FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Bitstream

Eine bottom-up/top-down Methode für interaktive Bitcoin-Visualisierungen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Matthias Gusenbauer, Bsc.
Matrikelnummer 01125577

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Univ.Ass. Dr.techn. Manuela Waldner, MSc

Wien, 14. Mai 2018

Matthias Gusenbauer   Eduard Gröller

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Bitstream

A Bottom-Up/Top-Down Approach to Data Loading for Interactive Bitcoin Visualizations

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Matthias Gusenbauer, Bsc.
Registration Number 01125577

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Univ.Ass. Dr.techn. Manuela Waldner, MSc

Vienna, 14th May, 2018

Matthias Gusenbauer

Eduard Gröller

Erklärung zur Verfassung der Arbeit

Matthias Gusenbauer, Bsc.
Gablenzgasse 64, 1160 Wien, Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. Mai 2018

Matthias Gusenbauer

Acknowledgements

I thank the past for what I have become and the future for what I will become. I thank society for living on the mirror's edge - all the problems and puzzles that have yet to be solved. I thank whatever made me love my work and for all the hardship that gives me the opportunity to grow personally. Foremost, I want to thank my partner Tajima Moe who, despite the geographical distance, has always supported me and been very understanding when I vented my frustration about exactly the things that do motivate me. I want to thank my family and my friends for being a part of the journey, for accepting me, giving me inspiration, and things to reflect upon. Lastly, I want to thank the virtually endless information highway called world wide web that is curse and boon simultaneously. Without these people and the limitless information to learn, I would not have been what I am today. thx and gg!

Kurzfassung

Die Analyse großer Datenmengen is ein immer größer werdendes Problem. Bitcoin hat mehr Daten erzeugt, als es möglich ist zu analysieren. Um diese Schwierigkeiten zu kompensieren, wurden verschiedenste Ideen, zum Beispiel Datenaggregation oder Datenminimierung, vorgeschlagen. Wiederum andere Arbeiten konzentrieren sich auch auf die Einführung neuer Visualisierungstypen, die auf die neuartige Visualisierung von Daten ausgerichtet sind. Die Visualisierung von Graphen durch Node-Link-Diagramme bleibt jedoch eine schwierige Herausforderung. Die Analyse des Bitcoin-Transaktionsgraphen ist aufgrund des Bitcoin-Protokolls und der Datenmenge ein schwieriges Problem. Diese Arbeit kombiniert zwei Datenverarbeitungsstrategien, um große Netzwerkdaten auf Standardhardware zu visualisieren. Durch Visualisierung werden Muster identifiziert, mit denen man Transaktionen deanonymisieren kann. Ein Proxy-Server vorverarbeitet Daten bevor sie auf einem Web-Client visualisiert werden. Der Proxy nutzt parallele Datenverarbeitung, um schnell genug für die interaktive Visualisierung zu sein. Dies geschieht durch inkrementelles Laden (Bottom-Up), was es ermöglicht, Daten sofort ohne (Vor-)Verarbeitung zu visualisieren. Die Blockchain als zentraler Datenspeicher von Bitcoin ist über 163 Gigabyte groß. Der daraus resultierende Graph hat mehr als 800 Millionen Knoten. Da diese Informationen zu groß sind, um sie zu visualisieren, verwenden wir auch einen Top-Down-Ansatz der Datenaggregation und Graphminimierung des Transaktionsgraphen. Mit dieser Methodik werden Probleme langer Verzögerungen gelöst. Das System wird durch den Dialog mit Sicherheitsexperten, im Bereich Crypto-Währungen, konzipiert und implementiert. Die explorative Analyse eines großen Datensatzes, wie etwa Bitcoin, wird durch die in dieser Arbeit vorgestellte Methodik ermöglicht. Weiters hilft es Sicherheitsexperten, den Geldfluss in einem Finanznetzwerk zu analysieren, das von Kriminellen wegen seiner Anonymität genutzt wird. Wir bewerten das Ergebnis anhand der Verarbeitungsleistung und der Rückmeldung dieser Sicherheitsexperten und vergleichen das Leistungsverhalten mit aktuellen, bewährten Vorgangsweisen.

Abstract

Analyzing large amounts of data is becoming an ever increasing problem. Bitcoin as an example has produced more data than is possible to analyze. In order to compensate for these difficulties, creative ideas that employ data aggregation or minimization have been proposed. Other work also focuses on introducing novel visualization types that are geared towards the visualization of blockchain data. However, visualization of graphs through node-link diagrams remains a difficult challenge. Analysis of the Bitcoin transaction graph to follow bitcoin (BTC) transactions (TXs) poses a difficult problem due to the Bitcoin protocol and the amount of data. This thesis combines two data processing strategies to visualize big network data on commodity hardware. The idea is to use visualization as a technique to analyze a data-set containing Bitcoin transaction information. Criminals use Bitcoin as a means of payment because of its guaranteed pseudonymity. Through visualization we aim to identify patterns that will allow us to deanonymize transactions. To do so we use a proxy server that does data preprocessing before they are visualized on a web client. The proxy leverages parallel computing to be able to do top-down and bottom-up data processing fast enough for interactive visualization. This is done through incremental loading (bottom-up), which enables to visualize data immediately without a (pre-)processing delay. The database containing the public Bitcoin ledger is over 163 gigabytes in size. The resulting graph has more than 800 million nodes. As this information is too much to be visualized, we also employ a top-down approach of data aggregation and graph minimization of the transactional graph. Through this methodology we intend to solve performance problems of long processing delays and the problem of fractured data where the data is shown only partially in the visualization. We collaborate with security experts who share insights into their expertise through a continuously ongoing dialog. Exploratory analysis on a big data-set such as the Bitcoin ledger, enabled through the methodology presented in this thesis, will help security experts to analyze the money flow in a financial network that is used by criminals for its anonymity. We evaluate the result through the performance and feedback of these security experts as well as benchmark the performance against current best practice approaches.

Contents

CHAPTER 1

Introduction

Information is ubiquitious; digital systems are part of all aspects of the lives of people. These systems are the arteries of a globally spanning information network. Many different types of data flow through this infrastructure and every part of this network produces and stores data. People have claimed that data is the new oil [1], and that the human species has progressed from the oil and atomic age into the information age [2].

1.1 Problem Statement

In an era where dat a is ubiquitous, it is important to be able to retrieve information encoded in the data. However, the amount of data collected is disproportionate to the capabilities to analyze it. This leads to many valuable datasets, considered "big data", that are difficult to analyze. For example, all YouTube users combined upload approximately 48 hours of video every *minute* to the website, or users on Facebook upload 100 terabytes of data *every day* [3]. While the processing capabilities of hardware increases by doubling approximately every 18 months (known as Moore's law), data production increases at an even faster pace. Kryder's law states that storage capacities double approximately every 13 months [4]. This phenomenon can be seen in Figure 1.1. It is clear that this imbalance of growth makes it impossible to analyze all the produced, stored and available data [5]. Therefore, it is necessary to not rely on the raw improvement of processing power but to find efficient solutions and speed improvements for the algorithmic side of the data processing pipeline.

There are two different approaches to transform collected data into information. One is through using statistical analysis. The other is leveraging the powerful capabilities of human sense-making through representing data visually, known as visualization. While the general goal of both methodologies is the same, the path from raw data to valuable information is vastly different. The focus of this thesis is on contributing to the body of knowledge in visual analytics of cryptocurrencies. The combination of analytical reasoning

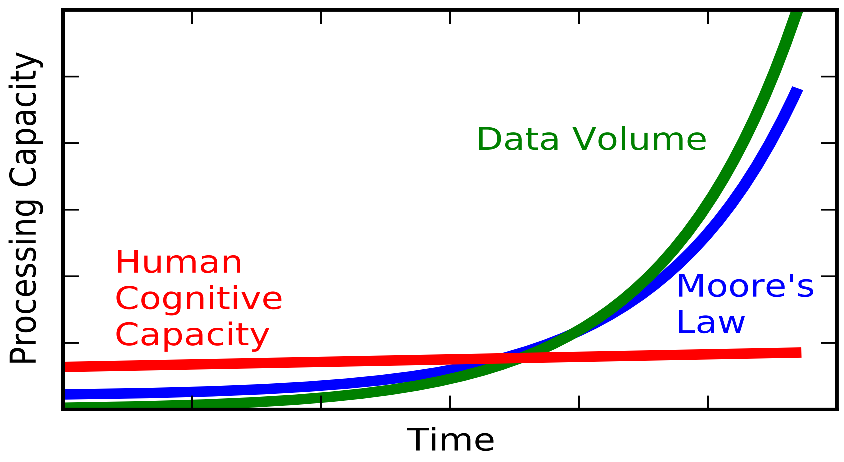


Figure 1.1: The development over time for processing, data storage and human capabilities is inbalanced. It is therefore necessary to use a mix of human and technical capabilities to counter the faster growth of data. [5]

and interactive visual interfaces, known as visual analytics, is a feasible approach to big data analysis. However, there is not one standard methodology for big data visualization as of early 2018. There has been a lot of work to explore possible solutions to the data abundance. Each tackles its own distinct subset of the total problem space. However, the general idea for all solutions is to reduce the number of data items to compensate for the performance limitations of the visualization pipeline. In each of the four processing steps shown in Figure 1.2 can be bottlenecks. Depending on the amount of raw data, loading them from the data storage can be too slow. When the data is loaded, data analysis itself can take too much time for interactive visualization. Filtering, again, can be slow if the data size has not decreased sufficiently in the analysis step. Mapping and subsequent rendering can also cause performance problems if the objects to draw are too numerous.

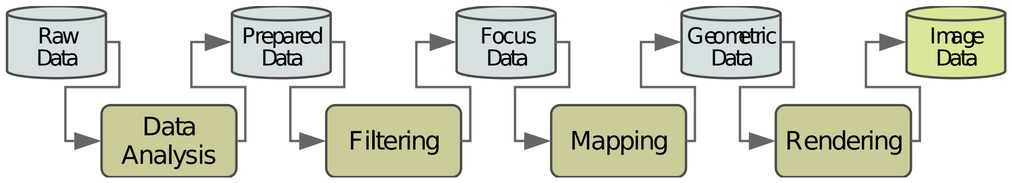


Figure 1.2: Schematic view of the visualization pipeline [6, 7], showing types of data at the top and processing steps at the bottom. Every single one of the processing steps can be a performance bottleneck if the dataset size exceeds computing capabilities.

To give the methodology proposed in this thesis real world relevance, we will use a cryptocurrency dataset for analysis. The motivation to use such a dataset comes from the fact that cryptocurrencies are considered a pseudonymous form of currency. Due to the pseudonymous nature of the currency it has been used by criminals for payment in malware campaigns, darknet market places and money laundering. Additionally, the underlying network produces large amounts of data, of currently more than 800 million nodes in a graph that fall into the previously described category of big data. The size of the dataset and the unique properties of the system make it difficult for law enforcement to follow the stream of money in the network.

It is possible to distinguish between three general groups of Bitcoin visualization techniques: visualization of Bitcoin statistics as bar charts, line diagrams and other similar methods, node-link diagrams to show the network topology and, lastly, novel visualization techniques that try to convey information through visualizations and are tailored to Bitcoin or sometimes are even artistic. Visualization of statistical data [8, 9, 10] of Bitcoin is limited in conveying topological information of the payment network. By using node-link diagrams one can overcome this limitation [11]. However, this type of visualization does not scale well enough with the number of nodes and links visible [12]. Processing power therefore limits the size of the analyzed dataset. This can be overcome by specialized hardware, which, in turn, reduces the accessibility to analysis techniques [13]. Lastly, it is possible to visualize the data in a completely new way [14]. By doing so, a user has an increased initial cognitive load to learn how to interpret the visualization. Furthermore, these visualizations can still be very slow and therefore hinder exploratory analysis [15]. It is our goal to mitigate the shortcomings of the node-link visualization approach. Being able to navigate the data space in an intuitive way through the chosen node-link representation, can help users to understand Bitcoin.

1.2 Contributions

The goals of this thesis are twofold. On the one hand it is about the general contribution to the science of big data visualization, while on the other hand, it contributes a novel method to visualize tens of thousands of transactions and addresses of the Bitcoin network. The three objectives for big data visualization are:

- Provide an interactive visualization of a subset of considerable size of the Bitcoin network.
- Hide the processing latency from the user.
- Reduce the data that reaches the visualization client in a meaningful way and hide the big data aspect from the client and the user.

The second set of goals is related to the application of the tool to analyze the cryptocurrency dataset by a user. After consultation with experts we identified two difficult to achieve tasks. One is to analyze the transaction patterns that an address is involved with. This is interesting for identifying money laundering operations and anonymity services. A second problem is to analyze whether two known parties are transacting with each other. Extracted from these questions we formulated the following two research questions:

Q1: Find interesting transaction patterns and structures in the Bitcoin network.

Q2: Be able to find the money flow from select points of interests to other points of interest.

1.3 Structure of this Thesis

The following chapters of this thesis are structured as follows:

Chapter 2 gives a summary of the history of visual analytics and outlines differences to information visualization and scientific visualization. Furthermore, the surveyed related work will be presented to give an overview of the status quo of big data visualization. The last part of this chapter will give a primer on the history and inner workings of Bitcoin as the oldest and most popular example of a cryptocurrency.

Chapter 3 will give an overview of the conceptual architecture of the bottom-up methodology proposed in this thesis. It will also be explained why certain design decisions have been made.

Chapter 4 presents the top-down data processing methodology for graph reduction and goes into the details of the database structure.

Chapter 5 highlights the actual technology that has been used. An overview of the technology stack and its algorithms is given. Complex parts of the implementation are highlighted and explained.

Chapter 6 presents the benchmarking results of various operations and user tasks performed with the system.

Chapter 7 discusses the previously presented results and their implications. Furthermore future work for improvements and additional features are discussed in detail.

Chapter 8 summarizes the thesis.

CHAPTER 2

Background

This chapter gives a brief introduction to Bitcoin and information about its relevancy. Additionally, it surveys the general state of big data visualization, especially for large graphs and, finally, concludes with the state of Bitcoin visualizations.

2.1 Bitcoin

In autumn of 2009 a paper with the title "Bitcoin: A Peer-to-Peer Electronic Cash System", containing the techical specification of a novel peer-to-peer (p2p) payment system, was published, on an online cryptography mailing list, by Satoshi Nakamoto [16]. It was his[1] answer to the previous financial crisis caused by the banking system.

2.1.1 History

After the initial publication of the Bitcoin whitepaper, Nakamoto published the first open source client needed to participate in the Bitcoin network following his specifications. At the same time, the first entry in the public ledger of the Bitcoin network, known as the genesis block, was created. This coincided with the creation of the first bitcoins (BTCs), that then could be traded within the network [3]. After the initial inception of the system, early adopters started to participate in the network by running their own servers. In 2018, almost ten years later, the network has, through several growth periods, reached a scale where the traditional financial system, state actors and other established institutions, have to pay attention to [17]. As of January, 26th of 2018 there are 16,826,975 BTCs, with a price of roughly 11,000 US dollars (USD) each, in circulation. This gives a total market capitalization of 190 billion USD.

[1]It is unknown whether Satoshi Nakamoto is a person or a group. The gender is also unknown but for simplicity and in conformity with Japanese names the pronoun "he" is used.

While the financial data of BTC is impressive for a system without any governmental backing, the technical details, such as how problems of distributed systems have been resolved in BTC, are of interest for this thesis. In order to fully understand the problems that we try to solve, a basic understanding of the BTC protocol is necessary. BTC was envisioned as an anonymous p2p payment system and as such, it has attracted not only legitimate users but also bad actors that use its technological properties to evade law enforcement, while engaging in questionable businesses like drug trade, cybercrime, extortion etc.

2.1.2 Technical Details

In order to understand why Bitcoin can act as an anonymous payment channel, it is important to understand some details of the protocol. The general approach of Bitcoin is a network that reaches consensus by itself without any central governing body. The network is comprised of nodes that are connected in a p2p fashion. All participating nodes in the network share a copy of a data structure called the *blockchain* that keeps a record of all transactions in the network. Satoshi Nakamoto solved finding a consensus by making it difficult to cheat through the usage of a "proof of work", which is an innovative use of cryptography.

Transactions in the network are collected in blocks that have to be mined by volunteers on the network. This mining is the "proof of work", where a miner has to find a partial pre-image hash collision in order for a block to be accepted as valid by the network. In detail this means that a miner has to find a message x for a given hash h. To make it possible within a feasible amount of time only part of the hash is fixed. The incentive for the miner to participate in the energy hungry and therefore costly endeavor is that if she can find a valid block she is rewarded with a certain number of BTCs. Once a valid block is found, the miner publishes it to the network and it is accepted, distributed, and added to the blockchain. The transactions that a miner has chosen to accept in the block she mined are therefore now part of the transactional history of Bitcoin. This process acts as consensus finding in the network with the benefit of allowing transactions to be sent. Figure 2.1 shows a small part of the blockchain consisting of three blocks. The figure shows a small part of the blockchain showing three blocks with height 512256, 512257 and 512258. The block data contains information about the included transactions and the previous block hash to uniquely identify a preceding block. This linking back to a previous block is what gives the blockchain its name. The nonce is a field that can be arbitrarily chosen by miners to change the input of the hash function in order to generate varying hashes. Mining is the process of trying different values for the nonce in order to find a hash that matches the requirements given by the system. Details on the complete structure can be found in the Bitcoin developer reference [18].

In order to transfer money from one party to another, one has to publish a transaction to the network. While, behind the scenes, this is also a very technically complex step, to understand the problem of following this value transfer through the network, it is enough to know the process on a conceptual level. If A wants to send money to B, she collects the amount of BTC she wants to send to B in a list of accounts, known as *inputs*, and specifies the accounts of B, and possibly others, as output of a transaction (TX). One requirement for this to be valid is that the sum of the total input has to be greater than or equal to the sum of the outputs. There are some peculiarities on how money can be transferred and how a miner is paid. However, the important part to note here is that the list of inputs is an unordered list. The same holds true for the list of outputs in the transaction. Figure 2.2 shows the structure of two transactions where Alice sends BTC to Bob who then sends money to a marketplace. The unordered structure of the addresses within a transaction means that the input and output addresses are not mapped 1:1. It is therefore impossible to directly know who receives which coins. The only aspect one can be certain of is that the creator of the transaction is under control of all the inputs. The figure shows two transactions, one sent by Alice and another one by Bob. The green colored overlay shows, which addresses are controlled by Alice, blue are Bob's addresses respectively. Alice uses ten BTCs controlled by two of her addresses to send 8 BTCs to Bob. Bob then sends a transaction with a third address that holds two additional BTCs to pay nine BTCs to a marketplace. Once such a transaction is included in the blockchain it is considered valid and immutable. This means that the value transfer from one party to another one has taken place. Although the addresses in this figure are highlighted by the party controlling it, this is not known to the system. For output addresses, it is not known, which belong to different stakeholders. The only known fact is, that all the cryptographic keys of the input addresses are controlled by one entity.

Although all the information in the blockchain is public, this property gives in theory very strong privacy properties. In reality however, there are many steps that can go wrong that compromise anonymity. The seeming anonymity of addresses that are nothing more than a hash of a cryptographic key, give a false sense of anonymity. Address reuse, connecting public addresses with persons, and other usage patterns, give enough metadata to trace the flow of money through the Bitcoin network [19]. Although modern end-user software to interact with Bitcoin is implementing more and more privacy enhancing features, at the end of 2017 address reuse was still at 42% [20]. To mitigate the creation of a meta-data trail not only the end-user software developed automatic creation of new addresses for each TX, but also dedicated services came into existence. One such type of service is called *mixer*. People who want to gain increased anonymity can send their BTC to this service that mixes the BTC together with other peoples' BTCs and, after they have taken a commission, finally return them to the original sender. Although these mechanisms for improved anonymity exist, users are usually bad at following proper security hygiene [21, 22]. In combination with the inherent nature of publicly available financial transaction data in the blockchain, this leads to a complete publication of ones' finances if on is not careful enough [19].

In the case of cybercrime, fraud and money laundering culprits usually are more careful when interacting with technology. However, even if the person who owns a certain address is unknown, it is still possible to follow the money through the network. At some point, in order to use the money, the BTCs have to be exchanged for fiat currency. As long as

Bitcoin is not widespread enough to be used as a general form of payment, this problem remains.

2.1.3 Bitcoin Network in Numbers

With an approximate total market capitalization of 200 billion USD and a daily trading volume of about 8 billion USD, it is clear that there is a lot of movement in the cryptocurrency space even with Bitcoin alone [23]. In the last 24 hours, as of January 29th 2018, there were a total number of 155 blocks mined. The time between blocks was at an average of 8.57 minutes and in total there were 1937.5 bitcoins mined. In these blocks, there were 216,358 TX included. While the total number of addresses used in these transactions is unclear, it is certain that it is a multiple of the number of TXs. In reality, there are many more addresses included in each TX. The already large numbers of the last 24 hours are exceeded by the total number of transactions and addresses in the network [24]. Up to the end of January 2018, 506,604 blocks have been mined, 295,802,277 transactions have been performed and the website, used as source for these numbers, alone counts 22,619,304 users, where each of them will have multiple addresses [25].

2.2 Related Work of Real-Time Visualization of Big Data

The visual system of humans has evolved to process data incredibly fast and with a high bandwidth [26]. It would be unwise to leave this potential unused in information processing. However, in order to fully support humans to gain insight from data, it is also important to allow for exploratory data analysis through interaction. The solution to this is a research field called visual analytics that relies on methodologies from visualization and many other fields, and enhances them with automatic data processing and analysis.

Visual analytics is particularly useful for unguided data analysis. Unfortunately, the visualization part, as well as the automatic machine analysis aspect, of visual analytics can be a bottleneck due to performance limitations in combination with big data. Therefore, visual analytics of big data is still subject to intensive research. This becomes even more problematic if interactivity is necessary for exploratory approaches. Researchers circumvent this by either aggregating data in a meaningful way [27, 28, 29], that still allows for analysis of the bigger picture, in a top-down approach, or using a bottom-up approach where only parts of the data are loaded, processed, and visualized progressively [30, 31].

2.2.1 General Big Data Visual Analytics

For analyzing large amounts of data, the requirements for storage, processing, and presentation are different than for smaller more managable datasets. Big data makes it necessary to use specialized hardware and software. Storage systems supporting petabytes and more can not fit inside commodity hardware. Preprocessing this amount of data requires another type of hardware with processing power in mind. This leads to a distributed system where not necessarily one single computer is part of the three layers of data storage, processing, and visualization. For each of the three layers there is an active academic [32, 33, 34] and commercial community that engages in proposing solutions to make the data managable [35]. To make it possible to visually analyze big data, one can focus on problems of either category [36].

