVInput.ts:31

_explicit_direction

```
_explicit_direction: boolean
```

Implementation of ICSVInput._explicit_direction
Defined in input/CSVInput.ts:30

_separator

```
_separator: string
```

Implementation of ICSVInput._separator
Defined in input/CSVInput.ts:29

checkNodeEnvironment

```
checkNodeEnvironment(): void
```

Defined in input/CSVInput.ts:207

Returns *void*

readFileAndReturn

```
readFileAndReturn(filepath: string, func: Function): IGraph
```

Defined in input/CSVInput.ts:86

Parameters

- **filepath:** *string*
- **func:** *Function*

Returns *IGraph*

readFromAdjacencyList

```
readFromAdjacencyList(input: Array<string>, graph_name: string): IGraph
```

Implementation of ICSVInput.readFromAdjacencyList
Defined in input/CSVInput.ts:94

Parameters

- **input:** *Array<string>*
- **graph_name:** *string*

Returns *IGraph*

readFromAdjacencyListFile

```
readFromAdjacencyListFile(filepath: string): IGraph
```

Implementation of ICSVInput.readFromAdjacencyListFile
Defined in input/CSVInput.ts:76

Parameters

- **filepath:** *string*

Returns *IGraph*

readFromAdjacencyListURL

```
readFromAdjacencyListURL(fileurl: string, cb: Function): void
```

Implementation of ICSVInput.readFromAdjacencyListURL
Defined in input/CSVInput.ts:35

Parameters

- **fileurl:** *string*
- **cb:** *Function*

Returns *void*

readFromEdgeList

```
readFromEdgeList(input: Array<string>, graph_name: string): IGraph
```

Implementation of ICSVInput.readFromEdgeList
Defined in input/CSVInput.ts:156

Parameters

- **input:** *Array<string>*
- **graph_name:** *string*

Returns *IGraph*

readFromEdgeListFile

```
readFromEdgeListFile(filepath: string): IGraph
```

Implementation of ICSVInput.readFromEdgeListFile
Defined in input/CSVInput.ts:81

Parameters

- **filepath:** *string*

Returns *IGraph*

readFromEdgeListURL

```
readFromEdgeListURL(fileurl: string, cb: Function): void
```

Implementation of ICSVInput.readFromEdgeListURL
Defined in input/CSVInput.ts:40

Parameters

- **fileurl:** *string*
- **cb:** *Function*

Returns *void*

readGraphFromURL

```
readGraphFromURL(fileurl: string, cb: Function, localFun:
Function): void
```

Defined in input/CSVInput.ts:45

Parameters

- **fileurl:** *string*
- **cb:** *Function*
- **localFun:** *Function*

Returns *void*

ICSVInput

```
ICSVInput:
```

Defined in input/CSVInput.ts:13

_direction_mode

```
_direction_mode: boolean
```

Defined in input/CSVInput.ts:16

_explicit_direction

```
_explicit_direction: boolean
```

Defined in input/CSVInput.ts:15

_separator

```
_separator: string
```

Defined in input/CSVInput.ts:14

readFromAdjacencyList

```
readFromAdjacencyList(input: Array<string>, graph_name:
string): IGraph
```

Defined in input/CSVInput.ts:19

Parameters

- **input:** *Array<string>*
- **graph_name:** *string*

Returns *IGraph*

readFromAdjacencyListFile

```
readFromAdjacencyListFile(filepath: string): IGraph
```

Defined in input/CSVInput.ts:18

Parameters

- **filepath:** *string*

Returns *IGraph*

readFromAdjacencyListURL

```
readFromAdjacencyListURL(fileurl: string, cb: Function): any
```

Defined in input/CSVInput.ts:20

Parameters

- **fileurl:** *string*
- **cb:** *Function*

Returns *any*

readFromEdgeList

```
readFromEdgeList(input: Array<string>, graph_name: string): IGraph
```

Defined in input/CSVInput.ts:23

Parameters

- **input:** *Array<string>*
- **graph_name:** *string*

Returns *IGraph*

readFromEdgeListFile

```
readFromEdgeListFile(filepath: string): IGraph
```

Defined in input/CSVInput.ts:22

Parameters

- **filepath:** *string*

Returns *IGraph*

readFromEdgeListURL

```
readFromEdgeListURL(fileurl: string, cb: Function): any
```

Defined in input/CSVInput.ts:24

Parameters

- **fileurl:** *string*
- **cb:** *Function*

Returns *any*

"input/JSONInput"

```
"input/JSONInput":
```

Defined in input/JSONInput.ts:1

JSONInput

```
JSONInput:
```

Defined in input/JSONInput.ts:42

constructor

```
new JSONInput(_explicit_direction?: boolean, _direction?: boolean, _weighted_mode?: boolean): JSONInput
```

Defined in input/JSONInput.ts:42

Parameters

- Default value **_explicit_direction:** *boolean* = true
- Default value **_direction:** *boolean* = false
- Default value **_weighted_mode:** *boolean* = false

Returns *JSONInput*

_direction

```
_direction: boolean
```

Implementation of IJSONInput._direction
Defined in input/JSONInput.ts:45

_explicit_direction

```
_explicit_direction: boolean
```

Implementation of IJSONInput._explicit_direction
Defined in input/JSONInput.ts:44

_weighted_mode

```
_weighted_mode: boolean
```

Implementation of IJSONInput._weighted_mode
Defined in input/JSONInput.ts:46

checkNodeEnvironment

```
checkNodeEnvironment(): void
```

Defined in input/JSONInput.ts:170

Returns *void*

readFromJSON

```
readFromJSON(json: JSONGraph): IGraph
```

Defined in input/JSONInput.ts:98

In this case, there is one great difference to the CSV edge list cases: If you don't explicitly define a directed edge, it will simply instantiate an undirected one we'll leave that for now, as we will produce apt JSON sources later anyways...

Parameters

- **json:** *JSONGraph*

Returns *IGraph*

readFromJSONFile

```
readFromJSONFile(filepath: string): IGraph
```

Implementation of IJSONInput.readFromJSONFile
Defined in input/JSONInput.ts:49

Parameters

- **filepath:** *string*

Returns *IGraph*

readFromJSONURL

```
readFromJSONURL(fileurl: string, cb: Function): void
```

Implementation of IJSONInput.readFromJSONURL
Defined in input/JSONInput.ts:56

Parameters

- **fileurl:** *string*
- **cb:** *Function*

Returns *void*

IJSONInput

```
IJSONInput:
```

Defined in input/JSONInput.ts:31

_direction

```
_direction: boolean
```

Defined in input/JSONInput.ts:33

_explicit_direction

```
_explicit_direction: boolean
```

Defined in input/JSONInput.ts:32

_weighted_mode

```
_weighted_mode: boolean
```

Defined in input/JSONInput.ts:34

readFromJSON

```
readFromJSON(json: object): IGraph
```

Defined in input/JSONInput.ts:37

Parameters

- **json:** *object*

Returns *IGraph*

readFromJSONFile

```
readFromJSONFile(file: string): IGraph
```

Defined in input/JSONInput.ts:36

Parameters

- **file:** *string*

Returns *IGraph*

readFromJSONURL

```
readFromJSONURL(fileurl: string, cb: Function): void
```

Defined in input/JSONInput.ts:38

Parameters

- **fileurl:** *string*
- **cb:** *Function*

Returns *void*

JSONEdge

```
JSONEdge:
```

Defined in input/JSONInput.ts:11

directed

```
directed: string
```

Defined in input/JSONInput.ts:13

to

```
to: string
```

Defined in input/JSONInput.ts:12

type

```
type: string
```

Defined in input/JSONInput.ts:15

weight

```
weight: string
```

Defined in input/JSONInput.ts:14

JSONGraph

```
JSONGraph:
```

Defined in input/JSONInput.ts:24

data

```
data: object
```

Defined in input/JSONInput.ts:28

Type declaration

- [**key:** *string*]: *JSONNode*

edges

```
edges: number
```

Defined in input/JSONInput.ts:27

name

```
name: string
```

Defined in input/JSONInput.ts:25

nodes

```
nodes: number
```

Defined in input/JSONInput.ts:26

JSONNode

```
JSONNode:
```

Defined in input/JSONInput.ts:18

coords

```
coords: object
```

Defined in input/JSONInput.ts:20

Type declaration

- [**key:** *string*]: *Number*

edges

edges: *Array<JSONEdge>*

Defined in input/JSONInput.ts:19

features

features: *object*

Defined in input/JSONInput.ts:21

Type declaration

- [**key:** *string*]: *any*

DEFAULT_WEIGHT

DEFAULT_WEIGHT: *number*

Defined in input/JSONInput.ts:9

"search/BFS"

"search/BFS":

Defined in search/BFS.ts:1

BFS_Callbacks

BFS_Callbacks:

Defined in search/BFS.ts:23

init_bfs

init_bfs: *Array<Function>*

Defined in search/BFS.ts:24

node_marked

node_marked: *Array<Function>*

Defined in search/BFS.ts:26

node_unmarked

node_unmarked: *Array<Function>*

Defined in search/BFS.ts:25

sort_nodes

sort_nodes: *Function*

Defined in search/BFS.ts:27

BFS_Config

BFS_Config:

Defined in search/BFS.ts:9

callbacks

callbacks: *BFS_Callbacks*

Defined in search/BFS.ts:11

dir_mode

dir_mode: *GraphMode*

Defined in search/BFS.ts:12

filters

```
filters: any
```

Defined in search/BFS.ts:14

messages

```
messages: object
```

Defined in search/BFS.ts:13

Type declaration

result

```
result: object
```

Defined in search/BFS.ts:10

Type declaration

- [**id:** *string*]: *BFS_ResultEntry*

BFS_ResultEntry

```
BFS_ResultEntry:
```

Defined in search/BFS.ts:17

counter

```
counter: number
```

Defined in search/BFS.ts:20

distance

```
distance: number
```

Defined in search/BFS.ts:18

parent

```
parent: IBaseNode
```

Defined in search/BFS.ts:19

BFS_Scope

```
BFS_Scope:
```

Defined in search/BFS.ts:30

adj_nodes

```
adj_nodes: Array<NeighborEntry>
```

Defined in search/BFS.ts:38

current

```
current: IBaseNode
```

Defined in search/BFS.ts:34

marked

```
marked: object
```

Defined in search/BFS.ts:31

Type declaration

- [**id:** *string*]: *boolean*

next_edge

```
next_edge: IBaseEdge
```

Defined in search/BFS.ts:36

next_node

```
next_node: IBaseNode
```

Defined in search/BFS.ts:35

nodes

```
nodes: object
```

Defined in search/BFS.ts:32

Type declaration

- [**id:** *string*]: *IBaseNode*

queue

```
queue: Array<IBaseNode>
```

Defined in search/BFS.ts:33

root_node

```
root_node: IBaseNode
```

Defined in search/BFS.ts:37

BFS

```
BFS(graph: IGraph, v: IBaseNode, config?: BFS_Config): object
```

Defined in search/BFS.ts:55

Breadth first search - usually performed to see reachability etc. Therefore we do not want 'segments' or 'components' of our graph, but simply one well defined result segment covering the whole graph.

constructor

Parameters

- **graph:** *IGraph*

  the graph to perform BFS on

- **v:** *IBaseNode*

  the vertex to use as a start vertex

- Optional **config:** *BFS_Config*

  an optional config object, will be automatically instantiated if not passed by caller