Due to limitations of the human perception and limited screenspace, it is not feasible to visualize raw large-scale datasets, even if it was computationally possible. Visualizing large datasets without any reduction during the processing step leads to overplotting. Showing too many elements on screen overwhelms the user and reduces the efficacy of the visualization. While there are multiple aspects where one can try to improve performance of data processing [33], there are two different approaches to load and transform data within the visualization pipeline, as shown in Figure 1.2:

- Top-down approaches start with an aggregation of the underlying data to give a general image of their structure while also avoiding overplotting. This requires an aggregation preprocessing step that generates these top-down views [27, 28, 29, 37, 38]. This approach also follows "*overview first, zoom and filter, details on demand*" [39], a central principle in visual analytics proposed by Shneiderman. For example, imMens by Liu et al. [27] uses data cubes, aggregations across multiple data dimensions, to show overview visualizations of the data. However, leveraging data cubes requires a preprocessing step to generate aggregational views of the data, or database, beforehand. SplatterJS [37] uses kernel density estimation as a form of aggregation to overcome overplotting. While it helps to reduce rendered elements on-screen, it may hide interesting data points. We use heuristics that fit our dataset to overcome this problem of losing detailed information. Nonetheless, the advantage of top-down approaches is that one can see the global structures within the data and then focus on possibly interesting details. Having a holistic view on the data, however, comes at the price of a preprocessing step that, depending on the size of the data, can be very costly.
- Bottom-up approaches start with a subset of the data and show only a small portion of the complete set. Sampling [30, 31] and specific user queries can be strategies for a starting point of loading data. However, this requires an analyst to navigate within the data to fully grasp global structures of them. An advantage is that the true representation of the data is loaded from the start without any preprocessing delay. Unfortunately, these properties result in the visualization of only part of the data so that the user is not aware of the global structures. Additionally, over time there will be more and more elements on-screen and overplotting emerges. Analyzing large datasets efficiently is not possible with this approach alone.

2.2.2 Visual Analytics of Large Graphs

Visualizing the Bitcoin transaction graph (TX-graph) falls into a subset of big data visualization because it is inherently a graph drawing problem. Graph visualization is comprised of several steps, where in each one can apply methodologies to increase performance. Data storage, preprocessing, visualization techniques, user interaction, and graph analysis are steps in the visual analytics pipeline that can be improved.

If large graphs are visualized without processing, they result in "hairballs" that diminish the efficacy of the visualization. So-called hairballs occur if the number of nodes and edges of a graph increases up to a point where structures within the graph become invisible. Figure 2.3 shows such a hairball of a Twitter hashtag. In order to be able to gain insights from graph visualizations and not overwhelm the user, it is important to apply tools and methodologies that reduce visual clutter. A wide array of methodologies in all of the previously mentioned stages of the visualization has been proposed. Scientists have found various solutions to the problem of overdrawing in graphs.

Edge bundling [40] is a strategy to reduce the number of edges in a graph. By visually aggregating edges that are close together and have the same general direction, it is possible to combine them into super edges. This in return, results in a smaller number of edges overall.

Network motifs [41] are another strategy to reduce the number of nodes. By finding recurring subgraph structures, known as motifs, in a bigger graph, these subgraphs can be collapsed into a super node, thus, reducing the overall number of vertices in a graph. Related to this technique is node clustering [42, 43] that is similar to motif clustering, but finds substructures in a graph according to different metrics.

Magic lenses [44, 45, 46] are an interaction technique where an arbitrarily shaped region can be selected by the user and the visualization within this region changes. This filtering does not reduce the number of visual elements. However, they are a useful tool to help the user to navigate large graphs by giving them a tool to zoom into regions of a graph. This is different to global zooming by providing the possibility to zoom into subregions of a visualization, similar to a magnifying glass.

Highlighting structures [47] is another interaction technique that does not reduce the number of visual elements, but aids the viewer in finding connections within a graph.

Novel visualizations [48] aim to mitigate visual clutter and overdrawing by not visualizing graphs as node-link diagrams, but rather in a completely new way. These visualizations can be very specific to a given problem and lose generality [15, 14].

If interactivity and timely visualization is not of concern, one can use top-down approaches that require preprocessing of the data. Aggregation hierarchies that have been built through a preprocessing step or that are naturally occuring within data, can be traversed through user interaction which helps users to understand data [50]. An alternative is to start with a known piece of information, such as a node or a structure, and expand the graph around this area of interest according to a degree of interest [51, 52]. With

FACETS, Pienta et al. [53] take large graphs as input and visualize them in a way that does not overwhelm the user. Their solution is to let the user choose a starting node and then through computing similarity and dissimilarity metrics in the neighborhood of this vertex, they expand the node-link diagram. Nodes either already traversed by the user or marked as similar or interesting (dissimilar) are also shown by the system. The metrics are computed comparing node feature distributions to the global feature distribution of the graph. Figure 2.4 shows this approach of combining node-link diagrams with computed metrics. While the user is expanding nodes the system also creates a user profile and incorporates it into the subjective interestingness calculation. Reducing the visualized graph in this manner avoids visualizing large graphs. However, this is in contrast to our work where we also visualize subgraphs but aggregate over data dimensions. Instead we aggregate the data through topological properties that do not provide any additional information. Section 4.3 details the aggregation steps of graph pruning, motif aggregation, etc., that are performed as part of this Thesis.

Apart from a pure bottom-up approach, it is also possible to follow a hybrid approach that combines aggregation with the expansion of the network. Stef van den Elzen and Jarke J. van Wijk proposed a system [29] that follows this approach. Their work combines multiple coordinated views that simultaneously show an aggregated view and a global view. In the global view not all data is shown. The visualization starts completely without edges and only shows them once a user has selected a region of interest. The selections are then aggregated and shown in a separate overview. Through user interaction, the graph can be further aggregated or expanded depending on the users' navigation in the data space. Their interface provides the user with a detailed and an aggregated view. However, this comes at a cost and the authors themselves note that if users have too many selections of interest, their approach fails and results in visual clutter. While hiding parts of the visualization initially helps mitigating the performance bottlenecks of the visualization pipeline, it also hides information from the user. Visualizing data in this way makes it difficult for a user to understand structural properties of the data. Figure 2.5 shows the visualization with a detailed view on the left and an aggregated overview on the right. Our work differs in the sense that it does hide elements of the network but only if they do not add any additional information to the visualization.

Numerous methodologies exist that address the same problems concerning the scalability in the visualization of big graphs. Often, aggregation strategies are domain specific and focus on a specific problem. Due to different properties of the data to be visualized, solutions that fit one situation can not be applied to our data. While imMens [27] is a system that can query big data and visualize the results, the necessary step of preprocessing the data makes it too static for a dynamic system such as Bitcoin. Another difference is that imMens operates on data different from graphs and therefore can not be used for Bitcoin. Given the size of the Bitcoin network with millions of nodes and edges, imMens' is not fast enough to be able to work with the blockchain as it does not scale to datasets of this size. Other systems proposed to analyze the dynamic TX-graph of Bitcoin, such as the work by McGinn et al. [13], address the issue of dynamic data.

However, this solution requires specialized hardware in the form of a render cluster and large display walls. The methodology of this thesis finds a place between highly dynamic and static systems with commodity hardware in mind.

2.2.3 Bitcoin Visualizations

Although the size of the blockchain data makes it difficult to analyze, there has been an active community working on the problem of visualizing the public ledger. One such solution is to calculate the statistical properties [11, 54, 55, 56], such as degree distribution, daily transaction volume, TX size, etc., of the system. These numbers can then be used to visualize phenomena (growth periods, company payment networks, user participation and so forth) in the network. Additionally, this approach allows the authors to visualize data over time to understand trends and developments. This can entail technical data that concerns itself with the health of the network or financial data that shows an overview of its usage. These visualization techniques include line plots, bar charts, and other typical visualizations from the statistics toolbox. Besides showing general properties [8, 9, 10, 57] of the network, these visualizations can also be used to answer particular questions about phenomena in the network. Kuzuno and Karam [9] use visualizations of the blockchain to specifically answer questions raised during an investigation of law enforcement. Specifically, their system is used to anwer questions in criminal cases of ransomware, marketplaces, and even DDoS extortion. They do so by visualizing the balance of a certain address over time, send and receive events per day, and hourly transaction count. They also plot this data over longer periods of time (weeks and months). Furthermore, their tool allows for querying blockchain data for transactions with user specified properties. Figure 2.6 shows one of their many visualizations they produce during the analysis of a Bitcoin address and its surrounding network. Their visualizations also allow the user to analyze the data on a transactional level, showing input and output addresses of specified transactions. Path finding between two addresses is also incorporated as a simple numerical output. Only the number of nodes and edges in a path from address **a** to **b** is shown.

Apart from visualizing statistical properties, it is possible to gain insight about the network by showing the payment transactions of Bitcoin as a graph. There has been work to extend purely statistical visualization with information about the underlying topology of the payment network [11, 54, 55, 56]. There have also been papers that explicitly focus on the underlying topology of the transaction graph in Bitcoin [12, 13]. Data preprocessing strategies for graph visualization include abstracting the data into a format that better suits analysis [11], enhancing the data with external information, or reducing the data due to performance limitations. Data reduction can happen through time constraints [13], topology connectivity, or other aggregation or pruning methods [12]. Pruning data either through heuristics or through filtering and leveraging of topology properties (connected components, subgraphs, etc.) is a simple, yet effective way, of combating the size of the dataset. Reid and Harrigan [11] focus explicitly on the network topology. In their paper, the blockchain data is preprocessed and split into two subgraphs representing the transaction network as well as the user network. Using a node-link diagram, even though only a small subgraph of the blockchain is analyzed, the authors were able to find interesting information about a BTC theft that has happened in the past. This shows that using topology information is a suitable tool to follow the flow of money throughout the Bitcoin network, despite the seeming anonymity Bitcoin provides. A big difference between their work and the methodology presented in this thesis is the fact that Reid and Harrigan have a preprocessing step involved. The raw data from the blockchain is taken and transformed into another graph representation that is then used for the visualization and the analysis task. Our work does not change the network topology by creating an artificial overlay network, instead it operates on the raw data of the blockchain.

In 2017, a paper [12] that aims to solve the difficulty of analyzing the Bitcoin TX-graph, by using a similar approach as proposed in this thesis, has been published. The proposed analysis system is split into a server-client model and draws inspiration from visual analytics to overcome the challenging size of the blockchain data. The approach is to filter the data heavily according to some heuristics and topologies the authors deem interesting. This includes filtering by time, transaction size, or addresses in regards to node degree or number of BTCs sent. This leads to heavy pruning of the TX-graph. In their use-case, they analyze a blob of 15,964 transactions, which they reduce by applying several filters. The final visualization is comprised of several ”islands” that fulfill the query criteria. Once such a subset is visualized, a user can apply more filtering to either highlight or remove visual elements. A result of this approach is that a user can see many disconnected subgraphs of the whole TX-graph. This hides the fact that some islands might be connected through paths that might be pruned by the filtering. The methodology of this paper therefore has the disadvantage that the global properties of the TX-graph can be lost during the analysis process. Furthermore, given that the authors are performing their computations on a very potent server with 128Gbyte of RAM and two Intel(R) Xeon(R) CPU E5-2620 v4 2.10GHz 8 core processors with a total of 32 threads, these hardware requirements limit the access and usefulness to analysts that might not possess such a powerful machine. Our work makes analysis of the Bitcoin network possible on off-the-shelf commodity hardware. This is possible by aggregating data and streaming it at the same time. Such a mixture of top-down and bottom-up approaches hides processing latencies even on less powerful hardware.

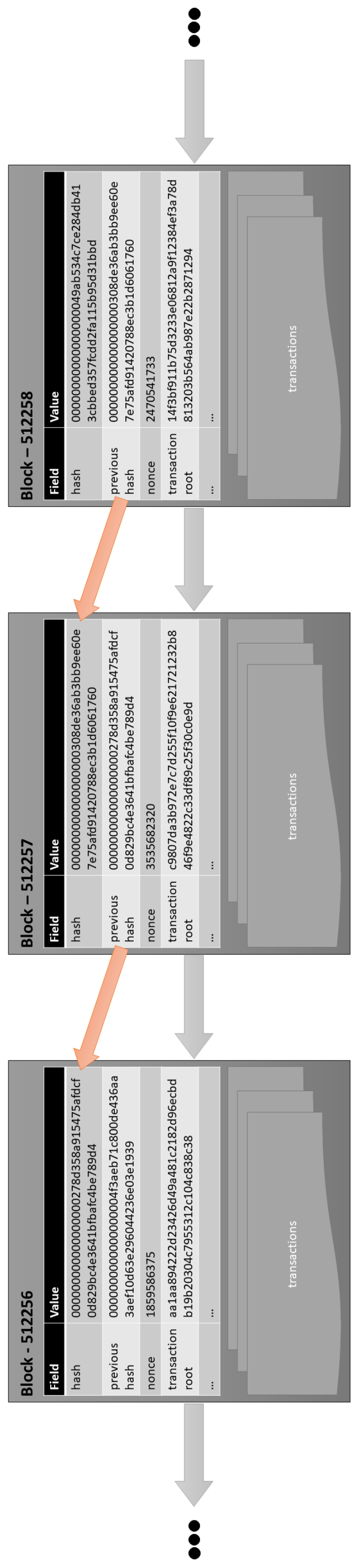
In contrast to pruning the data domain, one can also aim to visualize the not aggregated blockchain dataset and prune according to the temporal dimension. McGinn et al. [13] show visualizations of two fundamentally different data selections of the TX-graph. The first selection is the current mempool of the Bitcoin network. In the mempool are the currently published but not yet, through mining, validated transactions. The second data selection is done by choosing one particular block and visualizing the transactions occurring during this one validation period. Through this complete visualization, the authors can find anomalies within the blockchain on a large-scale level, as seen in Figure 2.7. However, due to the number of transactions in either the mempool or a block, the graph becomes hard to analyze on a fine grained level. This makes it feasible to analyze the Bitcoin network on a global scale at the expense of local, fine grained details. Additionally, the system is using specialized hardware to be able to show the visualization on a large scale. Their so called *data observatory facility* employs 64 screens and a distributed rendering cluster with a total screen-space of 132M pixels. This is in contrast to the work of this thesis where the explicit design goal is to leverage commodity hardware to analyze Bitcoin transactions in detail.

Apart from either pruning or using powerful hardware to overcome the difficulties posed by the amount of data, one can choose to use a different visualization technique that encodes the information differently [15, 58, 14]. Such an approach gives greater flexibility by compressing more information into smaller screen space. This can be very well suited for Bitcoin where there are many visual elements to be shown. However, the result is often a less intuitive visualization that initially requires users to learn how to read and understand the novel visualization technique.

By analyzing the money flow at the level of transactions, Battista et al. [15] were able to show the purity of a set of BTCs over time without overdraw of the node-link diagram. Purity refers to the certainty to which BTCs can be attributed to an address over the time of multiple transactions. While this allows an analyst to monitor whether a set of BTCs has been mixed with coins from another entity, all the topological information is lost and it is impossible to see transactions on a detailed level. In contrast, a node-link diagram is easily understood by analysts and does not require an initial learning period. Furthermore, it allows for the visualization of the complete TX-graph and does not hide information by ignoring graph topological information.

Instead of analyzing the purity of a certain transaction, it is possible to monitor the inputs and outputs, as seen in Figure 2.2, of a specific address over time. By assigning glyphs to the events of *incoming BTCs*, *outgoing BTCs* and *simultaneously incoming and outgoing BTCs*, Isenberg et. al [58] are able to visualize the state of an address over time in a clean way. While this visualization is easier to understand than BitConeView [15], it shows less information overall. By limiting the visualized data to only show the inputs and outputs of BTCs of a certain address over time, the applicability to a wide set of problems is limited.

Apart from the academic community, there are private entities that provide visualization services. While there are some visualizations that are publicly available and aim to showcase interesting data [59, 24, 60], there are also private companies that have a clear business model involving data analysis and visualization. As with scientific contributions, the publicly available collection of Bitcoin visualizations ranges from simple line graphs [24] to complex and rather artistic visualizations [61, 62, 63]. There are visualizations that focus on transaction data and try to visualize it to make it understandable. However, the visualizations are either limited in how much data is visualized [64] or by their performance, because too much data is shown [65]. The *Bitcoin Transaction Visualization* [64] is implemented as a ringbuffer and only shows 300 transactions as nodes in a force directed layout. It starts with an empty visualization and adds nodes and links for every TX that is published to the network. Once the buffer is full it removes the oldest nodes on-screen. With such a minimal number of nodes shown there are no performance degradations. Our visualization is able to process more than 23,000 nodes and 28,000 edges. The visualization from dailyblockchain [65] shows newly added unconfirmed transactions in the mempool. This visualization starts to suffer from performance problems when zooming and panning at a node count of 1500. General interaction becomes slow at 6000 nodes and even without interaction the frames per second degrade at 7500 visible nodes. The companies that sell a product to analyze the flow of money in the TX-graph also offer visualizations of the data analysis [66, 67]. The details of the methodologies used and the algorithms employed however are not publicly known and it is therefore impossible to retrieve performance metrics of commercial tools. This also stems from the secretive marketing on their websites that does not go into detail about the used algorithms or visualization techniques. Missing data on performance and the lack of access to the tools [68] without a commercial relationship make it impossible to compare these available tools with the methodologies proposed by the scientific community. The novelty of blockchain-technology and these commercial companies and products limits the available information. That means that, besides highlighting their existence, it is impossible to compare them against the methodology proposed in this thesis.



Block – 512258

Field	Value
hash	00000000000000000049ab534c7ce284db41 3cbbed357fcdd2fa115b95d31bbd
previous hash	000000000000000000308de36ab3bb9ee60e 7e75afd91420788ec3b1d6061760
nonce	2470541733
transaction root	14f3bf911b75d3233e06812a9f12384ef3a78d 813203b564ab987e22b2871294
…	…

Block – 512257

Field	Value
hash	000000000000000000308de36ab3bb9ee60e 7e75afd91420788ec3b1d6061760
previous hash	000000000000000000278d358a915475afdcf 0d829bc4e3641bfbafc4be789d4
nonce	3535682320
transaction root	c9807da3b972e7c7d255f10f9e62172121232b8 46f9e4822c33df89c25f30c0e9d
…	…

Block - 512256

Field	Value
hash	000000000000000000278d358a915475afdcf 0d829bc4e3641bfbafc4be789d4
previous hash	000000000000000004f3aeb71c800de436aa 3aef10d63e29604422 36e03e1939
nonce	1859586375
transaction root	aa1aa894222d23426d49a481c2182d96ecbd b19b20304c7955312c104c838c38
…	…

Figure 2.1: This Figure shows three blocks of the Bitcoin blockchain and how they relate to each other and form a chain.

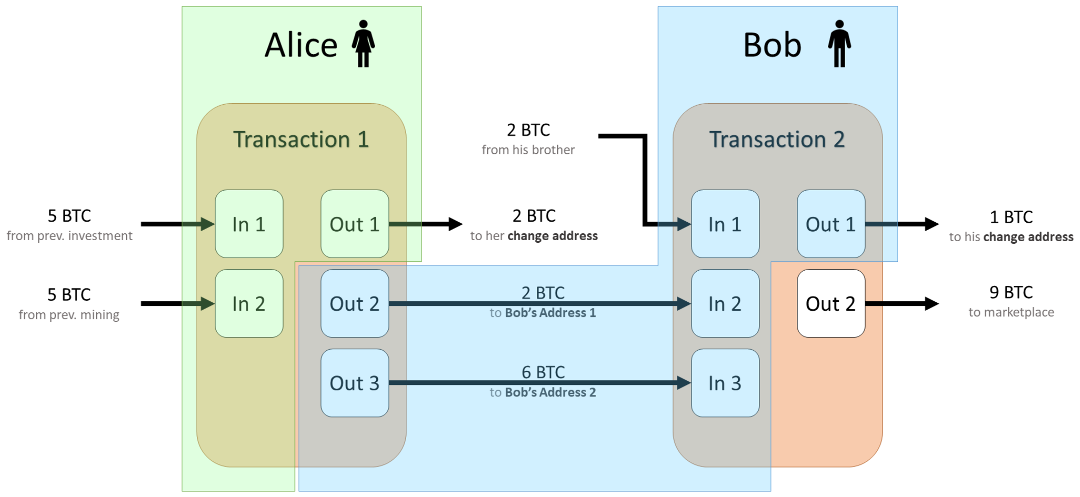


Figure 2.2: Structure of input and output addresses as they occur in Bitcoin TX. It is important to note that input and output addresses are not directly assigned 1:1. This generates a certain level of anonymity when transacting with Bitcoin.

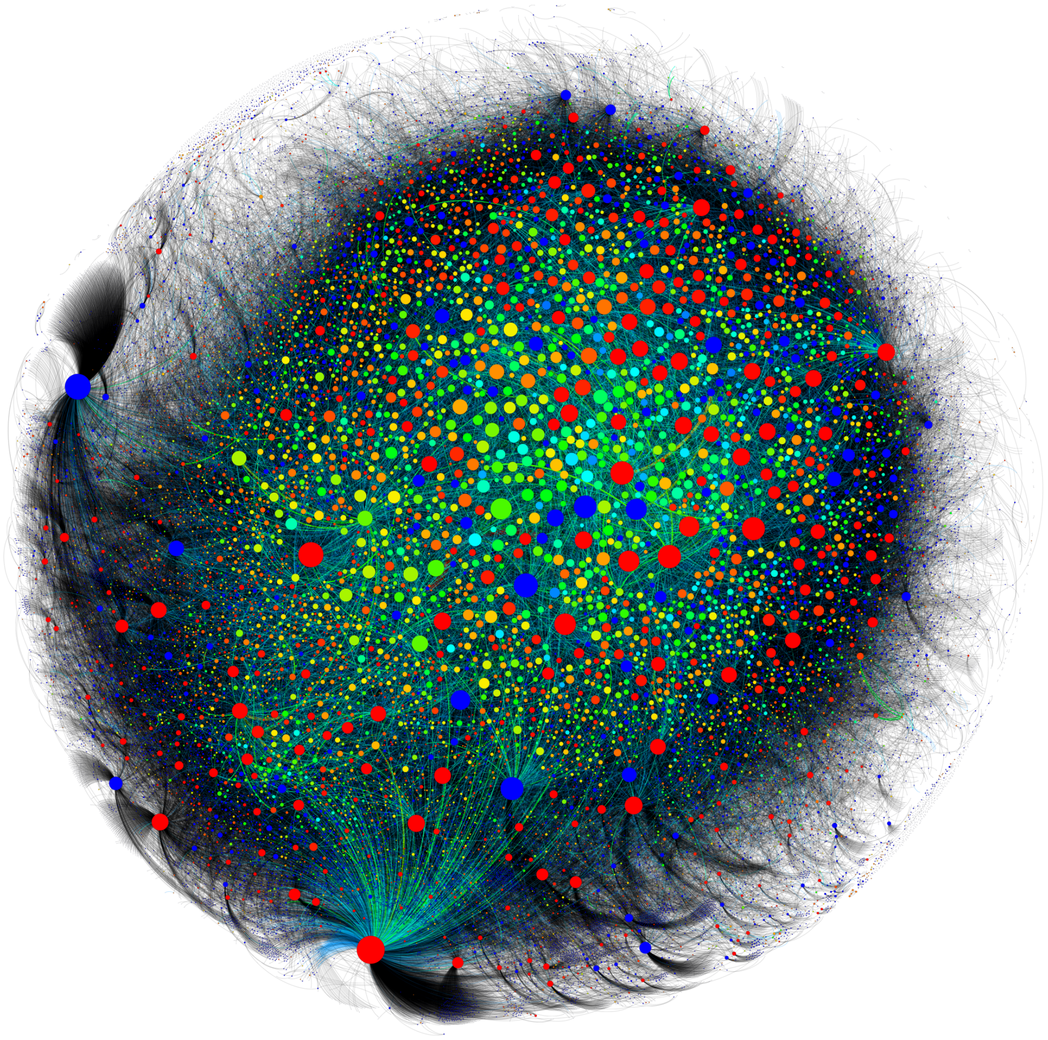
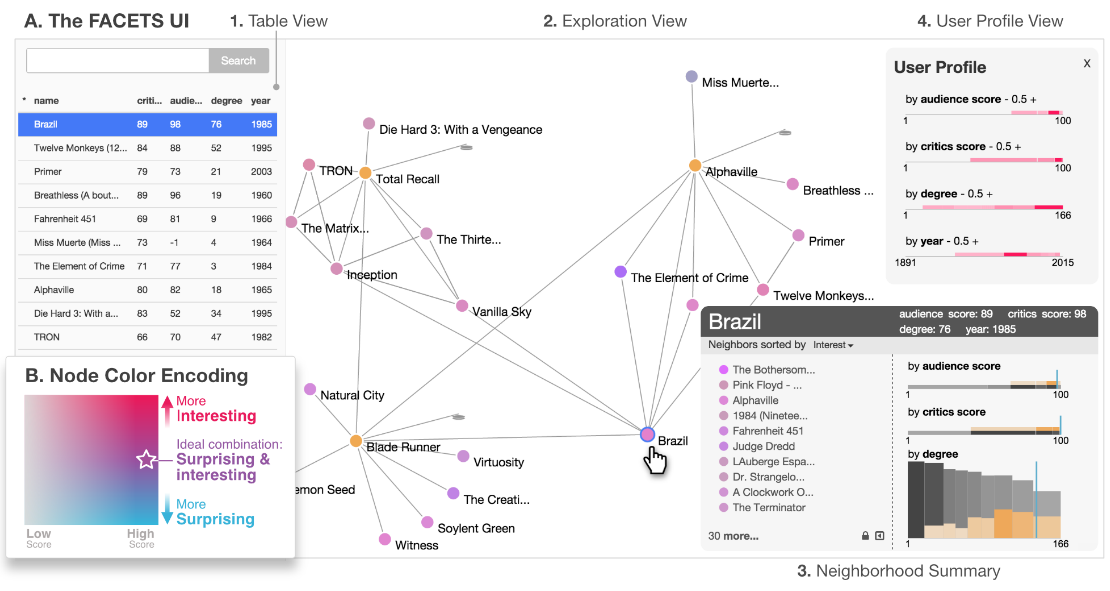


Figure 2.3: A graph hairball showing that overplotting graphs reduces their expressiveness. Structures and information are lost due to overplotting. This graph was visualized by analyzing gamergate hashtags on Twitter [49].



*	name	criti...	audie...	degree	year
	Brazil	89	98	76	1985
	Twelve Monkeys (12...	84	88	52	1995
	Primer	79	73	21	2003
	Breathless (A bout...	89	96	19	1960
	Fahrenheit 451	69	81	9	1966
	Miss Muerte (Miss ...	73	-1	4	1964
	The Element of Crime	71	77	3	1984
	Alphaville	80	82	18	1965
	Die Hard 3: With a...	83	52	34	1995
	TRON	66	70	47	1982

Figure 2.4: FACETS, a visualization approach that calculates degrees of interest and shows substructures of a graph accordingly. **B** shows the node color reference for a calculated metric. Orange nodes are nodes that have been manually expanded by the user. The other colored nodes are shown according to calculations performed by the system. The whole graph would have 17,000 nodes and 72,000 edges and if visualized without any processing would result in a hairball [53].

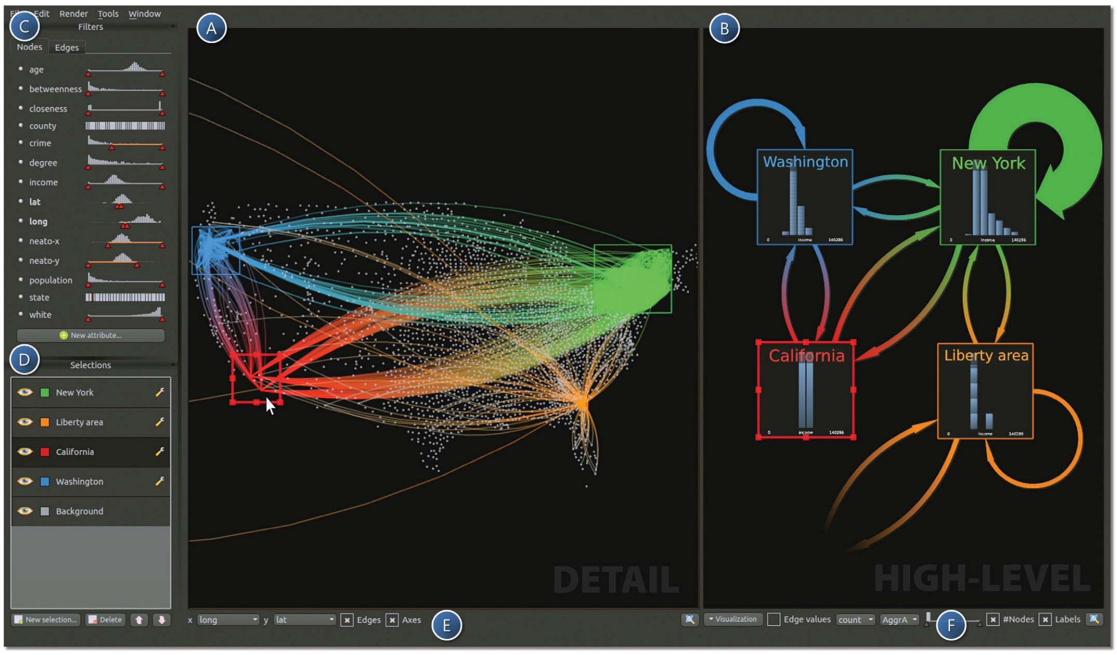
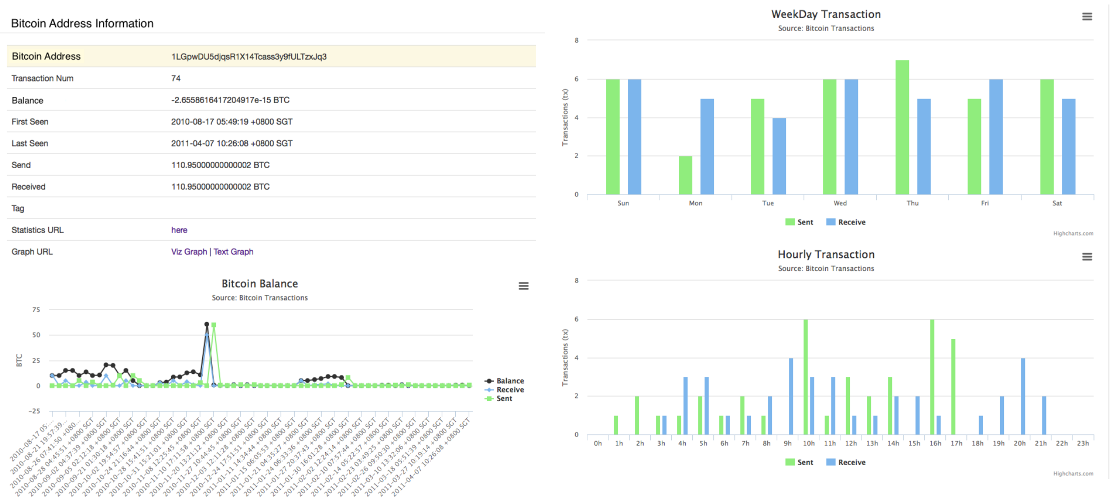


Figure 2.5: A bottom-up top-down hybrid approach as proposed by Elzen and Wijk. **A** shows a global view of the dataset. To overcome overdrawing, the authors simply start with a node-link diagram that does not show edges. Once a user selects an area of interest, the edges are rendered and the aggregation view is created and shown in **B** [29].


Bitcoin Address Information

Bitcoin Address	1LGpwDU5djqsR1X14Tcass3y9fULTzxJq3
Transaction Num	74
Balance	-2.6558616417204917e-15 BTC
First Seen	2010-08-17 05:49:19 +0800 SGT
Last Seen	2011-04-07 10:26:08 +0800 SGT
Send	110.95000000000002 BTC
Received	110.95000000000002 BTC
Tag	
Statistics URL	here
Graph URL	Viz Graph | Text Graph

Figure 2.6: Visualization of Kuzuno and Karam showing usage statistics of a certain Bitcoin address over time. [9]

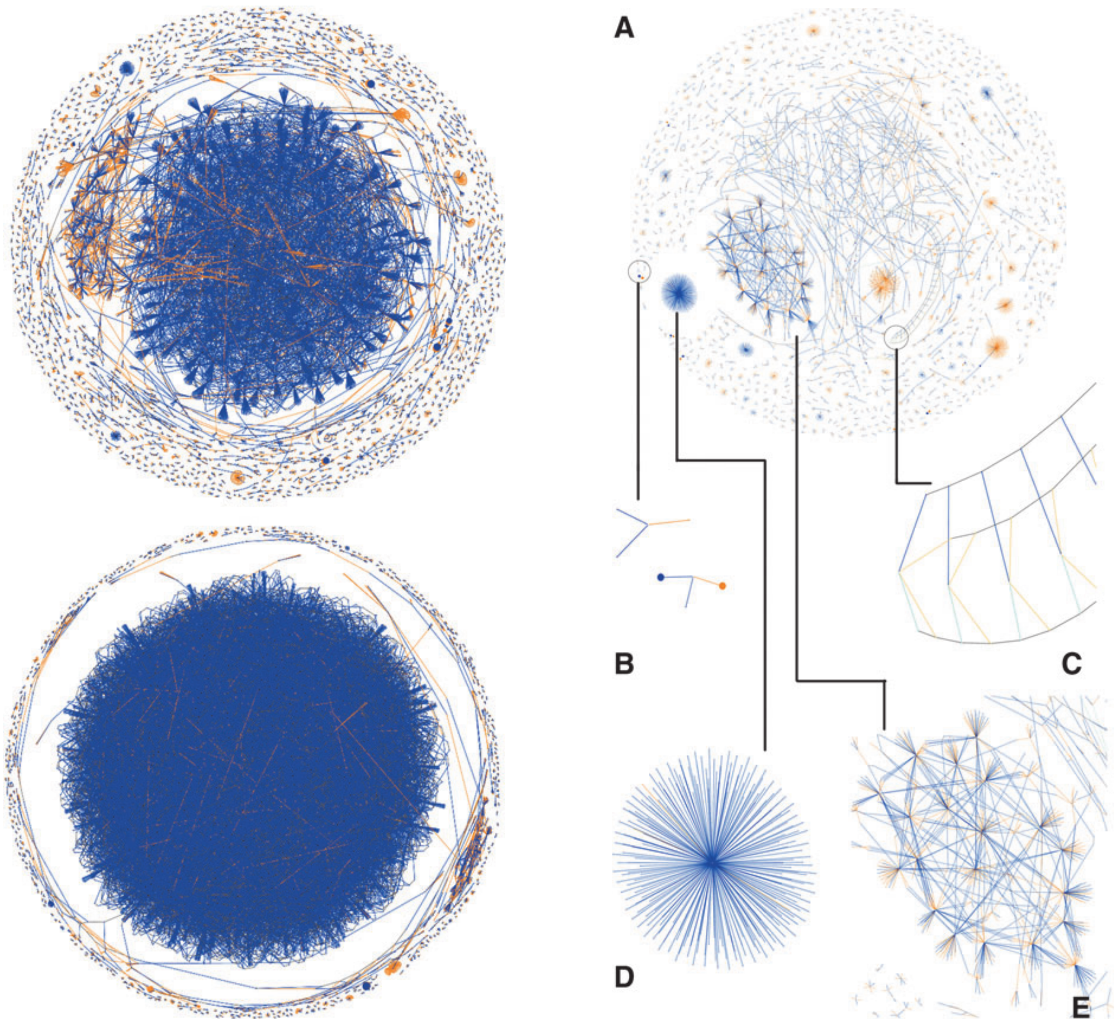


Figure 2.7: The left part shows the visualization of blocks 199884 and 232304 that were reported to possibly contain transactions of a money laundering operation. The right side shows another visualization with various subgraphs of one block of the blockchain. **D** shows a payout system with many recipients. **E** is possibly mixing service which is a service that shuffles BTC around. Generally, this manifets in a highly associated yet isolated structure in a block [13].

CHAPTER 3

Bottom-Up: Architecture and Processing Strategies

Instead of using a monolithic architecture for visualization and preprocessing, this thesis proposes a client-server architecture. This chapter explains the architecture in detail and also explains why certain designs choices were deemed the best fit for the Bitcoin data and the research questions outlined in Section 1.2.

3.1 Client / Server Architecture

Our proposed methodology uses a client-server architecture to gain flexibility. This increased flexibility allows for the use of hardware that best fits the respective stage of the visualization pipeline. Although no extraordinary hardware is necessary for data processing, this separation allows one to use thin clients that neither require powerful computational resources nor a large data storage for the blockchain. The client only needs enough processing power to be able to render the already heavily reduced data on-screen. The proxy server prepares the data and transforms it into a format that the client only needs to render. This reduces the computational workload for the client even further. Even though this separation exists, it is possible to have all the necessary parts of the system running on one machine. Figure 3.1 shows this separation into three different parts.

However, such a separation comes at a cost and requires an extra amount of communication between the individual parts. The whole system has two communication loops between its components. On a high level view, the client requests data from the processing proxy, which in turn requests the raw data from the database server. Once the database server returns information, it is sent to the proxy where it is processed and transformed into a format suitable for rendering by the client. After the proxy has processed the data, it is

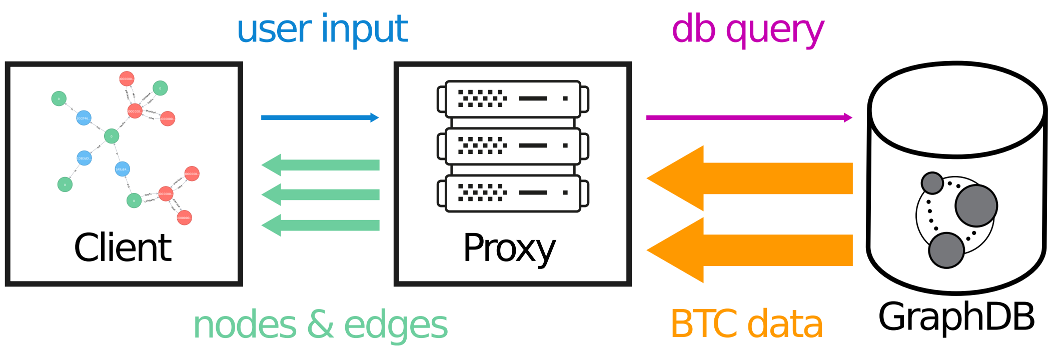


Figure 3.1: Control and data flow between the client, the proxy and the database server. User interaction triggers a database query. Once data is received the proxy processes the data and upon its completion sends the data to the client for visualization.

forwarded to the client which then renders it. This means that the client already receives data in the form of nodes and links that it only needs to render on-screen. Internally in the proxy server there are two communication loops. One that handles data only and another that is used for sending commands and responses. Having a less complex channel for data reduces the delay and the complexity of progressively streaming data to the client. Between the proxy server and the external components are two communication channels, one between the client and the proxy and another one between the proxy and the database server. The example of a control and data flow can be seen in Figure 3.2 and its corresponding architecture in Figure 3.3.

3.2 Client

The visualization client presented in this thesis provides a static visualization that is able to showcase the data processing capabilities of the server. Therefore, it is minimal with only limited user interaction and a simple structure. It is the gateway for the user to send queries about the blockchain to the proxy. The client supports two different request types: one to show the neighborhood around a specified node and another one to find paths between nodes in the TX-graph. Once the client receives an answer, it simply visualizes the data. The communication loop with the proxy server includes requests, with the necessary parameters, status responses from the server and data ready for visualization. Figure 3.2 shows the message structure between the client, the proxy server and the graph-database server. Once the client has sent a request and triggered progressive streaming, data is constantly loaded from the database and processed in the proxy and lastly sent to the client to update the visualization.

There are many different possibilities on how to visualize graph data. Two widespread techniques are rendering adjacency matrices or node-link diagrams. In this thesis the visualization is realized as a node-link diagram with a force-directed layout and directed

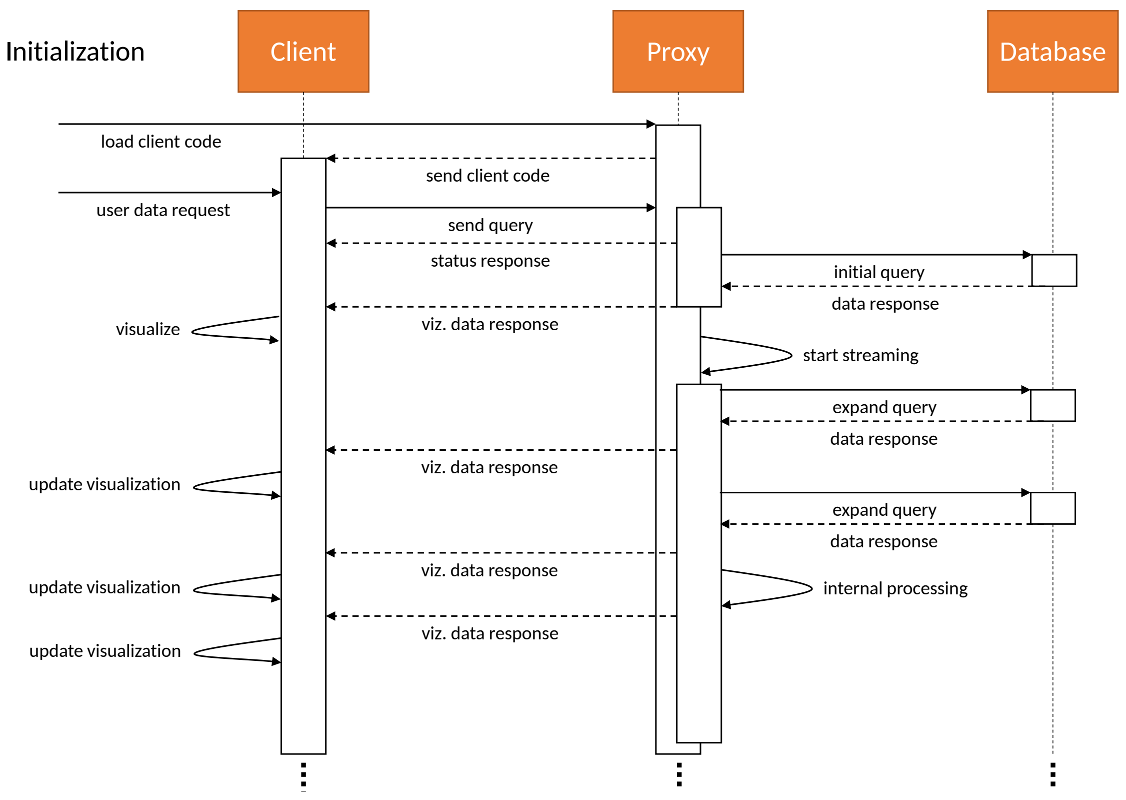


Figure 3.2: Interaction diagram showing the general communication between client, proxy server and database. Upon loading the client code from the proxy server and initiating a query, data is constantly streamed and processed from the database. Each time additional data arrives from the proxy server the client updates the visualization.

edges. There are three basic types of nodes that correspond to the blockchain data structures: blocks, transactions and addresses, that are colored blue, green, and red respectively. All edges are directed and multiple edges between two nodes are possible. This can occur between two blocks, a TX and a block, or between a TX and an address. The edge label gives additional information to understand the TX-graph. The directionality of TX and address edges is in the direction of value flow. Directed edges are shown as arrows from source to target node. Edges marked as "in" signal that an address is used as an input in the TX at the end of the edges. The same holds true for output edges labeled as "out" respectively. Although timestamps are contained in block nodes, the blockchain structure, as seen in Figure 2.1, is apparent in the edges connecting blocks with each other. This relative ordering uniquely identifies a successor and a predecessor for each block. Additionally, the visualization shows special nodes with visual markers (colors and strokes) to distinguish them from regular nodes. These special nodes are the result of the user query (start and end node of a path query) and the processing steps in the server (hairball centers and motifs from the aggregation and motif search).

While node-link diagrams show the structure of the Bitcoin network, the temporal component is difficult to convey in a force-directed layout. Therefore, we provide a second layout algorithm where the nodes in the visualization are not only positioned according to attracting and repelling forces, but rather by a time component. To do so, the time information in the transaction is mapped onto the x-axis. When the data is received from the proxy, the client calculates the minimum and maximum time of all the transactions it has received and then distributes all transactions along the x-axis according to their time data. This introduces the time component into the node-link diagram while still giving structural information of the graph. A user can switch back and forth between the different layouts as desired by pressing a button. For the proof-of-concept visualization this is the only interaction possibility for users.

When the server component streams data to the client and data is received, it checks whether nodes and edges have to be removed from the visualization. After removing elements that are not longer required and adding new elements to the cached data, the visualization is updated. The new nodes and edges are added to the visualization and the positioning computation of the force-directed layout is restarted to account for the new elements in the visualization.

3.3 Server

The main contribution of this thesis is the middle layer preprocessing the data in the processing server. Given the size of the blockchain data, performance was a central part of the design. Apart from speed, the requirements of the server were flexibility and the possibility to easily enhance the system with additional functionality. Therefore the server architecture itself is split into several parts. This allows for simultaneous processing and independent functionality for extending the functionality of the system.