Returns *object*

- [**id:** *string*]: *BFS_ResultEntry*

prepareBFSStandardConfig

```
prepareBFSStandardConfig(): BFS_Config
```

Defined in search/BFS.ts:151

Returns *BFS_Config*

"search/DFS"

```
"search/DFS":
```

Defined in search/DFS.ts:1

DFSVisit_Scope

```
DFSVisit_Scope:
```

Defined in search/DFS.ts:33

adj_nodes

```
adj_nodes: Array<NeighborEntry>
```

Defined in search/DFS.ts:35

current

```
current: IBaseNode
```

Defined in search/DFS.ts:37

current_root

```
current_root: IBaseNode
```

Defined in search/DFS.ts:38

stack

```
stack: Array<StackEntry>
```

Defined in search/DFS.ts:34

stack_entry

```
stack_entry: StackEntry
```

Defined in search/DFS.ts:36

DFS_Callbacks

```
DFS_Callbacks:
```

Defined in search/DFS.ts:17

adj_nodes_pushed

```
adj_nodes_pushed: Array<Function>
```

Defined in search/DFS.ts:23

init_dfs

```
init_dfs: Array<Function>
```

Defined in search/DFS.ts:18

init_dfs_visit

```
init_dfs_visit: Array<Function>
```

Defined in search/DFS.ts:19

node_marked

```
node_marked: Array<Function>
```

Defined in search/DFS.ts:21

node_popped

```
node_popped: Array<Function>
```

Defined in search/DFS.ts:20

node_unmarked

```
node_unmarked: Array<Function>
```

Defined in search/DFS.ts:22

sort_nodes

```
sort_nodes: Function
```

Defined in search/DFS.ts:24

DFS_Config

```
DFS_Config:
```

Defined in search/DFS.ts:8

callbacks

```
callbacks: DFS_Callbacks
```

Defined in search/DFS.ts:10

dfs_visit_marked

```
dfs_visit_marked: object
```

Defined in search/DFS.ts:12

Type declaration

- [**id:** *string*]: *boolean*

dir_mode

```
dir_mode: GraphMode
```

Defined in search/DFS.ts:11

filters

```
filters: any
```

Defined in search/DFS.ts:14

messages

```
messages: object
```

Defined in search/DFS.ts:13

Type declaration

visit_result

```
visit_result: object
```

Defined in search/DFS.ts:9

Type declaration

DFS_Scope

```
DFS_Scope:
```

Defined in search/DFS.ts:41

marked

```
marked: object
```

Defined in search/DFS.ts:42

Type declaration

- [**id:** *string*]: *boolean*

nodes

```
nodes: object
```

Defined in search/DFS.ts:43

Type declaration

- [**id:** *string*]: *IBaseNode*

StackEntry

```
StackEntry:
```

Defined in search/DFS.ts:27

node

```
node: IBaseNode
```

Defined in search/DFS.ts:28

parent

```
parent: IBaseNode
```

Defined in search/DFS.ts:29

weight

```
weight: number
```

Defined in search/DFS.ts:30

DFS

```
DFS(graph: IGraph, root: IBaseNode, config?: DFS_Config):
Array<object>
```

Defined in search/DFS.ts:198

Depth first search - used for reachability / exploration of graph structure and as a basis for topological sorting and component / community analysis. Because DFS can be used as a basis for many other algorithms, we want to keep the result as generic as possible to be populated by the caller rather than the core DFS algorithm.

constructor

Parameters

- **graph:** *IGraph*
- **root:** *IBaseNode*
- Optional **config:** *DFS_Config*

Returns *Array<object>*

[]}

DFSVisit

```
DFSVisit(graph: IGraph, current_root: IBaseNode, config?: DFS_Config): object
```

Defined in search/DFS.ts:57

DFS Visit - one run to see what nodes are reachable from a given "current" root node

constructor

Parameters

- **graph:** *IGraph*
- **current_root:** *IBaseNode*
- Optional **config:** *DFS_Config*

Returns *object*

prepareDFSStandardConfig

```
prepareDFSStandardConfig(): DFS_Config
```

Defined in search/DFS.ts:328

First instantiates config file for DFSVisit, then enhances it with outer DFS init callback

Returns *DFS_Config*

prepareDFSVisitStandardConfig

```
prepareDFSVisitStandardConfig(): DFS_Config
```

Defined in search/DFS.ts:284

This is the only place in which a config object is instantiated

(except manually, of course)

Therefore, we do not take any arguments

Returns *DFS_Config*

"search/PFS"

```
"search/PFS":
```

Defined in search/PFS.ts:1

PFS_Callbacks

```
PFS_Callbacks:
```

Defined in search/PFS.ts:25

better_path

```
better_path: Array<Function>
```

Defined in search/PFS.ts:29

goal_reached

```
goal_reached: Array<Function>
```

Defined in search/PFS.ts:30

init_pfs

```
init_pfs: Array<Function>
```

Defined in search/PFS.ts:26

node_closed

```
node_closed: Array<Function>
```

Defined in search/PFS.ts:28

node_open

```
node_open: Array<Function>
```

Defined in search/PFS.ts:27

PFS_Config

```
PFS_Config:
```

Defined in search/PFS.ts:10

callbacks

```
callbacks: PFS_Callbacks
```

Defined in search/PFS.ts:12

dir_mode

```
dir_mode: GraphMode
```

Defined in search/PFS.ts:13

filters

```
filters: any
```

Defined in search/PFS.ts:16

goal_node

```
goal_node: IBaseNode
```

Defined in search/PFS.ts:14

messages

```
messages: PFS_Messages
```

Defined in search/PFS.ts:15

result

```
result: object
```

Defined in search/PFS.ts:11

Type declaration

- [**id:** *string*]: *PFS_ResultEntry*

PFS_Messages

```
PFS_Messages:
```

Defined in search/PFS.ts:33

better_path_msgs

```
better_path_msgs: Array<string>
```

Defined in search/PFS.ts:37

goal_reached_msgs

goal_reached_msgs: *Array<string>*

Defined in search/PFS.ts:38

init_pfs_msgs

init_pfs_msgs: *Array<string>*

Defined in search/PFS.ts:34

node_closed_msgs

node_closed_msgs: *Array<string>*

Defined in search/PFS.ts:36

node_open_msgs

node_open_msgs: *Array<string>*

Defined in search/PFS.ts:35

PFS_ResultEntry

PFS_ResultEntry:

Defined in search/PFS.ts:19

counter

counter: *number*

Defined in search/PFS.ts:22

distance

distance: *number*

Defined in search/PFS.ts:20

parent

parent: *IBaseNode*

Defined in search/PFS.ts:21

PFS_Scope

PFS_Scope:

Defined in search/PFS.ts:41

CLOSED

CLOSED: *object*

Defined in search/PFS.ts:45

Type declaration

- [**id:** *string*]: *boolean*

OPEN

OPEN: *object*

Defined in search/PFS.ts:44

Type declaration

- [**id:** *string*]: *boolean*

OPEN_HEAP

```
OPEN_HEAP: BinaryHeap
```

Defined in search/PFS.ts:43

adj_nodes

```
adj_nodes: Array<NeighborEntry>
```

Defined in search/PFS.ts:52

current

```
current: NeighborEntry
```

Defined in search/PFS.ts:48

next_edge

```
next_edge: IBaseEdge
```

Defined in search/PFS.ts:50

next_node

```
next_node: IBaseNode
```

Defined in search/PFS.ts:49

nodes

```
nodes: object
```

Defined in search/PFS.ts:47

Type declaration

- [**id:** *string*]: *IBaseNode*

root_node

```
root_node: IBaseNode
```

Defined in search/PFS.ts:51

PFS

```
PFS(graph: IGraph, v: IBaseNode, config: PFS_Config): object
```

Defined in search/PFS.ts:67

Priority first search

Like BFS, we are not necessarily visiting the whole graph, but only what's reachable from a given start node.

config — a config object similar to that used in BFS, automatically instantiated if not given..

Parameters

- **graph:** *IGraph*
  the graph to perform PFS only
- **v:** *IBaseNode*
  the node from which to start PFS
- **config:** *PFS_Config*

Returns *object*

- [**id:** *string*]: *PFS_ResultEntry*

preparePFSStandardConfig

```
preparePFSStandardConfig(): PFS_Config
```

Defined in search/PFS.ts:175

Returns *PFS_Config*

"utils/callbackUtils"

```
"utils/callbackUtils":
```

Defined in utils/callbackUtils.ts:1

execCallbacks

```
execCallbacks(cbs: Array<Function>, context?: any): void
```

Defined in utils/callbackUtils.ts:4

Parameters

- **cbs:** *Array<Function>*
- Optional **context:** *any*
  this pointer to the DFS or DFSVisit function

Returns *void*

"utils/remoteUtils"

```
"utils/remoteUtils":
```

Defined in utils/remoteUtils.ts:1

retrieveRemoteFile

```
retrieveRemoteFile(url: string, cb: Function): ClientRequest
```

Defined in utils/remoteUtils.ts:10

todo : Test it !!!

Parameters

- **url:** *string*
- **cb:** *Function*

Returns *ClientRequest*

"utils/structUtils"

```
"utils/structUtils":
```

Defined in utils/structUtils.ts:1

clone

```
clone(obj: any): any
```

Defined in utils/structUtils.ts:8

Method to deep clone an object

Parameters

- **obj:** *any*

Returns *any*

findKey

```
findKey(obj: Object, cb: Function): string
```

Defined in utils/structUtils.ts:84

todo Test !!!

Parameters

- **obj:** *Object*
- **cb:** *Function*

Returns *string*

mergeArrays

```
mergeArrays(args: Array<Array<any>>, cb?: Function): Array<any>
```

Defined in utils/structUtils.ts:30

args an Array of any kind of objects

cb callback to return a unique identifier; if this is duplicate, the object will not be stored in result.

Parameters

- **args:** *Array<Array<any>>*
- Default value **cb:** *Function* = undefined

Returns *Array<any>*

mergeObjects

```
mergeObjects(args: Array<Object>): object
```

Defined in utils/structUtils.ts:60

Overwrites obj1's values with obj2's and adds obj2's if non existent in obj1

Parameters

- **args:** *Array<Object>*

  Array of all the object to take keys from

Returns *object*

result object

Legend

Module	Enumeration
Object literal	Enumeration member
Variable	Property
Function	Method
Function with type parameter	
Index signature	
Type alias	

Interface	Class
Interface with type parameter	Class with type parameter
Constructor	Constructor
Property	Property
Method	Method
Index signature	Accessor
	Index signature

Inherited constructor	Protected property
Inherited property	Protected method
Inherited method	Protected accessor
Inherited accessor	

Private property	Static property
Private method	Static method
Private accessor	