Modular Architecture

A central principle for the architecture of the proxy is parallelism. Due to the algorithmic complexity of certain graph operations, they take a long time to complete. Therefore the workload is parallelized in order to avoid idling of computing hardware. A second factor for this design choice was responsiveness. Once a query has been sent to the server, the client or a user wants to have full control over the processing pipeline. Short reaction times to commands by the user are necessary to minimize the perceived delay in the visualization client.

To comply with these design goals, the server architecture itself is split into three main parts. Each of them is running in their own process completely independent of each other. This architecture allows for constant listening for client commands, while at the same time, being able to send data when processing a chunk of the data has been finished.

The second central part of the server is a control process that is able to control the server and computing process. Its responsibility is to control startup and shutdown, as well as synchronization between the processes. Finally, the data processing module controls the database connection and the data processing logic. Figure 3.3 shows the three parts of the server architecture, as well as the communication with each other. A detailed view on an implementation level will be given in Chapter 5.

Message Passing

To synchronize all parts of the proxy server and the data processing, there is a lot of communication involved. Splitting the server into three parts results in having three specialized components.

Communication Module The communication part of the server is responsible for providing a communication interface to the client. To allow the client to send commands at the same time as data is processed and sent, a non-blocking, two-way communication channel is established. Through this channel, the client can send requests and receive resulting data at the same time. Another important part is the initial delivery of the client code in order to provide an easy-to-use solution for users. This part of the proxy handles all the communication with the client.

Control Module Once a user sends a request and it is received at the communication endpoint, all necessary computations need to be started and the communication module as well as the data processing module of the proxy need to be synchronized. The responsibility of the control module is to establish communication between the data processing module and the communication module. Furthermore, it needs to control internal states and commands. Due to the fact that data processing can be computationally expensive, it is necessary to have an external component that is non-blocking and is able to control the process flow.

Data Processing Module After the control module has initialized the necessary internal state, it passes the client's query on to the data processing module. This module takes care of database communication, preprocessing and controls the incremental streaming of the processed data to the client. This module also takes care of all the necessary data conversions from database results to the internal representation and finally to a format adequate for visualization in the client.

Figure 3.3 shows the message passing between all the components including the internal channels of the proxy. First, a client loads the code from the proxy server. Once the code is loaded, it opens a bidirectional communication channel and the user can send a query to the server. The communication module accepts the request and forwards it to the controller. The controller processes the query, performs necessary actions and starts data processing. Once the data processor has received the query, it checks which type of data is required. It then sets its internal processing state and formulates an appropriate initial query for the database. Once the database returns data it is processed and sent directly to the communication module which then forwards it to the client. Upon receiving the data, the client visualizes it and listens for further data or waits for user input. Figure 3.2 shows an example of how such a message passing could occur between the client, the proxy and the database server.

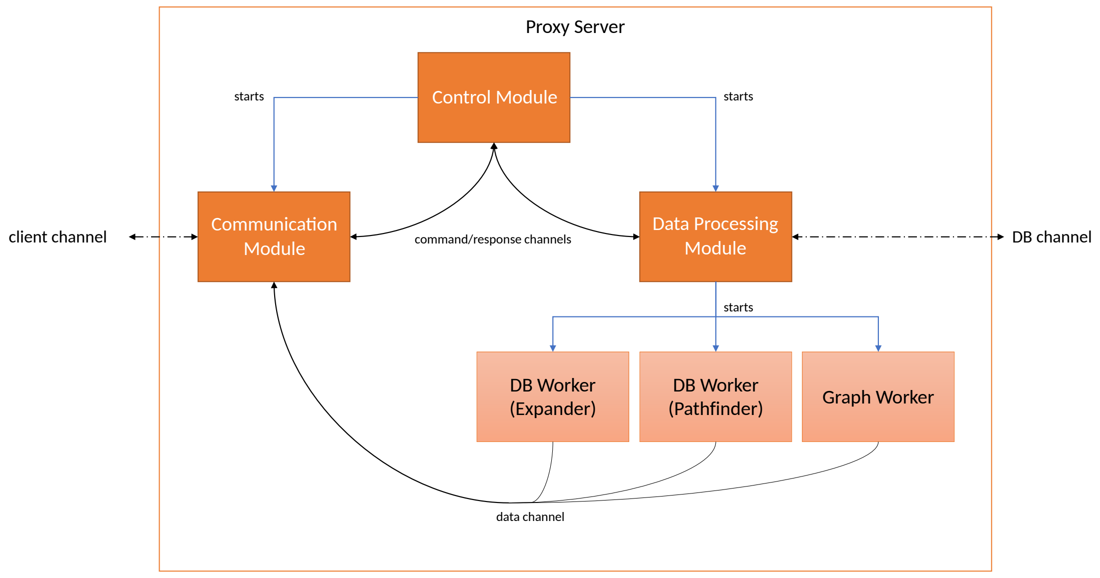


Figure 3.3: Message passing diagram of the internal proxy modules. The control module starts the communication module and data processing module for simultaneous processing. When the client sends a request, it is forwarded to the data processing module through the control module. The data processing module generates a database query and processes the returned data. After aggregating the data and transforming it into a format suitable for the client, it is sent directly to the communication module. This module then forwards the data to the client asynchronously.

Progressive Loading

For optimal performance and hardware saturation, the data processing methodology of this thesis puts a focus on parallelism. This means that, although the client sends only one single query, the proxy server splits the query into different subqueries. This helps to fully utilize processing power and reduces stalling while the server waits for responses from the database or costly graph operations. Depending on which query the user has performed, different parallelism strategies are employed.

The user can query the neighborhood of a node in the TX-graph for visualization. The TX-graph contains the data of the Bitcoin blockchain (blocks, transactions, and addresses) after mapping it to a nodes and edges. Section 4.1 gives a detailed view on how this data is mapped. Querying the neighborhood of a node gives the general surrounding structure of it inside the TX-graph and shows how it is embedded in the Bitcoin network. Although this type of query is also used for the more complex path query provided by the proxy, in itself it can also be interesting to see how the node interacts with the network. The general algorithm for this neighborhood expansion is as follows:

1. The client sends a query with a hash **A** of a node as starting node to the proxy.
2. The proxy initializes its state and sends a query to retrieve all neighboring nodes of **A** to the database.
3. When the database query returns data, the resulting nodes are added to an internal in-memory graph representation for further processing. Additionally, the proxy puts the resulting nodes in a queue for further processing. Finally it transforms the data into readily visualizable nodes and edges and sends them to the client.
4. The client receives the information and visualizes the graph as a node-link diagram.
5. The proxy server keeps expanding the neighborhood structures by querying the neighborhoods of all nodes in the list of nodes to process next. Depending on the size of the list, only the oldest parts of it are processed in a first in, first out (FIFO) order. If the database query returns new neighborhood nodes, these are added to the queue, unless they have already been processed. Next they are added to the in-memory graph of the proxy server. Lastly, the data gets sent to the client and the visualization is updated.
6. This process is continued until the client sends a termination request to the proxy and the controller shuts down the processing module.

Apart from querying the neighborhood of a node it is also possible to query for paths between two nodes. If the user sends a *path query* to find possible paths between two Bitcoin addresses, the system starts a database path search, a neighborhood expansion, and a path search in the internal, in-memory graph of the proxy server. Every time one of these threads returns data, it is immediately forwarded to the client to update the visualization. This results in the following process:

1. The user sends a query, searching for paths between address **A** and **B**, to the proxy.
2. The proxy initializes its state, and formulates a database query that looks for a path between those two addresses and then sends it to the database.
3. The database returns a single path between the addresses and the proxy adds it to its in-memory graph.
4. Next the proxy transforms the data into nodes and links and sends it to the client, which visualizes it.

5. Without any further requests from the client, the proxy server starts two additional processes that constantly keep loading more paths from the database and looking for paths in the internal graph representation. At the same time, the proxy puts the nodes of the initially found path into a list for neighborhood expansion and starts processing this queue.
   a) For each of the nodes in the list, the nodes with distance one are queried. Once the data is returned from the database, the new nodes are added to the list and the internal graph. Furthermore, an update with the new and updated nodes is sent to the client.
   b) The proxy queries the database for paths that have not yet been found in the graph. If such a new path is found, it is added to the list of known paths and the proxy server's in-memory graph. Then, the nodes of the path are added with highest priority to the list for neighborhood expansion. Lastly, the update of the graph is sent to the client.
   c) While the proxy server is processing data, it constantly searches for new paths in the cached in-memory graph, induced by the neighborhood expansion. Finding paths in the in-memory graph is done through a breadth-first search [69]. If new paths are found, their nodes are once again added to the node list. Additionally, they are marked as known paths for the database path queries and, finally, sent to the client.
6. This continues until the client sends a stop or termination request to the proxy.

Constantly sending visualization updates to the client helps to mitigate initially long loading times. For users, this results in the perception of no significant loading times and most importantly, interaction and analysis possibilities while more data is being streamed. If the surrounding network of a node is very large or finding paths between nodes takes a long time, the user is provided with information to analyze while in the background, the proxy server continuously loads more data. Although this approach does not give the complete information to the user at once, it reduces waiting times for the client significantly. The longer the client waits, the more information it gets and the closer the information will be to the complete data [31]. However, this methodology requires an initial query. A starting point, or a set of interesting nodes, known to the user, is necessary for this methodology to work. Alternatively, even when the user does not have information to start with, it would be possible to choose a random starting point for the query. However this might not be ideal for the user's task at hand and this problem will be discussed in Chapter 7.

Another aspect to note is that, while in the description above the step of sending data to the client is kept brief, it is more involved than previously explained. To be able to visualize Bitcoin as such a large dataset without overwhelming the client, the graph data is processed and reduced before sending it to the client. The steps involved in the sending process in the data processing module of the proxy server are explained in the following Chapter.

CHAPTER 4

Top-Down: Graph Data Processing

Besides the bottom-up approach of progressively streaming data to the client, reducing the number of nodes and links to be visualized is necessary to be able to handle the size of the blockchain. Dividing the data into chunks and sending them to the client piece by piece does not reduce the amount of data. After some time of streaming, rendering will still be a bottleneck and the client visualization will result in visual clutter. To mitigate this bottleneck, in the rendering stage of the visualization pipeline, reducing the number of data items is essential. The blockchain data runs through several phases of transformation throughout the whole pipeline. In this Chapter, the graph minimization and filtering during data analysis is highlighted.

4.1 Bitcoin Data

The source data for the visualization is the Bitcoin blockchain. The blockchain saves all the transaction information in the Bitcoin network. It consists of three main entities, *blocks*, *transactions* and *addresses*. Each of these entities contains information that gives a comprehensive view of the state of the network.

Figure 4.1 shows the fields contained in each block. *HashPrevBlock*, *nVersion* and *nTime* are parameters for the general state of the network. *HashPrevBlock* is a pointer to the previous block and this linkage is what makes the chain of blocks a **blockchain**. *nBits* and *nNonce* are important fields for miners. *nBits* gives the difficulty of the mining process in which new coins join the system. *nNonce* is the variable field miners can use to change the input of the hash function and therefore the resulting hash. The combination of these two fields gives miners the information they need to validate new blocks through the so called "proof-of-work" [70]. *HashMerkleRoot*, *#vtx* and *vtx[]* are the information about transactions between Bitcoin users, included in the blocks.

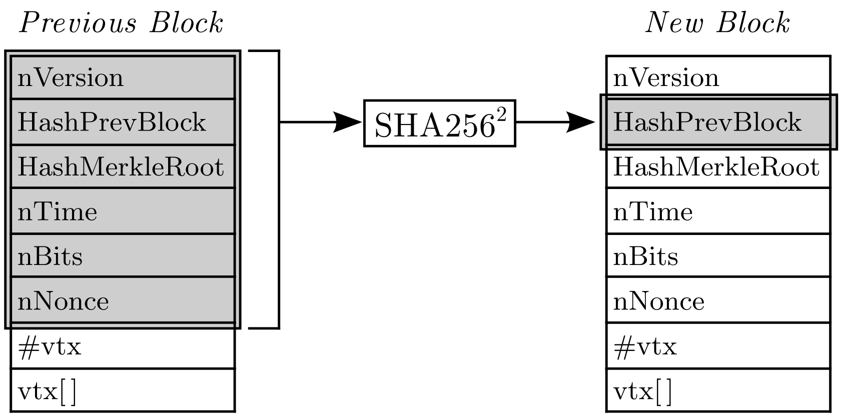


Figure 4.1: The Bitcoin block structure with all header fields and the reference to the previous block. *HashPrevBlock* is the unique reference to the previous block in the data structure. To obtain this hash, the header information of a block is taken as input of a $SHA256^2$ function. *nNonce* is the variable field for validating new blocks and mining new BTCs. For a full description of the included data the interested reader can refer to the Bitcoin Developer Reference by Krzysztof Okupski [18].

The first transaction in each block is called a "coinbase transaction" and it is a special transaction with no inputs and where the miner can choose the output address. This creation of BTC is meant as an incentive for miners to perform the costly validation of new blocks because the miner can send the newly created BTC to herself. The other transactions that are included in the list are transactions the miner chooses to include. Participants of the network can publish their transactions to the network and miners then decide which they include when they publish a valid block. As can be seen in Figure 4.2, transactions themselves contain several fields. *scriptPubkey* and *scriptSig* are necessary to proof to the system that a transaction posted to the network is valid and the BTC are under control of the sender. Generally, *scriptPubkey* contains an *address* that is tied to a cryptographic key and can be considered as being similar to a bank account number. For a transaction, the fields *hash* and *n* identify which output of a previous transaction belongs to an input.

4.2 Blockchain Data to Graph Mapping

With *blocks*, *transactions* and *addresses* as nodes, it is possible to map the Bitcoin data to a graph. Decoding the references of the inputs of *transactions* and the *addresses* contained in the *scriptPubkey* field gives edges between *transactions* and *addresses*. Conceptually, this is equal to what happens during sending BTCs from one user to another as shown in Figure 2.2. To show which *transactions* are included in which *block*, another set of edges is introduced. Special coinbase transactions that add new BTCs to the system are connected to *block* nodes by an additional edge, signaling this special case. For temporal mapping between a *block* and its preceeding *block*, a directed edge can be added between

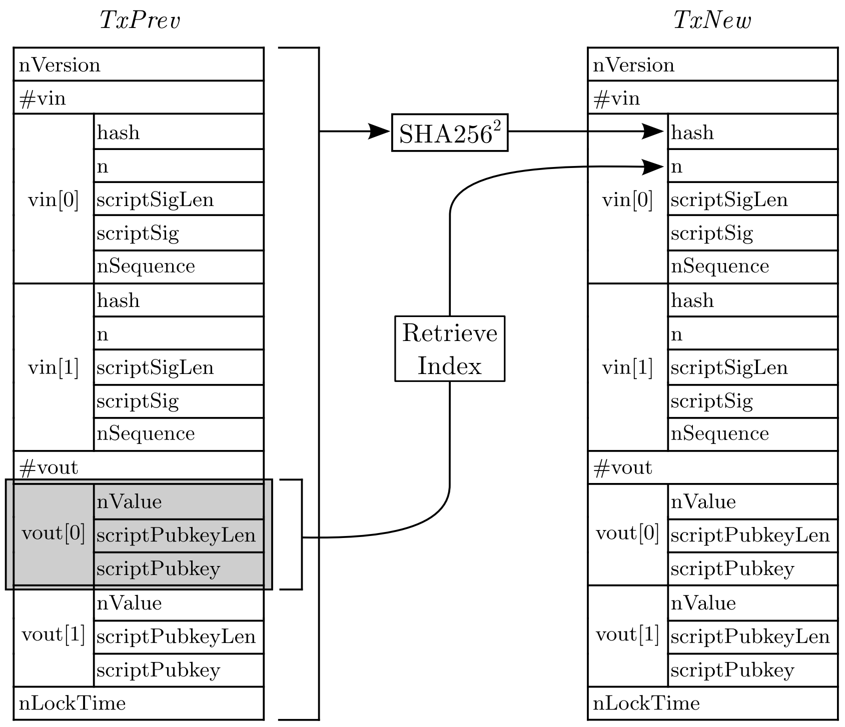


Figure 4.2: This Figure is taken from the Bitcoin Developer Reference and should only illustrate how two TX relate to each other. Both TXs have two input and two output addresses. *TxNew* shows that in order to identify a previous source of BTC for a transaction, the fields *hash* and *n* of *vin* are necessary. *Hash* identifies the previous TX and *n* gives the exact output address [18].

them. Mapping the data fields of *blocks*, *transactions*, and *addresses* from the blockchain to the nodes and edges of a graph results in a graph structure as shown in Figure 4.3. Green nodes are *blocks* with a uniquely identified predecessor and successor. *Transactions* are colored in blue and are connected to *blocks* by two distinct edge types. One edge label signals in which *block* the transaction is included. If an additional edge with the edge label "coinbase" exists between a *TX* and a *block*, then this transaction is marked as special; meaning that it does not have any input addresses. This separates special coinbase *transactions* from standard ones. For example the first blue node from the left in Figure 4.3 has two edges between itself and the green *block* node and is therefore marked as a special coinbase transaction. Directed edges connecting to the red nodes, or *addresses*, contain the information whether an *address* participates in a *transaction* as output or input. Both cases are simultaneously possible as seen in this Figure.

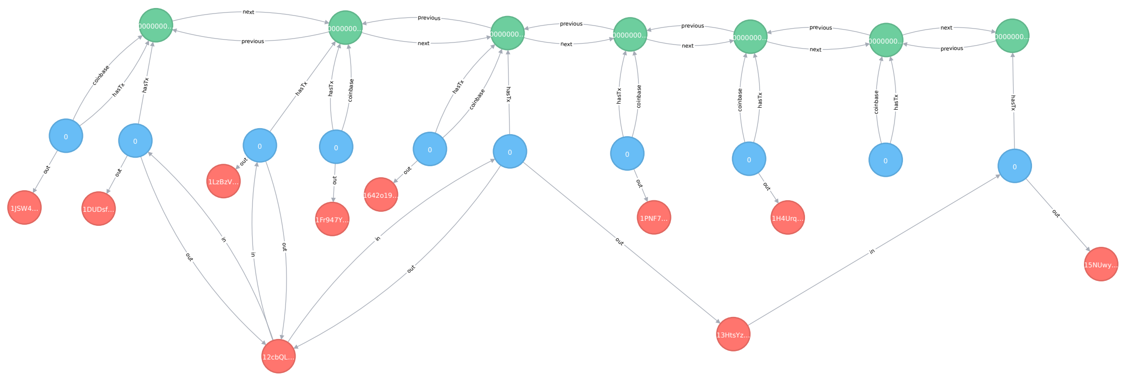


Figure 4.3: This Figure shows how mapping the Bitcoin blockchain to a graph structure looks like. Three different node types, blocks, TX and addresses are shown with different colors and are connected with directed edges.

4.3 Data Processing Steps

As described in Section 3.3, sending responses to the visualization client involves reducing the amount of data to be sent. This happens in several steps throughout the visualization pipeline as shown in Figure 1.2. On a high level, the complete graph data is reduced step by step by applying heuristics that are suitable for Bitcoin. Data reduction entails pruning of nodes that do not give additional information as well as aggregating nodes that do not contribute to understanding of finding paths between nodes or analyzing how a graph is embedded into the Bitcoin TX-graph. Visualizing the complete TX-graph is not possible due to its size and therefore these steps are applied to reduce the size. The Bitcoin network is very active and currently more than 150,000 new TXs are included in the blockchain every day.

4.3.1 Large Graph

Our dataset contains only a subset of the whole blockchain, spanning from Sat Jan 03 18:15:05 2009 UTC to Thu Jul 29 20:52:49 2010 UTC, and already exceeds the possibility of visualization on commodity hardware. During this timespan, a total of 71,036 blocks has been mined, containing 97,254 TX with 94,270 unique addresses. This results in a total of 262,560 nodes in our graph. These nodes are connected with a total number of 470,201 edges. It is not feasible to try and visualize the complete graph with interactive visualization in mind. Therefore, it is necessary to reduce the amount of data that reaches the rendering stage of the visual analytics pipeline. However, with our goal of finding possible paths between two addresses, to follow the flow of money in the Bitcoin TX-graph, we do not need to visualize the complete graph. For the second use case of visualizing the neighborhood of a certain node, it is also possible to apply aggregation and pruning.

4.3.2 Graph Pruning

Given knowledge about the TX-graph, an easy solution to reduce the number of visual elements is to prune nodes that do not add information to the visualization. As mentioned in the previous chapter, starting from a user query the data gets expanded. Neighborhood expansion is one part of this process where neighborhoods of nodes are loaded from the database. When that happens and a node only has one single neighbor, it does not add necessary information to the visualization about the TXs between **A** and **B**. Such a node can be omitted without providing the user with less information, but increasing the performance of the overall system. Apart from this default pruning strategy, it is possible for the client to specify a limit for the degree of the nodes to be pruned. Nodes with a degree below the specified limit are omitted from the visualization. This is a simple way to only retrieve a graph with highly connected nodes. However, this might lose information about possible paths and is therefore not suitable for finding paths between two nodes.

Another pruning strategy for finding paths between addresses is to only visualize the TX-graph. By omitting blocks from the visualization, the number of nodes as well as edges, is reduced. Although the number of transactions and addresses exceeds the number of blocks in the dataset, the total number of nodes is still quite significant. Therefore, hiding blocks in the visualization is another useful strategy to improve performance without sacrificing information for finding the flow of BTC from one address to another.

4.3.3 Graph Aggregation

Apart from simply removing nodes, another strategy to reduce the number of data elements is to aggregate them. Although aggregation in the data domain is possible [27], the goal of this thesis is to show the TX-graph's network topology. Therefore we use several types of topological aggregations that fit well for the Bitcoin graph data and do not hide necessary nodes and edges in the client visualization for the given user tasks. Through visually inspecting the TX-graph, it was found that the previously mentioned hairballs are very prevalent in the visualizations. Another common theme was the occurrence of certain subgraphs, called network motifs [41].

Hairballs The main use of sending a Bitcoin TX is to transfer value from a user **A** to a user **B**. Although this can be done with a minimum of two addresses per TX, senders can make TXs use as many addresses in one TX as the network protocol allows. This results in transactions that have hundreds or more participating addresses. If a majority of these addresses have been one-time use addresses used for storing BTCs over a long time, they do not contribute much to the general topology information of the TX. Therefore, it is possible to collapse these nodes into the TX and visually mark it. Such a reduction is shown in Figure 4.4. If even just a few transactions in a graph are of this structure, many nodes can be aggregated and saved from being displayed. An analysis on how significant this reduction in size is, is given in Chapter 6.

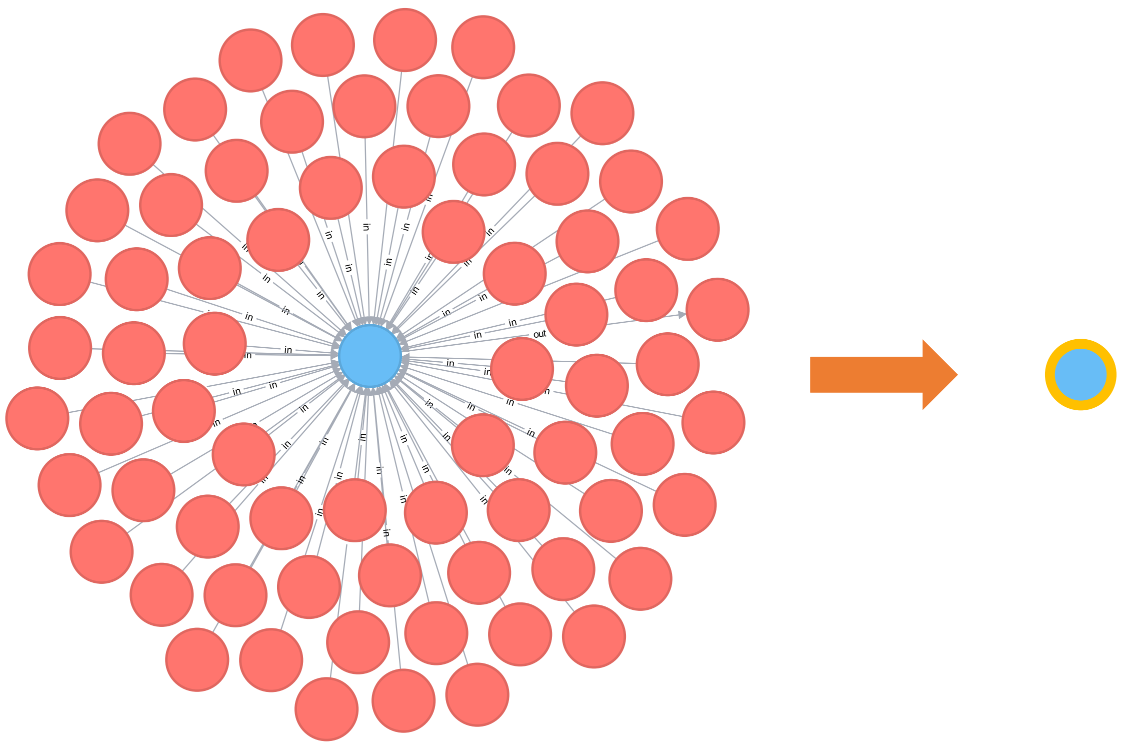


Figure 4.4: A node n containing many neighbors with $degree(n) = 1$ has its neighbors removed and is then marked as a special hairball node.

Motifs It is possible to search for recurring subgraph structures in a graph and replace them with super nodes [41]. Super nodes are nodes that are added to a graph when multiple nodes and edges are collapsed into one node. In the case of Bitcoin, user behavior emerges in specific transaction (TX) patterns, that in turn result in so-called motifs. The methodology proposed in this thesis searches for two distinct motifs that are interesting from a Bitcoin analysis point of view.

- **Self Sending Loops:** If a user sends BTCs to another address and then back to the original sender address, this behavior results in a loop in the graph visualization. For finding paths between two nodes, removing these motifs does not hide any vital information. Even if a path would incorporate such a self-loop, pruning these structures only results in a shorter path without omitting information. When a node that is source and target of a self-loop is marked as such and the loops are removed, the number of nodes in the final visualization is reduced while providing all structural information. Such a motif as well as its replacement can be seen in Figure 4.5.
- **Fan-Structures:** When addresses are reused to send BTCs to another addresses through multiple transactions, a fan-like pattern occurs. Figure 4.6 shows this motif in a graph taken from the real Bitcoin data. If considered

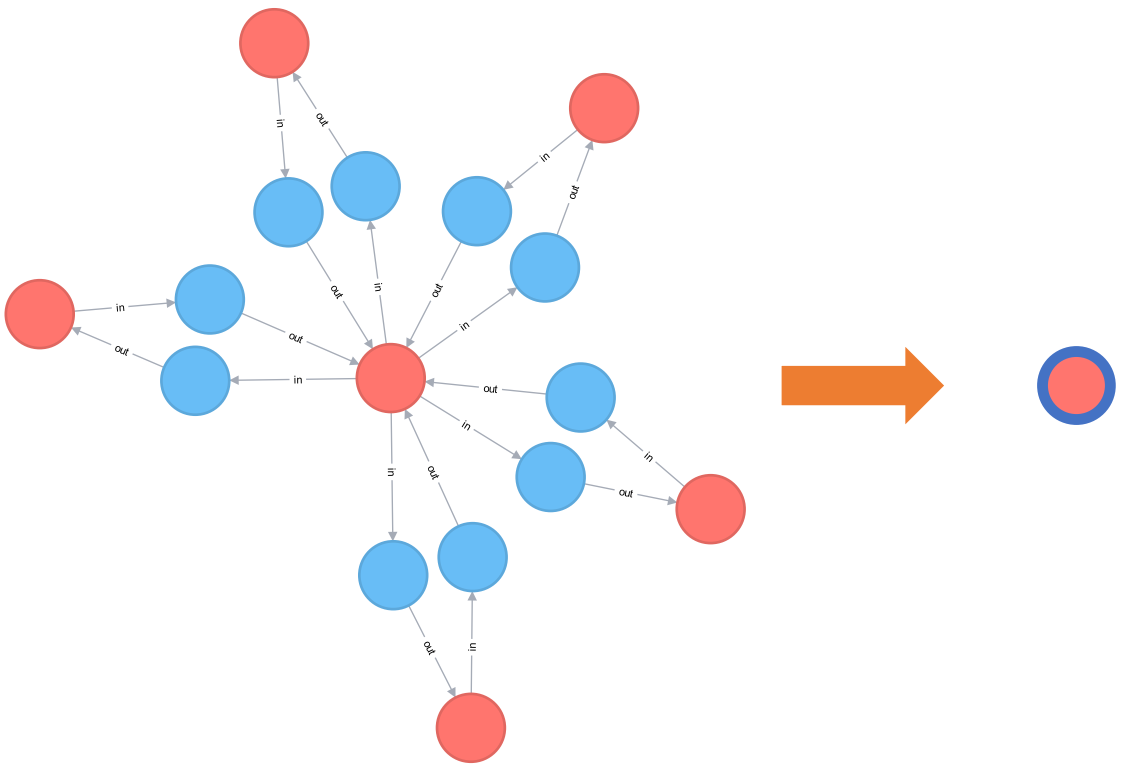


Figure 4.5: A node that has loops containing other addresses with sending back BTC to the original address. These loops do not contribute much information to the graph but increase the node count in the graph. For the question of finding BTC flow between two addresses, they do not add any information at all. By marking the central node as a loop-node, it gives the same information without rendering all nodes. In the client this results in a node stroked in a special color.

carefully, all occurrences of the motif can be collapsed into a super node. Even the smallest possible replacement of just two TX-nodes would result in one super node. Thus only half of the nodes of the source motif are left. If the subgraph occurs in larger numbers and the start as well as the end node are the same an even higher number of nodes can be replaced by one single super node.

Replacing substructures by collapsing them into one super node in combination with progressively streaming the data to the client helps to minimize the computational requirements for the client as well as provides the user with an interactive visualization without significant preprocessing delays. Mapping of nodes and edges to their respective rendered representation is similar to the Neo4j Browser shown in Figure 5.1. However, one big difference is that the size of the nodes is mapped to their degree. This can be seen in the client visualization shown in Figure 5.4. There is no distinction between

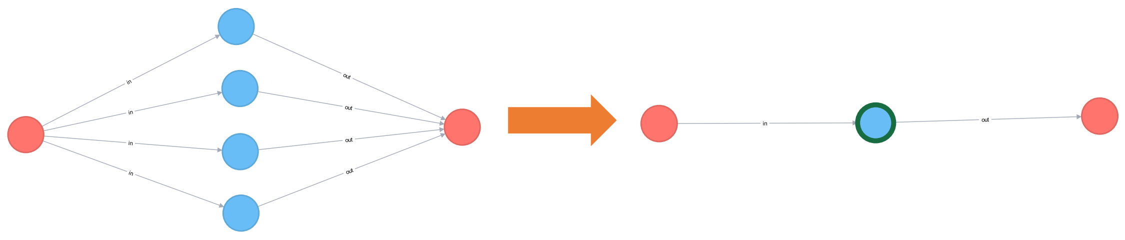


Figure 4.6: The fan-pattern between two nodes. One address sends BTCs to another one through several transactions. By collapsing this fan-structure into one artificial super node, the node count can be reduced.

outbound and inbound edges and just the sum of all edges per node is taken. Additionally, the thickness of the outbound edges is mapped to the amount of BTCs as saved in the outbound edges of a TX.

CHAPTER 5

Implementation and Algorithms

While Chapter 3 and 4 provided insight into the general methodology of this thesis, this chapter presents in detail, how the solution has been implemented. It provides a general overview of the used technology stack and highlights how some peculiarities of the used frameworks and tools have been dealt with.

In general, the work done in this thesis was realized in Python 3.6, web languages, such as HTML and Javascript, and Cypher. The main part of the project, namely the proxy server, has been implemented in Python and provides the client code via a web server. The client code is written in HTML and Javascript using D3.js as visualization framework. For the communication between the proxy server and the client a message structure for commands and data has been realized using JavaScript Object Notation (JSON).

5.1 Message Passing

The messages sent between the server and the client are JSON encoded messages. Both requests and responses are split into two parts: a header containing general meta information and a body that differs between message types.

The header of a request supplies general parameters to the proxy server. The examples in the following listings also contain fields that are not yet used by the proxy server. They are placeholders for possible future work that would enhance the functionality of the server. The *data* dictionary, *truncate* in the *query* dictionary and *coi* in the requests are currently not implemented and are reserved for future functionality. The idea of these parameters is to allow the client to send information about its memory capacity to the proxy server. This would allow for resource limitation guided progressive loading and processing. The header, for example, includes what kind of request it is and whether streaming should be performed. In total, there are four different message types (new, cancel, halt, and shutdown) that a client can send as a request to the proxy server.

Listings 5.1 to 5.4 show the JSON structure of each of the request types. The difference between the cancel and the halt request is, that after receiving a cancel request from the client, the proxy sends all data to the client currently being processed, whereas the halt message immediately terminates the streaming and the proxy does not forward the currently processed data. For new-message types there is also another subset of requests that corresponds to the path and neighborhood queries. These new-queries themselves have parameters necessary for the proxy to specify an initial user provided query.

```
============================================================
Cancel Request
Parameters:
  clear_cache: [true, false]
============================================================
{
  "general": {
    "type": "cancel",
    "streaming": false, //streaming is stopped
    "data": {
      "capacity": 5,
      "used": 2
    },
    "clear_cache": true //cache is cleared
  }
}
```

Listing 5.1: A simple cancel request that cancels a previously sent request and stops the streaming process. The client can also set a boolean "clear_cache" flag and signal the proxy whether the previously loaded data should be cleared or not. Data processing in the server is stopped. However, all data that is currently processed by the proxy will be still forwarded to the client.

```
============================================================
Halt Request
Parameters:
  clear_cache: [true, false]
============================================================
{
  "general": {
    "type": "halt",
    "streaming": false, //streaming is stopped
    "data": {
      "capacity": 5,
      "used": 2
    },
    "clear_cache": true //cache is cleared
  }
}
```

Listing 5.2: A simple halt request that cancels a previously sent request. The client can also set a boolean "clear_cache" flag and signal the proxy whether the previously loaded data should be cleared or not. When the proxy receives a halt-signal it immediately stops streaming of all the data to the client, even if there is still some data currently being processed.

```
=========================================================
Shutdown Request
Parameters:
  clear_cache: [true, false]
=========================================================
{
  "general": {
    "type": "shutdown",
    "streaming": false, //streaming is stopped
    "data": {
      "capacity": 5,
      "used": 2
    },
    "clear_cache": true
  }
}
```

Listing 5.3: A simple shutdown request not only cancels a previous request but fully shuts down the proxy server. This means that the progressive streaming connection is closed and even the webserver, for loading client code, is shut down. All processing and data loading in the proxy server is stopped and the client cannot connect to it anymore.

```
=========================================================
New Request
Parameters:
  streaming: [true, false]
  aggregation: optional
    degree_cut: integer
=========================================================
{
  "general": {
    "type": "new",
    "streaming": true, //streaming is stopped
    "data": {
      "capacity": 5,
      "used": 2
    },
    "aggregation": {
      "degree_cut": 1
    }
  },
  "query": {
   ...
  }
}
```

Listing 5.4: This listing shows part of a client request, sending a new query to the proxy. When a new query is sent, the client can also include an aggregation parameter. If the "aggregation" field is included the client *has* to specify "degree_cut" as a parameter. It defines the minimum degree of the nodes to return. Nodes with a degree less than the parameter will be pruned from the returned graph.

When the client sends a "new" request, it also has to specify a query. This query is a starting point for the streaming and the streaming strategy depends on which query the client sends. In order to support path queries and neighborhood expansion, there are two different queries.

The first query requests all paths between two nodes of the TX-graph. Although it is possible to query paths between transactions, to answer questions about who sent BTCs to whom, only address nodes are useful. Paths between transactions or blocks do not give this information and therefore, while technically possible, provide not an answer to the question of following the flow of BTCs through the TX-graph. Listing 5.5 shows the structure and the possible parameters of a query asking for paths. The listing shows part of a client request, sending a new path query to the proxy. When a new query is sent, the client can also include an aggregation parameter. If the "aggregation" field is included, the client *has* to specify "degree_cut" as a parameter. If aggregation is enabled, all graph minimization steps, including graph pruning, are performed. The path query supports multiple parameters to allow for finding distinct paths. "path_type" allows to specify which paths are searched for. Details on path finding are given in Section 5.5.2. The "label" fields for the start and end node specify which node type they are. The end node is implicitly given as the last node in the "hops" array. "rel" allows to specify the directionality of the path as well as the maximum distance. This is necessary for the performance as path search in graphs can be computationally costly.

```
==========================================================
New Path Request
Parameters:
  streaming: [true, false]
  aggregation: optional
    degree_cut: integer
  path_type: ["all_shortest", "shortest", "all"] // supply one of
     these parameters
  start:
    label: ["Address", "Transaction", "Block"] // supply one of
       these parameters
    params:
        hash: string //the hash of a node in the Bitcoin network
  hops:
    rel:
      direction: [-1, 0, 1] //directionality: -1 - (node)->(start)
         | 0 - undirected | 1 - (node)<-(start)
      dist: integer //maximum path length from start node
    node:
      label: ["Address", "Transaction", "Block"]
      params:
        hash: string // the hash of a node in the Bitcoin network
==========================================================
{
 "general": {
  "type": "new",
  "streaming": true,
  ...
  "aggregation": {
   "degree_cut": 1
  }
 },
 "query": {
  "type": "path",
  "truncate": 1000,
  "subgraph_labels": ["Address", "Transaction"],
  "path_type": "all_shortest",
  "start": {
   "label": "Address",
   "params": {
    "hash": "1A9zE4C215zA8HfigEdwxP8ApV8miEgudX"
   }
  },
  "hops": [{
   "rel": {
    "var": "r",
    "label": null,
    "direction": 1,
    "dist": [50]
   },
   "node": {
    "var": null,
    "label": "Address",
    "params": {
     "hash": "1JyEmxiMso2RsFVfBcCa616npBvGgxiBX"
    }
   }
  }],
  "coi": null
 }
}
```

Listing 5.5: This listing shows a client request of type "new". Such requests start the loading of data and graph minimization in the proxy server. The client can pass various parameters to the proxy server within this request.

Another interesting query to retrieve topological information about a node in a graph is the neighborhood query. It starts streaming the surrounding graph neighborhood of the initially specified node and gives the user information about how, for example, an address interacts with the Bitcoin network. Listing 5.6 shows the JSON structure of the query.

```
==========================================================
New Neighborhood Request
Parameters:
  streaming: [true, false]
  coi:
    label: ["Address", "Transaction", "Block"]
    hash: string //the hash of a node in the Bitcoin network
==========================================================
{
  "general": {
    "type": "new",
    "streaming": true,
    ...
  },
  "query": {
    "type": "surrounding",
    ...
    "coi": {
      "label": "Address",
      "hash": "13VgbQM6ssD9VnafHp3DVVidUzTaB9B49i"
    }
  }
}
```

Listing 5.6: This listing shows part of a client request, sending a new neighborhood query to the proxy. This query is much simpler than the path query as only one node needs to be specified. It acts as a center node from which the graph neighborhood is expanded. Although streaming can be set to *false* in this query, the usefulness of this query would be very limited. If the neighborhood query has streaming disabled only the directly connected neighbors of the start node will be returned. The resulting visualization would not give much information about the interaction of the starting node with the TX-graph.

Internally after the requests are received by the proxy server, they are transformed into Python objects. This allows for greater flexibility and makes it easier to retrieve parameters and use the query object for representing the internal progressive loading states. Another advantage is that, through the usage of Python enumerations for task types and query types, programming errors are minimized.

The other side of client-server communication are responses. As with requests, the communication is encoded as JSON and there are two different response types: status responses to send feedback about the received request to the client, shown in Listing 5.7, and data messages to send data to the client.

```
===========================================================
Status Response
===========================================================
{
  "header": {
    "message_type": "status",
    "status": "OK",
    "code": 3
  }
}
```

Listing 5.7: A simple status response from the server. In addition to the message type, a textual status representation and a status code are returned.

```
===========================================================
Data Response
===========================================================
{
  "header": {
    "message_type": "data",
    "hierarchy": false,
    "chunk_nr": 0,
    "node_types": ["block", "transaction", "address"]
  },
  "nodes": [{
    "id": 0,
    "hash": "1A9zE4C215zA8HfigEdwxP8ApV8miEgudX",
    "processed": false,
    "delete": false,
    "special_type": ["start"],
    "type": 2
  },
  ...],
  "links": [{
    "source": 0,
    "target": 1,
    "label": "in",
    "delete": false
  }, {
    "source": 1,
    "target": 2,
    "label": "out",
    "delete": false,
    "amount": 2.0,
    "type": "pubkey",
    "req_sigs": 1,
    "n": 0
  },
  ...],
  "paths": [...]
}
```

Listing 5.8: This message is comprised of four important parts, "header", "nodes", "links", and "paths". The header gives information about which data chunk is being received. The "node_types" field in the header and "type" in each node are necessary to resolve the node type in the "node" array. This is done in order to shorten the overall size of the JSON responses. *nodes* is an array of node objects that contains all the information saved in the blockchain. *links* is an array with the edges between the nodes. *source* and *target* refer to the *id* field of the nodes.

The JSON result shows all the fields contained in the blocks, TX and addresses. An abbreviated form of the response that includes data is shown in Listing 5.8. For the aforementioned special nodes as described in Chapter 4, the field *special_type* is used. Each of the nodes that partakes in the aggregation steps has its own label. Additionally, the start and end nodes of the path query also have special labels. *delete* is the field that signals the client that a node needs to be removed.

By referencing node types and nodes by ids, it is possible to reduce the size of the JSON response. Additionally, this is the format required by D3.js [71] to visualize node-link diagrams. Providing the client with a ready to visualize dataset, the processing requirements in the client are reduced. The client only has to take care of updating its internal data when new nodes and edges are streamed from the proxy. After this updating step, the visualization is updated and data marked for deletion are removed from the visualization. Finally new data are added to the visualization.

5.2 Complementary Work and Additional Queries

The complementary thesis by Alexandra Mai is concerned with on creating visualizations that focus on the temporal component of Bitcoin [72]. Her work focuses on past and current transactions and provides analysts with a forensic tool to visually explore the Bitcoin blockchain in-depth. Additionally her visualization client allows users to cluster and calculate the statistical probability TXs.

5.3 Database

Before any visualization can be done, the blockchain data has to be accessed. Although it is possible to load data through the Bitcoin client Application Program Interface (API), this is not the fastest method. For queries on graph data this would also result in complex queries and slow performance because operations, such as to look for paths or neighborhoods, are not supported in the Bitcoin API. Having the possibility to send queries like those to a data storage system that is made for graph data seems natural. Therefore, the blockchain data was loaded into a Neo4j [73] graph database in a preprocessing step. Neo4j is a database especially made for graph data and therefore graph theoretical queries can be performed without complicated query syntax or performance degradation. Using a Python script that iterates over the blockchain block by block and retrieves all its TXs and participating addresses, the data was inserted into the graph database. The script starts with the first block in the blockchain and creates a block-node in the graph database. For each TX in a block a TX-node is created. Subsequently, each address in a TX is loaded and added to the database. By linking input and output addresses of each TX with the TX-node in the database, the edges are introduced. Finally the edges between the TX-node and block are created. After processing a block is finished, the script continues with successive block and its TXs and addresses are processed.

Communication with the graph database happens through Cypher. It is a query language similar to the Structured Query Language (SQL). A major key difference, however, is that it is designed for querying graphs. Therefore, it supports graph pattern matching and graph queries, such as shortest paths, that regular relational database query languages do not support.

While Cypher is very flexible, some queries can be cumbersome to implement. Therefore, for database creation, neomodel [74] was used. Having an Object Graph Mapper (OGM) simplified this step as only the mapping from a Python object to database fields has to be specified. This is similar to Object-relational mapping for relational databases but instead is aimed at graph databases. Neomodel takes care of creating all the necessary Cypher queries in the background. Listing 5.9 shows the OGM structure used to create the database.

```
=========================================================
Object Graph Mapping for the Blockchain
=========================================================
from neomodel import (StructuredNode, StructuredRel, StringProperty,
    IntegerProperty,
   UniqueIdProperty, FloatProperty, RelationshipTo,
      RelationshipFrom, One, ZeroOrOne)


class Block(StructuredNode):
   version = IntegerProperty()
```

```python
    hash = StringProperty(unique_index=True)
    hash_next_block = StringProperty()
    tx_merkle_root = StringProperty()
    time = IntegerProperty()
    median_time = IntegerProperty()
    bits = StringProperty()
    nonce = StringProperty()
    confirmations = IntegerProperty()
    weight = IntegerProperty() # block size (after segwit)
    difficulty = IntegerProperty()
    chainwork = StringProperty() # number of total hashes/work up to
        this block (hex)
    size = IntegerProperty()
    stripped_size = IntegerProperty()
    height = IntegerProperty()

    # Relationships
    prev = RelationshipTo('Block', 'previous', cardinality=One)
    next = RelationshipTo('Block', 'next', cardinality=One)
    tx = RelationshipFrom('Transaction', 'hasTx')
    coinbase_tx = RelationshipFrom('Transaction', 'coinbase',
        cardinality=One)


class OutRel(StructuredRel):
    amount = FloatProperty()
    type = StringProperty()
    req_sigs = IntegerProperty()
    n = IntegerProperty()


class Transaction(StructuredNode):
    uid = StringProperty(unique_index=True)
    version = IntegerProperty()
    lock_time = IntegerProperty()
    size = IntegerProperty()
    vsize = IntegerProperty()
    hash = StringProperty()

    # Relationships
    block = RelationshipTo('Block', 'hasTx')
    coinbase_tx = RelationshipTo('Block', 'coinbase',
        cardinality=ZeroOrOne)
    in_address = RelationshipFrom('Address', 'in')
    out_address = RelationshipTo('Address', 'out', model=OutRel)


class Address(StructuredNode):
    hash = StringProperty(unique_index=True)

    # Relationships
    in_tx = RelationshipTo('Transaction', 'in')
    out_tx = RelationshipFrom('Transaction', 'out', model=OutRel)
```

Listing 5.9: Object Graph Mapper (OGM) used to create the Neo4j graph database. It shows the three different node classes and a special outgoing relationship class.

Even though Cypher is very powerful in expressing and querying graph structures, not all queries were easily possible. In order to drastically simplify time-related queries, the creation time of blockchain blocks has also been copied into the transaction nodes after the initial database creation. After the blockchain data has been loaded into the database, viewing it through the Neo4j Browser, a web interface for visually inspecting the database, results in a structure as seen in Figure 5.1.

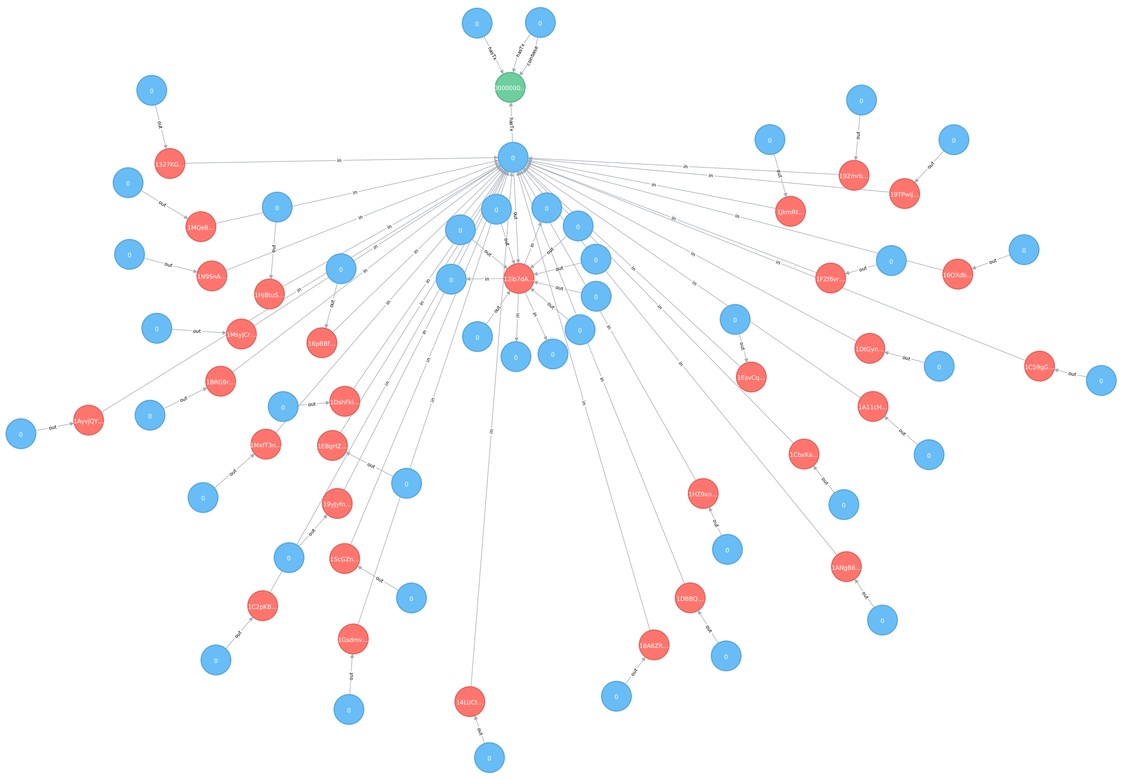


Figure 5.1: A small example of a block with transactions and its connected addresses. The green nodes show blocks, the blue nodes are transactions and the red nodes are addresses. All edges between the nodes are directed. It is a heavily pruned (77 of 262,560 nodes) and reduced example from the database. This is a screenshot taken from the Neo4j Browser with a different color scheme than the one used in the client visualization presented in this thesis. Due to changes in the Neo4j browser implementation, blocks and transactions have swapped colors in this figure.

5.4 Client

For increased flexibility and the least number of software requirements for the client, it is realized as a thin client using web technologies. The client code is based on HTML, CSS, and Javascript and uses only D3.js as framework for rendering. D3.js is a Javascript framework that helps to map data to the Document Object Model (DOM) automatically. The advantage of using web technologies lies in the fact that they are universally available. Only a web browser is necessary to download and run the client code. However, this advantage comes with the downside that, generally, web implementations do not perform as well as native implementations. Nonetheless, for the goals in this thesis this should not pose a problem. The data that reaches the client should be reduced through the processing steps in the proxy server, so that reduced processing capacities should not matter. Apart from D3.js, which is an external library, everything else has been implemented in standard web technologies that work with every browser.

When a user navigates her web browser to the website of the proxy server, the browser downloads all the necessary files for the client to run. This includes the website, the style information, the visualization and rendering code in Javascript, and code to open a websocket connection to the proxy server. After the browser has established this websocket link, the user can send queries to the proxy server through the user interface. When data is returned through the websocket, the received JSON is parsed into Javascript objects and the visualization update functionality is called.

The visualization is rendered as Scalable Vector Graphics (SVG) through D3.js. Layouting is also done through the forceSimulation functionality of the D3.js API. There are two different layout mechanisms implemented for the node-link diagram. One is the traditional force-directed layout that is comprised of attracting and repelling forces. The second layout algorithm changes the node positioning while attracting forces of the edges remain intact. Nodes are positioned in the visualization according to the times of the blocks and the TXs along the x-axis. Besides switching the layout algorithm, there is no user interaction implemented.

To showcase the pruning and aggregation as well as the progressive loading capabilities of the server, coloring of special nodes is employed in the client. Blocks are blue, TX are green and addresses are red. Nodes can have various states and nodes that are yet to have their neighborhood expanded use the light version of these colors. They can be nodes that are already fully expanded, aggregated or have pruned neighboring nodes. Fully expanded nodes and not yet expanded nodes share the same node color but their saturation is different. The proxy server can mark certain nodes as special nodes and they will be marked by different visual properties. For special nodes a distinction between hairball aggregation and motif aggregation is made. Start nodes of either the path or the neighborhood query are marked with a colored stroke as well. The path query also returns the final node of the paths as a special node with its own stroke color. Additionally, special nodes have different stroke styles. Figure 5.2, 5.3 and 5.4 show the markings and different node colors and sizes as visualized in the client.

Figure 5.2 shows the visualization as implemented by the client and the control panel to the right. In this Figure all possible node types are visible. Although the graph in the proxy server contains thousands of nodes and edges, the client node-link diagram is small enough to allow for an interactive visualization.

Figure 5.3 shows a more detailed view of the client graph, including the motif and hairball aggregation as well as different states of the neighborhood expansion. The node with a purple stroke is the start node and the node stroked in pink is the target of the path query. Nodes with an orange circle are hairball nodes. Light-green nodes of large size are fan-motif super nodes the orange nodes mark "loop"-motifs. Dark colored nodes are fully expanded in the proxy graph and light colored nodes do not have their neighborhood loaded yet.

Figure 5.4 shows the same graph and focuses more on the two red nodes with large diameters. With the exception of motif super nodes, the node diameter increases with the degree of the graph vertex. The two red nodes show this clearly as they have many incoming and outgoing edges. Figure 5.5 shows the same graph with the second layout option. The TXs are placed along the x-axis according to their respective timestamps. This ordering gives information on how BTCs have been sent over time from a start to an end address.

5.5 Server

The core part of the system proposed in this thesis is the proxy server. It combines almost all processing in itself and therefore most of the work is performed there. The main implementation of it is done using Python 3.6 and it uses several frameworks for client communication, graph storage, and processing. This has the advantage that graph processing, implemented through graph-tool [75], has algorithms written in C++ and, therefore, improved speed over Python. Python is a high level language that is compiled to byte-code and not to machine-code. Therefore, the performance of a pure Python implementation is not on par with a native machine-code implementation such as C++.

Graph-tool implemented by Tiago P. Peixoto [75] is a library for graph analysis written in C++, with a heavy focus on being used through Python. The combination of speed with the ease of use through Python was the main reason for choosing graph-tool over a completely native Python implementation such as NetworkX [76]. Graph-tool provides the necessary algorithms for path finding, motif search, and data storage for the implementation of the methodology presented in this thesis. For providing the client code through a webserver and client communication in the form of websockets, the implementation uses Twisted [77] and Autobahn [78]. These two projects are closely related but provide different functionality. While Twisted provides an easy to use web server to support static content, Autobahn makes it easy to open and handle websocket connections. For communicating with the database, the same framework as for the database initialization is used. Neomodel with its convenience OGM functions uses the neo4j-python-driver [79], provided by the Neo4j project, in the background. However, for querying the database, no OGM is used but rather raw Cypher queries are sent. This allows for more complex queries with results that have been preformatted for easy processing in Python.

When the proxy server is started, it initializes a controller, which creates all necessary synchronization objects (multiprocessing.Queue, multiprocessing.Event) and spawns two more processes. It passes the necessary data pipes and synchronization event objects to the newly spawned processes. These two processes are the server component that acts as interface to the client, and the data processing module that is responsible for all the data storage, data processing, and database communication. Once everything is set-up, the proxy waits for a client connection and a subsequent query to start processing data.

When a client connects and sends a query to the proxy, the request passes through all three components. For data requests, the data module translates the client query into a database query and sends it to the Neo4j database. Once data is returned, it is added to the proxy server's in-memory graph and the graph is processed in different stages. Before the graph is sent to the client, it needs to be translated into nodes and links and embedded into a JSON message. To minimize network requirements, only new or updated nodes are sent to the client. Previously sent nodes and edges that have not undergone a change since they were sent are skipped. Generally, the graph data passes the following stages:

1. Remove blocks from the graph as they do not necessarily give additional information. This is an optional step and mostly makes sense for path queries where only transaction patterns are of interest.

2. Remove disconnected components. If removing blocks has resulted in creating multiple connected components, the components, where the nodes of the initial query are not part of, are removed.

3. Detect hairballs and prune them and mark the aggregated nodes as special nodes.

4. Find and replace motifs. This step searches for previously mentioned fan- and loop-motifs.

5. Remove all nodes and edges that have been previously sent to reduce the data to translate into JSON and send them over the network.

The next sections detail the operations performed during this sending preparation step. This graph processing happens every time new graph data has been added to the in-memory graph through constantly querying the database during neighborhood expansion and database path querying. Additionally, the graph is constantly searched for new paths, as described in Section 5.5.2, and if one has been found, this processing cascade is also started before the path is sent to the client.

5.5.1 Neighborhood Expansion

Neighborhood expansion takes nodes of found paths and loads their neighborhoods. This helps to show how paths between nodes are embedded into the blockchain. If a Bitcoin user has reused an address, this also reveals other related addresses. Additionally, neighborhood expansion is a query that terminates quickly and, therefore, continuously sends data to the client. Visualization updates in the client help to give feedback to the user that data streaming is still being performed by updating the client. This hides long latencies of queries with longer computation time. Finding paths in a graph can take a long time and users can perceive that as unresponsive. If a neighborhood query is started, neighborhood expansion is the main processing algorithm to load new data. For a path query, additionally loading more data in this manner stems from the assumption that nodes that are topologically close to existing paths might occur in more paths between two nodes of interest. Users transacting with each other may possibly do so more than once and therefore their addresses and transactions will produce multiple paths with overlapping address nodes in the graph.

The general algorithm works as follows. Starting with an initial set of vertices, split them into s subsets of size k, spawn s threads and let each of the threads load the neighborhood of each node in the respective subset. After all threads have returned, the results are merged into a single result set and the new nodes are added to the processing queue. The new nodes and edges are added to the graph and graph processing (aggregation, path search, etc.) is performed. Finally, the new data is sent to the client. As long as no termination signal has been received or the node list, of nodes to be processed, is empty, the procedure is repeated. Algorithm 5.10 shows the complete procedure including queue splitting, thread spawning, and merging of results.

```
==========================================================
Parallel and Sequential Neighborhood Expansion
==========================================================

def neighbourhood_worker(thread_idx, result, node_ids, distance,
    iterative):
    tmp_data = self.db.get_multi_neighbourhood(node_ids, distance,
        iterative)
    result[thread_idx] = tmp_data

if self.neighbourhood_threading:

    tmp_results = [None] * self.nr_neighbourhood_threads
    thread_chunk_size = math.ceil(len(next_query) /
        self.nr_neighbourhood_threads)

    # Start threads with their workload share
    for i in range(self.nr_neighbourhood_threads):
        thread_chunk_data = set(
           itertools.islice(next_query, i * thread_chunk_size, (i + 1) * thread_chunk_size))

        self.neighbourhood_threads[i] = threading.Thread(target=neighbourhood_worker,
                                             args=(
                                             i, tmp_results, thread_chunk_data, [1], False))
        self.neighbourhood_threads[i].start()

        # Join them back together
        for i in range(self.nr_neighbourhood_threads):
           self.neighbourhood_threads[i].join()

        # Merge results together
        graph_data = ([[[], []]], tmp_results[0][1]) # empty container - layout as from db
        for i in range(self.nr_neighbourhood_threads):
           graph_data[0][0][0] += tmp_results[i][0][0][0]
           graph_data[0][0][1] += tmp_results[i][0][0][1]

           self.update_processed_nodes(next_query) # mark expanded nodes

else: # Perform db query sequentially
    graph_data = self.db.get_multi_neighbourhood(next_query, [1], True)
    self.update_processed_nodes(next_query) # mark expanded nodes
```

Listing 5.10: Python code showing two neighborhood expansion strategies. Either querying the database with one big query or splitting the nodes to query into chunks, according to a number of assigned threads, can be chosen as strategy. If the parallel strategy is to be performed, then all queries are started in parallel and after data is returned from the database it is assembled and processed further. Overlaps are processed when the data is added to the in-memory graph of the proxy server.

5.5.2 Path Finding

Path finding is only done if the user starts a path query and works on both the in-memory graph data as well as the whole blockchain data. Both processing methodologies are very similar. The only difference is that one uses graph-tool path finding algorithms [69] while the other sends path queries to the database. Due to a slight difference in functionality, the way previously found paths are handled is different.

After spawning two threads, one for each type of data source, the first step for the database path-finding algorithm is to retrieve a list of previously found paths, used as a blacklist, and create a query to send to the database. Using such a blacklist ensures that no previously found path is returned from the database. Once a new path is found and received it is added to the list of found paths and the data is added to the graph. The nodes in the path are also added to the neighborhood expansion queue as the first nodes to process next. This is done to prioritize newly found nodes and to load the surrounding neighborhood of the new path. In the next step the new data is added to the in-memory graph of the proxy server. Then, if the query enabled aggregation, the aggregation steps are performed, the graph is transformed into JSON, and the data update is sent to the client.

The in-memory path finder works slightly differently. It iterates over all paths in the graph, found by a breadth-first search, and for each new path that is not already known, it adds the path to the already known paths. Then, for all nodes in the path that have not yet been neighborhood-expanded, add them to the beginning of the node list. Then, start the graph minimization procedure and send the current graph data to the client. Algorithm 5.11 shows the code of the database path search and Algorithm 5.12 shows in-memory path search. The path finding is performed by graph-tool and does a fresh breadth-first search every time it is called.

```
=========================================================
Database Path Search
=========================================================

def db_path_finder(self, query):

  while self.stream_ctrl.is_set() and self.do_progressive_loading:

     # Profile
     measurement_cache = []

     # Get copy of currently known paths (threading)
     tmp_curr_paths = self.gm.paths.copy()

     # Apply path length multiplier
     for path in query.path:
        path['rel']['dist'][-1] *= self.db_path_length_multplier

     # Query database for a new path that has not been found yet
     result_data = self.db.get_all_paths_id(query.start, query.path,
                                 [[self.gm.graph_id_to_db_id(v) for v
                                    in path]
                                    for path in tmp_curr_paths],
                                 self.path_timeframe[0],
                                 self.path_timeframe[1])

     # Split graph data and path data
     path_data = result_data[0][0].pop(-1)

```

```python
            # Check if there was a path found
            if len(path_data):

                # Update graph
                self.synchronized_graph_update(result_data)

                # Handle process path data
                # Update found paths and next nodes to process (queue first)
                tmp_graph = self.get_graph_snapshot(self.gm.graph)
                tmp_path_nodes = set() # set -> automatically avoid duplicates in path
                paths = []

                for path in path_data:
                    [tmp_path_nodes.add(node) for node in path if node not in self.processed_nodes]

                    # Process new data to generate JSON
                    p_tuple = tuple(self.gm.db_id_to_graph_id(id, tmp_graph) for id in path)
                    v_list, e_list = self.gm.add_path(p_tuple, tmp_graph)

                    # Clean already sent vertices, edges
                    v_list = [vertex for vertex in v_list if self.gm.prop_map.nodes.unsent[vertex]]
                    e_list = [edge for edge in e_list if self.gm.prop_map.edges.unsent[edge]]

                    paths.append([{p_tuple: False}, v_list, e_list])

                # If there are new nodes -> update node processing queue
                if len(tmp_path_nodes) > 0:
                    self.update_node_queue(list(tmp_path_nodes), True)

                data = self.gm.path_list_to_json(self.nr, paths)

                if self.stream_ctrl.is_set():
                    self.data_update_signaling(data)

            else:
                self.db_path_length_multplier *= 1.5 # Expand path search radius if no paths are found

        self.background_db_path_thread = None # To signal termination
```

Listing 5.11: Python code showing the database path search. If no path is found for a specific length, the maximum path length is increased. This helps to terminate the query in a feasible amount of time. Once a path is found, it is added to the in-memory graph and the list of known paths is updated. Finally, an update is sent to the client.

```
=========================================================
In-memory Path Search
=========================================================

def graph_path_finder(self, query):

  while self.stream_ctrl.is_set():

     # Find all paths between nodes in current graph
     new_paths = self.gm.path_in_graph(
                  self.gm.find_by_hash(
                      query.start['params']['hash'])[0],
                  self.gm.find_by_hash(
                      query.path[-1]['node']['params']['hash'])[0],
                  self.nr,
                  self.get_graph_snapshot(self.gm.graph),
                  p_type=query.path_type)

     # If there are paths -> process ELSE sleep...
     if len(new_paths) > 0: # update the nodes to process queue and add path nodes first IFF not processed yet

        # Get nodes in paths that have not been processed
        tmp_path_nodes = set()

        [[tmp_path_nodes.add(self.gm.prop_map.nodes.id[node]) for node in path if
          node not in self.processed_nodes]
             for
             path in new_paths]

        # If there are unprocessed path nodes -> update node queue (prepending)
        if len(tmp_path_nodes) > 0:
           self.update_node_queue(list(tmp_path_nodes), True)

        # Send path data to client
        try: # Check if aggregation is to be performed
           data = self.graph_to_json(self.nr,
                               self.gm.prune_low_degree(
                                   self.get_graph_snapshot(
                                       self.gm.graph),
                                   self.current_task.aggregation.degree_limit))
        except AttributeError: # No aggregation
          data = self.graph_to_json(self.nr,
                              self.gm.get_unsent_graph(
                                  self.get_graph_snapshot(self.gm.graph)))

```

```
            # Send to client
            if self.stream_ctrl.is_set():
               self.data_update_signaling(data)


         else: # ...else pause thread for half a second if no new paths
            are found to save processing capacities
            time.sleep(0.5)

      self.background_graph_path_thread = None
```

Listing 5.12: Python code showing the in-memory path search. It is a simple algorithm constantly searching for new paths. If no new paths are found, it pauses for a moment to save CPU resources. If a path is found it is added to the list of known paths and the path is sent to the client with the updated nodes and edges.

5.5.3 Pruning

Apart from finding new paths, graph minimization, to overcome performance and overplotting problems, is an important part of the proposed methodology. Pruning is part of minimizing the full blockchain data into a smaller dataset. Two methodologies are applied in this thesis. One is to simply prune nodes below a certain node degree threshold and the other one is to remove nodes with high degree and merge the surrounding nodes into the central node. This is in contrast to other work, as presented in Chapter 2 that reduces data size by aggregating across data dimensions or prunes through temporal or spatial limitations.

Degree Pruning

Thresholding nodes by their degree involves finding all nodes n where $deg(n) < t$ for a threshold t and removing them from the graph. However, after removing some nodes the graph might become disconnected and the component including the nodes of the initial user-provided query needs to be kept for further processing. The other components are removed from further graph processing. The parameter t is set to one by default. This means that the graph can not be split into multiple connected components. Only if the user specifies a different parameter multiple connected components can occur. All paths inside a path have at least degree one and will not be removed. Therefore, no necessary information is lost. Algorithm 5.13 shows the simple steps involved to prune the graph.

```
===========================================================
Degree Pruning
===========================================================

def prune_low_degree(self, graph, degree=1):

   # Get only tx<->address subgraph
```

```
    graph = self.get_tx_subgraph(graph)

    # Filter graph by degree
    graph_degrees = graph.degree_property_map('total') # total =
        in-degree + out-degree
    filter_degree = graph.new_vertex_property('bool', val=0)
    map_property_values(graph_degrees, filter_degree, lambda x: True
        if x > degree else False)

    graph = GraphView(graph, vfilt=filter_degree)

    # Filter small connected components
    graph = self.filter_components(graph)

    # Filter already sent
    #graph = self.get_unsent_graph(graph)
    return graph
```

Listing 5.13: Python code showing pruning of all nodes below a certain degree.

5.5.4 Hairball Aggregation

Hairball pruning is similar to degree pruning but it needs a few extra steps. First, nodes with a degree higher than a threshold t are selected. This is a threshold that is supplied by the user and set to 5 by default and during the testing in Chapter 6. Then, for each of the nodes all the degrees of all neighbors are inspected. Count all neighbors with degree 1 and if the central node has more than five such neighbors, it is marked as hairball. All neighboring nodes with $deg(n) = 1$ and their connecting edges are removed from the graph. Also check for previously marked hair balls if they are still considered hairballs, if not remove the special label. Finally, once again select the correct component in the graph and remove disconnected components. The whole algorithm can be seen in Listing 5.14.

```
==========================================================
Hairball Aggregation
==========================================================

def prune_hair_balls(self, graph, degree=10):
    min_neighbourhood_size = 5
    max_degree_range = 9999999999999999999

    filter_hairball = graph.new_vertex_property('bool', val=1)

    # Find all vertices with high degree
    vertices = find_vertex_range(graph, 'total', [degree,
        max_degree_range])

```

```
tmp_special = {}

# For all vertices
for v in vertices:

   # Counter for deg(n) == 1 neighbours
   hairball = 0
   h_neighbours = []
   # Check if should be marked as hairball
   #hairball = False

   # Check if the neighbours...
   for n in v.all_neighbours():
      # Have a degree of 1
      if n.out_degree() + n.in_degree() < 2:
         hairball += 1
         h_neighbours.append(n)

      else: # have higher degree...
         # ...and were previously marked as hairball vertex
         if self.prop_map.nodes.delete[n]: # Change delete flag
             && mark for resend
            self.prop_map.nodes.unsent[n] = True
            self.prop_map.nodes.delete[n] = False

            # also change the edge (v,n) to be not deleted
            if graph.edge(v, n): # Change delete flag && mark for
                resend
               if self.prop_map.edges.delete[(v, n)]:
                  self.prop_map.edges.unsent[(v, n)] = True
                  self.prop_map.edges.delete[(v, n)] = False
            else:
               if self.prop_map.edges.delete[(n, v)]:
                  self.prop_map.edges.unsent[(n, v)] = True
                  self.prop_map.edges.delete[(n, v)] = False

   # If the node qualified as hairball
   if hairball >= min_neighbourhood_size:
      # For all neighbours
      for n in h_neighbours:
         # Remove them from the graph and set delete flag
         filter_hairball[n] = False
         self.prop_map.nodes.delete[n] = True

         # Marke edges to delete
         if not graph.edge(v, n): # check directionality
            self.prop_map.edges.delete[graph.edge(n, v)] = True
         else:
            self.prop_map.edges.delete[graph.edge(v, n)] = True

      tmp_special[v] = 'hair'

   else: # is not a hairball in current iteration
      if v in self.v_special: # check if vertex was hairball previously
         if 'hair' in self.v_special[v]: # has hair label
            if len(self.v_special[v]) > 1: # has more labels
               self.v_special[v].remove('hair') # remove hair label
            else: # only hair label -> else delete from special nodes
               del self.v_special[v] # check that this does not delete the vertex from graph

# Update special nodes
if len(tmp_special) > 0:
   self.add_specials_by_v(tmp_special)

   graph = GraphView(graph, vfilt=filter_hairball)

   # Filter small connected components
   graph = self.filter_components(graph)

return graph
```

Listing 5.14: Code for pruning all nodes in the graph that have more than 5 neighbors with a degree of 1 and marking them as a special node. This code also checks if nodes have previously been removed and need to be added to the client visualization again. If so, the proxy server marks the nodes to be updated and sent to the client. This needs to be done to correctly update the client graph when new data from the database is loaded.

5.5.5 Motif Search

Due to some limitations of graph-tool, motif search is by far the most complex operation. Even though graph-tool is implemented in C++, motif search remains a computationally very intensive operation. With increased motif size, the computation also increases and therefore only small but recurring motifs are processed. Once a motif is found, graph-tool returns the location of the motif but not its orientation. For a rotational invariant motif, such as a loop, this does not pose a problem. However, for fan-motifs where there is a specified source and a target, this invariance is not given. Graph-tool seems to order the resulting vertices according to their vertex index in the in-memory graph and not according to the way the search pattern has been specified during the motif search function call. For rotation invariant motifs such as the self-loop no additional steps are necessary. For the fan-motif additional steps to find its orientation are necessary before it is possible to replace motifs with super nodes in the graph.

Motif aggregation is performed through multiple, intermediary steps. These steps include locating the motif, computing the motif orientation for fan-motifs, sorting motifs according to their source node, filtering motifs that would destroy topology if aggregated and finally, the motifs in the graph are removed. The first step is to find the motifs in the graph. Then, a map containing the common start node for all motifs is created. Once this map has been computed it is possible to find overlapping motifs and replace more than one motif with only one super node. For the rotationally sensitive fan-motif, the additional step of orienting the motif in the graph is performed. The source node is identified by finding the node that has two outgoing edges to two other nodes from the motif as target. After this node is identified, one neighboring node from the source is selected and once again, the node, in the set of nodes of the currently processed motif, that has an outgoing edge from the intermediary node is selected. This gives the motif target node. Figure 5.6 shows this orientation process in detail. In this example vertex 4 is marked as starting point. Then by traversing the center node, with index 1, the end vertex 2 is found. After the source and target vertices have been identified, the orientation of the motif in the graph is known. After this step, the processing is the same for the loop- and the fan-motifs. Once the motifs are oriented and sorted according to source nodes, the motifs are filtered and only motifs that are not part of bigger graph structures are kept to aggregate. Motifs where the central nodes (marked in green in Figure 5.6) of the motif have to have $deg(n) == 2$. Otherwise, they would have edges to other nodes not part of the motif. Aggregation of these motifs would destroy topological information. After filtering the motifs, they are merged into a bigger structure for easy removal. The last step involves removing the motif nodes in the graph but differs for both motifs. For fan-motifs, the removed nodes are replaced by introducing a super node with two edges between the source and the target nodes of the motifs. For loop-motifs all vertices except the source node are removed from the graph. Source nodes of loop-motifs are marked as special nodes and are marked for updating in the client. After the client received the updated loop-motif nodes, it visually marks them by highlighting them an orange color. Algorithm 5.15 shows the code of all the necessary steps for the motif aggregation. Currently there is no distinction between addresses and transactions as start nodes of the motifs. This can be updated in the future to distinguish between the two different cases. Both cases differ in what they mean semantically. If the central nodes of a fan-motif are TX, then it means that one address sends BTCs to another address through multiple transactions. If the central nodes are addresses, however, it means that the same addresses are used to transact money in two distinctive TX. By treating these two cases differently more semantic information can be conveyed to the user.

```
=========================================================
Motif Aggregation
=========================================================

def generate_motif_graph(self, graph):
   # generate motifs
   query_motifs = self.generate_motifs()
   # find motifs
   motifs = self.find_motifs(graph, query_motifs) # find occurrences of the query_motifs in the graph
   self.filter_motifs(graph, motifs) # filter motifs that are connected to other nodes not contained in any motifs
   merge_map = self.merge_motifs(motifs) # merge fan-motifs with the same source and target nodes together
   return self.process_motif_graph(graph, motifs, merge_map) # remove loop-motifs and replace fan-motifs with super nodes

def filter_motifs(self, graph, motifs):

   if len(motifs[1]):
      # filter all motifs with central nodes deg(n) > 2
      deg_map = graph.degree_property_map('total')
      del_marker = []
      for v_start, m_list in motifs[1].items():
         for m in m_list:
            # if either of the central nodes has connections to other nodes
            if deg_map[graph.vertex(m[2])] > 2 or deg_map[graph.vertex(m[3])] > 2:
               motifs[1][v_start].remove(m)  # remove motif from the motif list

         if len(m_list) == 0: # if no motifs are left for a certain source node remove the start node from the list
            del_marker.append(v_start)

      for i in del_marker:
         del motifs[1][i]

def merge_motifs(self, motifs):
   merge_map = {}
   # for all start nodes
   for v_start, m_list in motifs[1].items():
      # compare all motifs with each other and check for overlaps
      for i in range(len(m_list)):
         for j in range(i + 1, len(m_list)):
            indices = [i, j]
            if m_list[i][2] == m_list[j][2]:
               indices += [2, 2]
            if m_list[i][2] == m_list[j][3]:
               indices += [2, 3]
            if m_list[i][3] == m_list[j][2]:
               indices += [3, 2]
            if m_list[i][3] == m_list[j][3]:
               indices += [3, 3]

            if len(indices) == 4: # the two compared motifs overlap on at least one node
               if v_start in merge_map:
                  merge_map[v_start].append(indices)
               else:
                  merge_map[v_start] = [indices]
   return merge_map

def process_motif_graph(self, graph, motifs, motif_map):
   v_virt = graph.new_vertex_property('bool', val=False)
   e_virt = graph.new_edge_property('bool', val=False)

   v_m_filt = graph.new_vertex_property('bool', val=True)
   e_m_filt = graph.new_edge_property('bool', val=True)

   # for all nodes with motifs
   for v_start, motifs in motifs[1].items():
      for m in motifs: # for all motifs of node
         v_m_filt[graph.vertex(m[2])] = False # set vertices in filter to be removed from the graph
         v_m_filt[graph.vertex(m[3])] = False

         e_m_filt[graph.edge(v_start, m[2])] = False # set edges in filter
         e_m_filt[graph.edge(v_start, m[3])] = False
         e_m_filt[graph.edge(m[2], m[1])] = False
         e_m_filt[graph.edge(m[3], m[1])] = False

      # Add node and edges instead of motif
      v_m_virt = graph.add_vertex()
      v_virt[v_m_virt] = True
      v_m_filt[v_m_virt] = True
      e_m_virt = graph.add_edge(v_start, v_m_virt)
      e_virt[e_m_virt] = True
      e_m_filt[e_m_virt] = True
      e_m_virt = graph.add_edge(v_m_virt, m[1])
      e_virt[e_m_virt] = True
      e_m_filt[e_m_virt] = True

   # generate graph with the motifs filtered out
   g_agg = self.get_graph_snapshot(graph)
   g_agg.set_filters(e_m_filt, v_m_filt)

   # generate graph with only the super nodes
   g_virt = GraphView(graph, vfilt=v_virt, efilt=e_virt)

   # get all virtual vertices...
   v_tmp = {}
   for v in g_virt.vertices():
      if v_virt[v]:
         v_tmp[v] = 'virtual'

   # ...and mark as special nodes
   self.add_specials_by_v(v_tmp)

   # return the graph with the replaced nodes
   return g_agg

def find_motifs(self, graph, q_motifs):

   def sort_motifs(motifs):
      # Sort motifs according to their source node
      l_map = {}
      for motif in motifs:
         if motif[0] in l_map:
            l_map[motif[0]].append(motif)
         else:
            l_map[motif[0]] = [motif]

      return l_map

   def orient_motifs(motifs):

      m_oriented = []

      # For all motifs...
      for m in motifs:
         # For each vertex in motif...
         for v in m:
            # Get neighbours
            n_out = graph.get_out_neighbours(v)
            # Check if source node (intersection of motif and
                neighbours == 2)
            s_n_out = set(n_out)
            s_motif = set(m)
            v_intersect = s_n_out.intersection(s_motif)
            if len(v_intersect) == 2:
               # Found source...
               # Remove source && central vertices from vertex set
                   to get target
               v_tgt = s_motif - s_n_out - {v}

               # Construct final array to store motif with
                   orientation
               m_oriented.append([v] + list(v_tgt) +
                   list(v_intersect))
               break

```

```
        return m_oriented

    def format_motifs(motifs):
        m_list = []
        for m in motifs:
            m_list.append(list(m))

        return m_list


    # Iterate over query motifs and locate them in the graph
    res_motifs = []
    for i in range(len(q_motifs)):
        res_motifs.append(motifs(graph, k=q_motifs[i].num_vertices(),
            motif_list=[q_motifs[i]], return_maps=True))

    if len(res_motifs[0][2]):
        res_motifs[0] = format_motifs(res_motifs[0][2][0])
        res_motifs[0] = sort_motifs(res_motifs[0])
    else:
        res_motifs[0] = {}
    if len(res_motifs[1][2]):
        res_motifs[1] = orient_motifs(res_motifs[1][2][0])
        res_motifs[1] = sort_motifs(res_motifs[1])
    else:
        res_motifs[1] = {}

    return res_motifs
```

Listing 5.15: This listing shows the code necessary for finding and replacing fan- and loop-motifs in the graph. First the motifs need to be located in the graph. Then they are filtered and if they are embedded in larger graph structures, which means that they are connected to nodes not part of a motif, they can not be removed from the graph. Next, they are merged together if there are overlaps between motifs of the same type. Finally, all the nodes of the remaining motifs are removed from the graph. Nodes that were start nodes of loop-motifs are marked as special nodes. For fan-motifs, the motif is removed and replaced by a super node in the graph. Additionally for each iteration the algorithm checks if the motifs are still valid and if not reintroduces the nodes into the graph and sends the update to the client. This is necessary to account for updates when more data is returned from the graph database during streaming.

5.6 Data Sending

Sending data to the client involves passing the current in-memory graph to all necessary processing steps. In each step the number of vertices is reduced until finally a minimized graph remains. This graph contains all special nodes and aggregations and can be translated into JSON. During transformation of the graph data, only nodes that need updating, deletion, or have not yet been sent to the client are included in the final JSON string.

5.6.1 Lost in Translation

During the implementation of this thesis, several avenues were explored and discarded. Generally the approaches did not perform in a way that concurred with the speed goals, of hiding processing latency from the user, that were initially set. This is just a small summary of less successful techniques. Most complications stemmed from the highly parallel nature of processing. Operating on a graph while adding new elements and creating JSON for updating the client required more synchronization than initially anticipated. A simple approach of just adding elements whenever new data is returned from the database does not work because race conditions can occur or the program crashes because values changed that should not have. Iterating over the graph elements for JSON creation takes a long time and during this operation it is not possible to insert new data.

The initial idea was to make all graph operations, such as inserting new data, writing JSON, and some processing steps (pruning and aggregation), blocking. This resulted in a very bad performance where after around 10 minutes only a few thousand nodes have been loaded into memory. Instead of crashing the server or having many parts of the program that are blocking, the idea was to create a second copy of the graph. However, due to the implementation of graph-tool as a C++ library and the reliance on pointers for certain search operations, synchronization between the two graph versions was overly complicated. Additionally, copying large graphs, especially when done regularly, takes a toll on performance as well.

Finally, instead of copying graphs or large blocking code chunks, using graph snapshots proved to be the fastest and least complicated way of handling the system's parallelism. Graph-tool provides *GraphViews* that are similar to database views. This means it is easy to freeze the state of a graph and operate on it. This is used for all the pruning and aggregation steps as well as converting the data into JSON for the client. *GraphViews* share the same data in memory but do not change if new data is added. Therefore, iterating over nodes and edges does not cause the Python runtime to crash. Creating *GraphViews* is fast and does not require a lot of additional memory. Only during insertion of new data, blocking access is necessary. However, Section 7.2 presents a possible solution to this problem that is reserved for future work.

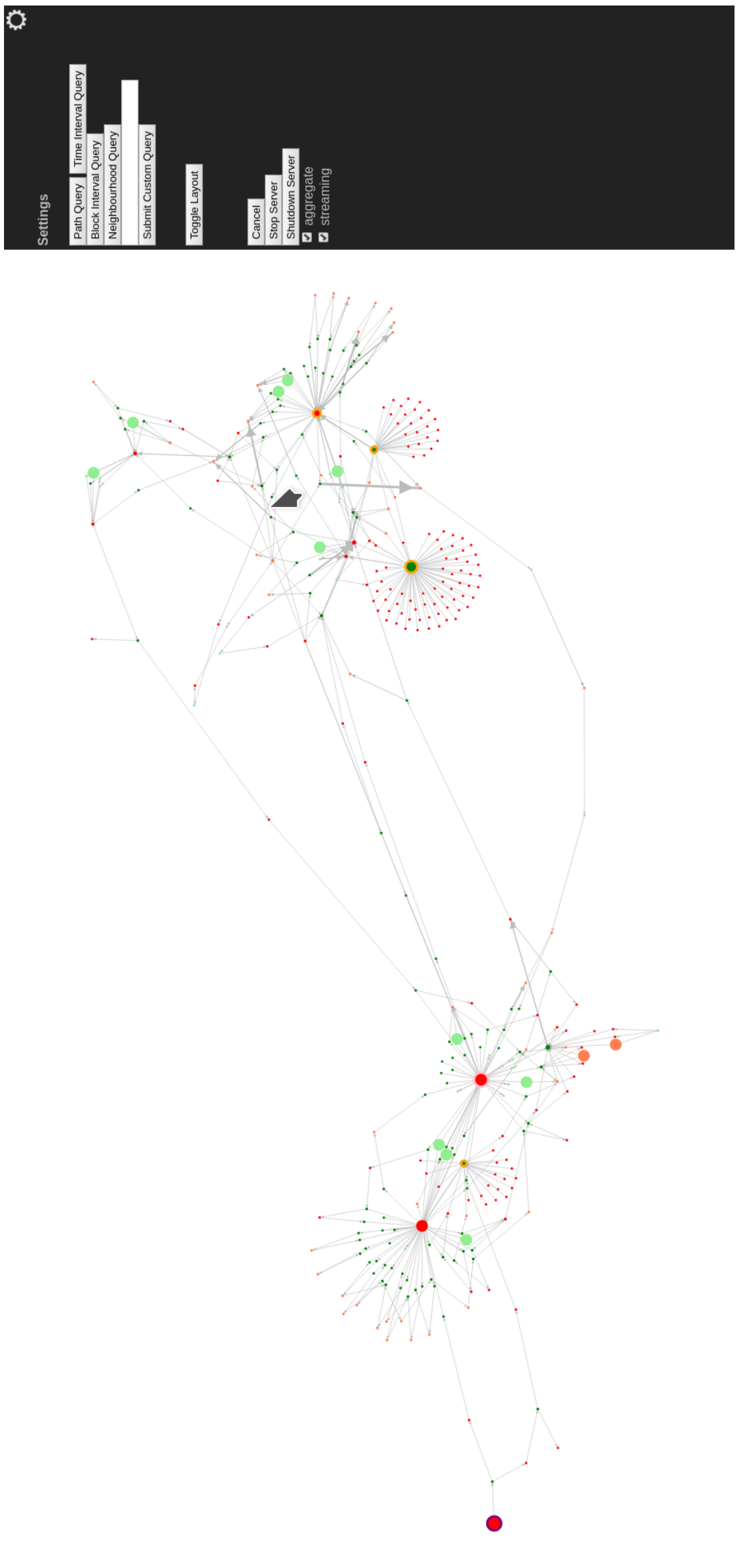


Figure 5.2: A screenshot of the client implementation. It shows transactions and addresses in green and red respectively. Less saturated versions of the colors signify that the nodes have not yet been expanded. The right hand panel shows the interaction possibilities for the user.

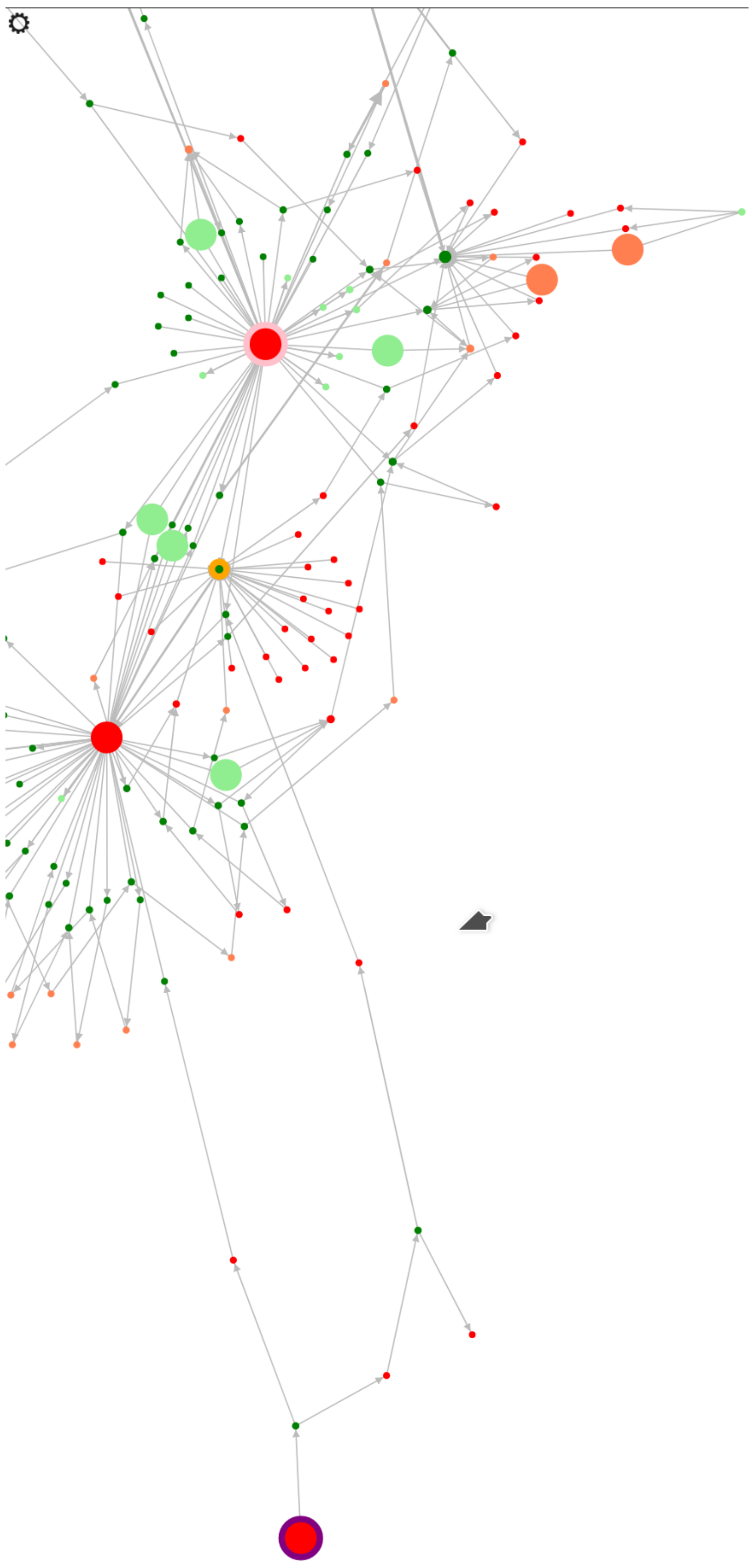


Figure 5.3: A detailed view of the graph shown in Figure 5.2. Orange stroked nodes have undergone hairball pruning. A purple stroke (leftmost node) marks the start of the path query and the pink stroke (right hand, large node) marks the end node of the paths. The edge width corresponds to the transaction output in BTCs. The super nodes of motifs shown in light green and orange with a large diameter are clearly visible.

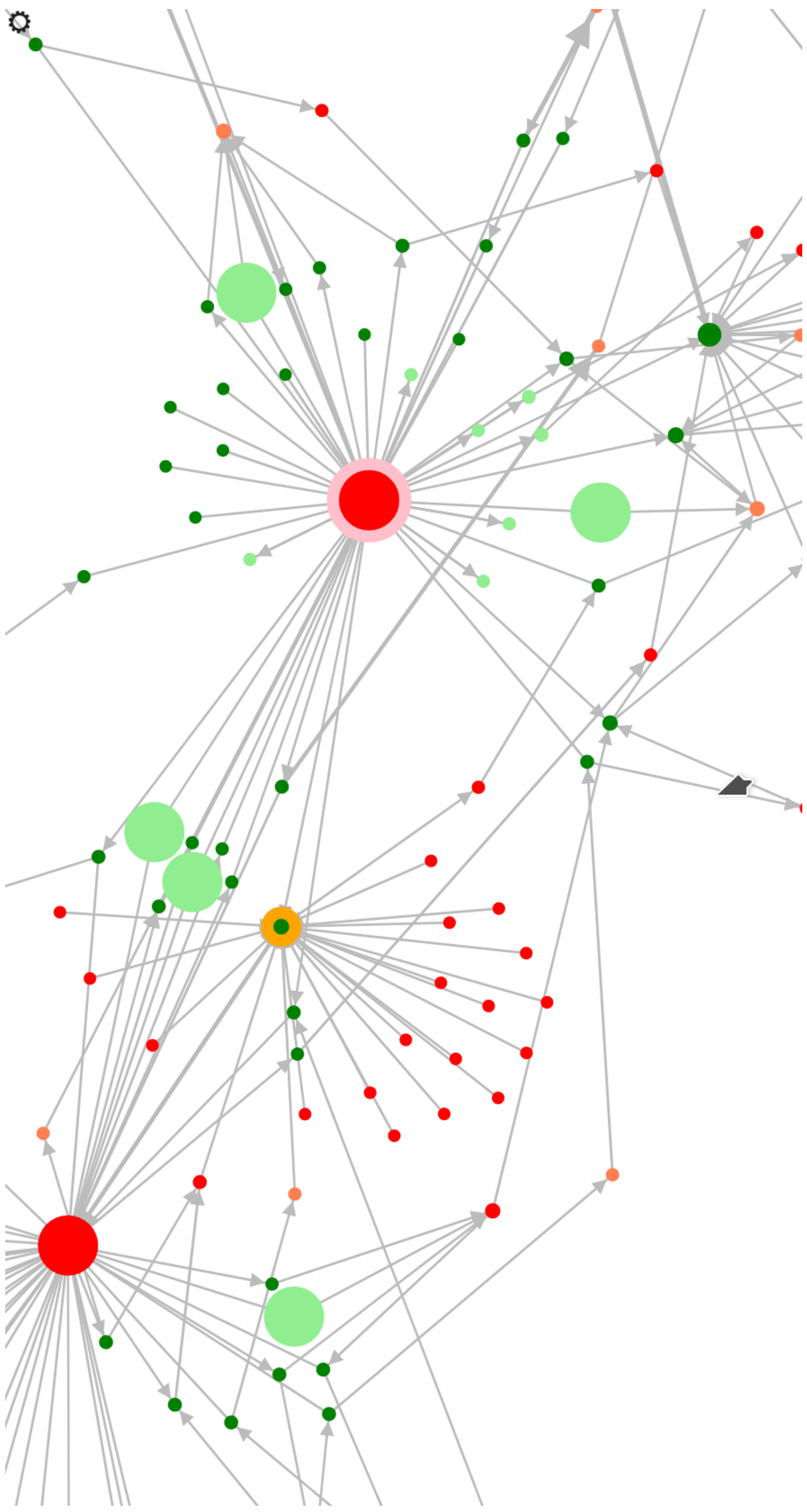


Figure 5.4: An even more detailed view of the client visualization as shown in Figure 5.2 and Figure 5.3. The hairball node visible with the orange stroke is shown in the center. The fully saturated red nodes that are connected to it, have a node degree ≥ 2 in the proxy server. During the graph minimization step, the nodes linked to them have been removed in the proxy because they do not contribute information as described in Section 5.5.3.

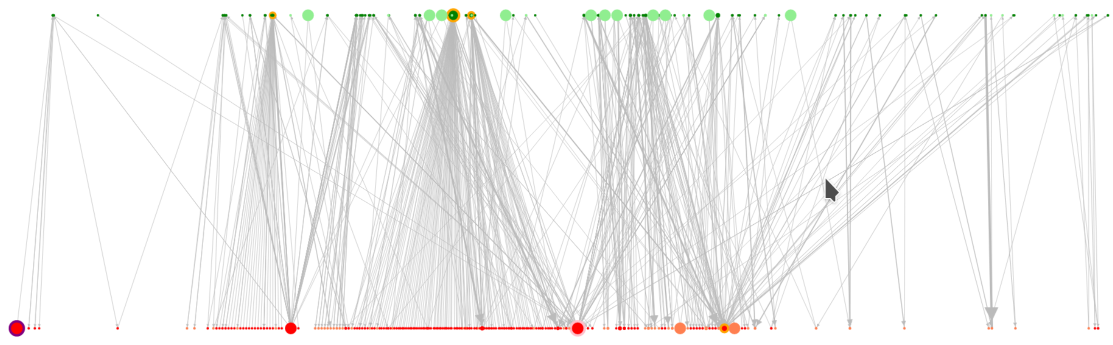


Figure 5.5: This Figure shows the same graph as in the previous figures, but with the time constrained layout as described in Section 3.2. The timestamps contained in the transactions (green nodes) are mapped to the x-axis. This view shows how BTCs were sent between two addresses over time.

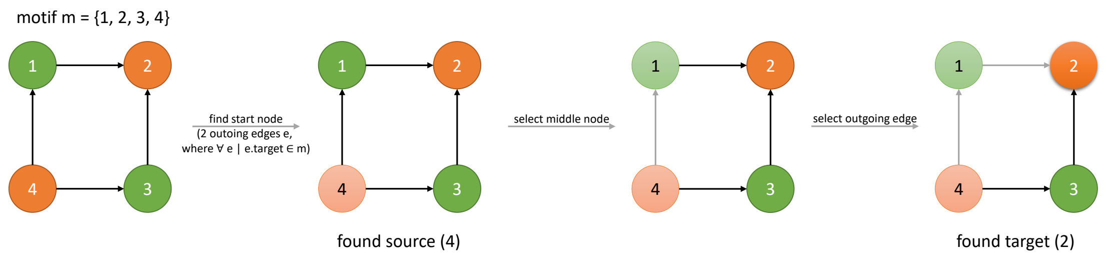


Figure 5.6: This Figure shows how the orientation of the motifs is determined schematically. Due to graph-tool sorting the vertex indices in ascending order during motif detection, the orientation of motifs is lost. For the rotationally sensitive fan-motif the orientation needs to be established before any further processing is possible.

CHAPTER 6

Results

For testing the performance of the methods proposed in the thesis, different metrics can be applied. It is possible to compare the approaches with each other by measuring run time. However, with the defined goal of minimizing the blockchain TX-graph in a meaningful way, it is important to also look at by how much the proposed system reduces the node and edge count from the original dataset. It also gives insight into whether some operations justify the complexity effort or if they do not contribute much to the graph minimization.

6.1 Processing Speed

All benchmarks were performed on an average computer with components that are, at the time of writing this thesis, several years old. Although, as in previous chapters mentioned, it is possible to have the database, the proxy server, and the client running on one physical machine, the database was stored off-site and only the proxy and the client were running on the same computer.

The database server is using an Intel(R) Xeon(R) E5-2640v2 @ 2 GHz CPU, 128 GB RAM and 1 GBit/s connectivity. The database itself is encapsulated in a virtual machine with 4 cores and 32 GB RAM.

The processing proxy runs on an Intel(R) Xeon(R) E5-2630v2 @ 2.6 GHz with 8 GB of ECC RAM and is connected to the network with a measured speed of 27 MBit/s. Network speeds between the client and the processing proxy are negligible because the data never leaves the physical machine. However, running them on the same physical machine results in shared computing resources. The measurements were run multiple times and the average runtime of measurements was taken.

Time (m:s)	Graph Size (V / E)	DB Query	Graph Update	Graph → JSON
00:00	2,413 / 3,098	2,313	3,579	932
00:14	3,797 / 4,927	2,813	4,264	1,431
00:45	5,404 / 7,094	2,795	3,917	1,683
01:50	11,045 / 13,342	6,793	13,557	3,597
02:35	12,682 / 15,525	3,978	7,718	3,301
04:31	19,755 / 23,770	8,302	26,249	3,978
5:40	21,134 / 25,712	4,312	10,357	3,899
7:41	22,809 / 28,208	2,610	7,823	5,100
8:57	23,691 / 29,524	2,540	9,153	4,954

Table 6.1: Timings of different steps while expanding the neighborhood of graphs in milliseconds. It can be seen that querying the database is independent of graph size. However, updating the graph depends on the number of nodes and edges to insert and the size of the graph they are inserted into. Runtimes for transforming the graph into JSON increase sublinear with graph size.

6.1.1 Graph Expansion Timings

The various stages of the processing pipeline have different runtimes depending on how much data they have to process. Generally, database queries, graph insertion, graph minimization, and JSON creation are the most resource intense operations. Although parallelization was a focus while implementing the system, inserting new data into the in-memory graph is required to be sequential. This results in stalls and waiting of processes for resources to be freed. An analysis of the neighborhood expansion, the database path query, and the in-memory path query performance is given in the following paragraphs. Tables 6.1 to 6.3 shows the progression of time, in minutes and seconds, in the first column. The second column shows the graph size in vertices and edges. DB Query, Graph Update, and for the third table Graph Path Search and State Update, and Graph → JSON show the runtime of these steps during processing in milliseconds rounded to one decimal place. Table 6.1 shows various graph sizes and the respective timings for the neighborhood expansion. Stalling during graph insertion is clearly seen when a certain operation requires more time to compute than later in time even though the graph is smaller. This can be seen when the graph grows by 5,641 from 5,404 nodes to 11,045 nodes at time 1:50 and by 1,637 nodes from 11,045 to 12,682 nodes at 2:35 and their respective timings are compared. The duration of the graph update for the insertion of 5,641 nodes takes almost twice as long.

Apart from neighborhood expansion, the graph database is also queried for new paths simultaneously. The operation of finding paths in the database differs from the neighborhood expansion because updating the graph is only necessary for the number of nodes in the newly found path. Typically, the result only includes a few dozen nodes. Compared to the update time of the neighborhood expansion, adding a new path from the database is much faster. However, the main bottleneck for path finding in the TX-graph is with

Time (m:s)	Graph Size (V / E)	DB Query	Graph Update	Graph → JSON
00:00	2,413 / 3,098	509	160	1.0
00:14	3,797 / 4,927	705	191	0.1
00:45	5,404 / 7,094	7,563	167	5.9
01:50	11,045 / 13,342	1,343	275	0.1
02:35	12,682 / 15,525	12,057	354	0.1
04:31	19,755 / 23,770	2,156	446	0.1
5:40	21,134 / 25,712	13,044	479	0.1
7:41	22,809 / 28,208	2,679	441	0.1
8:57	23,691 / 29,524	14,194	457	0.1

Table 6.2: Timings of different steps while loading new paths between two nodes of the Bitcoin TX-graph from the graph database in milliseconds. Searching paths in the graph database is very unpredictable due to its dependency on graph topology. Once a path has been found only a few dozens of nodes need to be inserted into the graph and transformed into JSON. Therefore, these steps are very fast compared to the database query.

the database itself. Table 6.2 shows that the database query runtime exceeds all other steps during processing. Generally, query times increase over time. Although this is not strictly monotonous, the more paths have been found, the larger the search space has to be. Timings of the database query, however, are more unpredictable than the other steps in the processing pipeline as it depends on the graph topology. Once a path is found, the data returned by the database is very small consisting only of a few nodes and edges and therefore updating the graph, the search state, and generating an updated JSON is very fast.

Finding new paths in the in-memory graph is much faster than querying the database. For this stage, generating an update for the visualization in the graph is the most time intensive task. The total runtime depends on the number of paths found and the length of them. Generating JSON from the graph scales with the number of nodes that need to be updated. As can be seen in Table 6.3, the native C++ implementation of the library, used to find new paths in the in-memory graph, shows its superior performance compared to the graph database (DB Query) path search. The graph-tool library takes only a fraction of the time to find paths. Additionally, runtimes for analyzing the graph generally increase, but not as drastically as expected. The difference for analyzing the paths only increases by about 30 milliseconds for an increase of 20,000 nodes and 25,000 edges in the graph.

The results support the assumption that only querying the graph database might introduce long waiting times at the client to receive new data. Neighborhood expansion continuously adds new data to visually give feedback in the client and provide the user with more information. Having a path search for the in-memory graph is necessary to support finding new paths that might have been introduced with the neighborhood expansion.

Time (m:s)	Graph Size (V / E)	Graph Path Search	State Update	Graph → JSON
00:00	2,413 / 3,098	143	1.5	468
00:14	3,797 / 4,927	22	1.1	448
00:45	5,404 / 7,094	151	2.8	1,155
01:50	11,045 / 13,342	105	2.1	1,126
02:35	12,682 / 15,525	94	4.5	667
04:31	19,755 / 23,770	138	4.4	477
5:40	21,134 / 25,712	174	24	1,966
7:41	22,809 / 28,208	172	5.2	2,765

Table 6.3: Timings of different steps while searching new paths in the in-memory graph in milliseconds. Path search with graph-tool only requires a few hundred milliseconds and is much faster than querying the database. Once a path is found updating the search state and sending the found paths to the client is very fast due to a small number of nodes and edges.

Furthermore, the runtime cost of searching new paths in the in-memory graph are negligible.

6.1.2 Graph Minimization

Graph growth not only influences when new data is inserted into the graph or sent to the client. Foremost, costly graph minimization steps are influenced by increasing graph size. By analyzing the graph minimization steps in detail, it is possible to see whether some steps pay off in terms of graph reduction efficacy and run time. To see the effects of minimization techniques for different graph sizes, three points in time of loading the blockchain are taken as an example.

The steps involved in reducing the graph before sending are:

1. Generating the TX-graph, effectively removing blocks from the graph.
2. Filtering disconnected components from the graph that are not connected to the main component. This removes floating nodes that can not be reached from the nodes of the initial user query.
3. Prune nodes that have many neighbors with $deg(n) = 1$ and are therefore not contributing to the topological information but require many elements on screen.
4. Remove all nodes with $deg(n) = 1$ that only have one outgoing edge. This means that they have not yet been expanded and also do not add information about the flow of BTCs.
5. Replace all fan-motifs with one super node and remove all loop-motifs from the graph.

6. Remove already sent nodes and nodes that do not need updating.

Tables 6.4 to 6.6 show the timing results and the vertex and edge reduction for each step during preparing the in-memory graph for sending it to the client.

- *TX subgraph* results from the removal of blocks and their connected edges from the graph.
- The *component graph* is the resulting graph after only the connected component containing the initial user query is kept.
- After all nodes with a high degree have been analyzed and all connected nodes with $deg(n) = 1$ have been removed, the *hairball graph* is left.
- Pruning all outer nodes with a degree of one and only outgoing edges, results in the *rim pruned graph*.
- After motif detection and replacement and the introduction of additional super nodes the graph is referred to as *motif graph*.
- Lastly, all the already sent nodes and edges are removed and only nodes that have not yet been sent to the client or nodes that need updating are sent to the client. This results in the *unsent graph*.

For a graph with 959 nodes and 1,196 edges this results in the measurements shown in Table 6.4. The differences in the number of node and edge reduction are not yet clearly visible due to a small graph. However, generating the motif graph and the hairball graph take the longest. While during the hairball removal step only 118 nodes and edges are removed, motif reduction only decreases the graph size by 32 nodes and 81 edges. However, motif reduction takes almost twice the computation time. For such a small graph it is already more than 300 milliseconds.

When the graph grows to around 11,000 nodes and 14,000 edges, the differences in performance become even more obvious. With a runtime of almost 12 seconds the motif reduction only removed 69 nodes and 498 edges. This is compared to the hairball removal, that takes about half the time, and removes almost half of all nodes in the original graph. Measurements that capture this behavior can be seen in Table 6.5.

This holds if the graph increases even further. Table 6.6 shows a graph with around 23,000 nodes initially and the reduction in each step of the processing pipeline. Although hairball removal takes a long time for large graphs and motif removal is only 3.8 seconds slower, the graph reduction efficiency is still the best for all steps involved.

Reduction Method	# V_prev	# V_post	# E_pre	# E_post	Time
TX Subgraph	959	356	1,196	499	11.9
Component Graph	356	356	499	499	13.5
Hairball Graph	356	238	499	381	184.5
Rim Pruned Graph	238	227	381	363	32.0
Motif Graph	227	195	363	282	319.8
Unsent Graph	195	195	282	0	15.0

Table 6.4: During sending the graph to the client and before JSON generation, several graph minimization steps are applied. This table shows the reduction of nodes and edges in each step as well as how long it takes to perform the steps. Even for small graphs (959 vertices and 1196 edges) motif search already is computationally intensive and does not contribute much to the overall graph reduction.

Reduction Method	# V_prev	# V_post	# E_pre	# E_post	Time
TX Subgraph	11,538	9,789	14,148	9,455	24.8
Component Graph	9,789	8,706	9,455	9,129	17.8
Hairball Graph	8,706	4,207	9,129	4,630	6405.6
Rim Pruned Graph	4,207	2,409	4,630	2,802	145.8
Motif Graph	2,409	2,340	2,860	2,362	11,935.6
Unsent Graph	2,340	2,340	2,362	182	88.5

Table 6.5: The efficacy and timings of graph minimization steps for a graph with 11538 vertices and 14148 edges. It can be seen that motif search for larger graphs is even less performant and does not scale well. As with the small graph in Table 6.4 the reduction of nodes in the motif aggregation step is very small.

Reduction Method	# V_prev	# V_post	# E_pre	# E_post	Time
TX Subgraph	23,218	19,613	28,600	18,267	5.6
Component Graph	19,613	16,693	18,267	17,366	136.1
Hairball Graph	16,693	6,247	17,366	6,920	11,608.0
Rim Pruned Graph	6,247	3,685	6,920	4,313	236.0
Motif Graph	3,685	3,580	4,399	3,792	15,395.5
Unsent Graph	3,580	3,580	3,792	143	79.9

Table 6.6: Reduction performance of the employed graph minimization techniques for a graph with 23,218 nodes and 28,600 edges. Although hairball removal takes a long time, compared to the performance of motif reduction, the cost / performance ratio is a lot lower.

6.2 Expert Reviews

After showing the proof of work client as shown in Section 5.4 to domain experts, their feedback was very promising. Although it was made clear that the processing proxy is without value if the visualization is not able to display paths, patters, motifs, etc., they have highlighted that such a visualization system is sought after in the Bitcoin community. In the cryptocurrency community there are many ongoing efforts to solve the problem of visualizing large blockchain graphs and that, to their knowledge, no one has yet found a solution to efficiently visualize the TX-graph. They also stated that with a more polished client, they would be eager to use the system for their own research. According to the experts, the client user interface needs improvements in terms of flexibility and usability. Nonetheless, they already have concrete projects in mind that could benefit from the system. These projects involve analyzing mining pool payout patterns or spending habits of miners. They also explicitly highlighted the benefit of the entity graph as used by Reid and Harrigan [11] presented in Section 2.2.3 and would be interested in adding this type of graph to the system. While our system operates on the TX-graph itself, Reid and Harrigan compute an entity graph based on clustering of addresses. By doing so they analyze spending habits on the entity level instead of the address level. Additionally, for solving ongoing problems the experts face, the dataset needs to be expanded to cover the whole blockchain. With the current time frame not enough data is available to answer their specific questions.

CHAPTER 7

Discussion

The results in the previous chapter show that bottom-up and top-down approaches to reduce the data to be rendered and immediately start the visualization are a suitable approach to hide data processing latencies and improve scaling of visualization performance with large graphs. Although the approach presented in this thesis focuses on Bitcoin, by using different aggregation and querying strategies it could be extended to work with different types of graph data.

7.1 Performance Discussion

Cryptocurrencies as a recent monetary trend with an ever increasing amount of data and growing relevance for various stakeholders require novel solutions. While the scientific communities of visual analytics and cryptocurrencies are very active, there is still room for improvements of analyzing blockchain data. Naturally, this stems from the fact that blockchain technologies are very new and pose a unique set of requirements and challenges. A limiting factor of the related work on node-link visualizations is scaling with data size and the performance degradation if the data becomes larger than a few thousand nodes. So far, Bitcoin visualization projects have focused on pruning the data through constraining the visualized timeframe [13], topology [12] or by using a completely new kind of visualization that makes understanding network topology difficult or impossible [15, 58, 14]. Additionally, some related visualizations require a static preprocessing step that makes it difficult to add additional data to the data storage. In a dynamic system as Bitcoin, however, ease of adding data to the data storage without costly processing is paramount.

The methodology proposed in this thesis improves on these shortcomings by choosing a hybrid approach of top-down graph minimization with bottom-up progressively streaming data from the data source. Using a graph database for storing makes it easy to add additional data to the database. Visualizing the blockchain is done over time and the longer the visualization continues to load data, the more complete the visualization will be. Through aggregation and user defined pruning thresholds, a significant reduction of nodes and edges is achieved. That allows for visualization on readily available hardware. Through progressive loading, an analyst can start inspecting the TX-graph immediately after the system has been queried. Although, the focus of the presented visualization client is on topological information, other types of visualizations can be created using the proxy server. Section 7.2 goes into detail about possible improvements to the proxy server.

The performance measurements shown in Chapter 6 make it clear that the approach of dynamic graph minimization is very effective and possible without special hardware. After measuring the performance of each step involved in aggregating the graph, the bad performance of network motifs is unexpected. Through visualization of the Bitcoin TX-graph it was established that reducing node count in the visualized graph is very promising. After consultation with domain experts, it has been established that other network motifs, such as loops between TX and addresses, might be more prevalent. Although the performance of motif reduction is bad compared to other minimization steps, more benchmarks with different network motifs need to be done to fully exclude motifs from the list of promising reduction techniques. On the other hand, pruning neighbors of nodes with very large degree that have no outgoing edges themselves, has proven itself as the most effective reduction method. While it seems that this does not scale well with an increasing size of the graph, the problem might be avoided by implementing this step using NumPy [80], mitigating the comparably slow performance of Python.

Although the approaches presented in this thesis work well, there is still room for improvement of the processing speed of the graph. Currently the proxy server is considered experimental and before it can be deployed in a real world analysis setting, it needs more work. Additionally, even if the server reduces the graph data efficiently at, some point the number of elements to render will be too high and the performance degrades. The way the Bitcoin protocol works makes it difficult to follow money between users. The work presented by Reid and Harrigan [11] uses an entity graph that makes this information apparent and it could be better to visualize this entity graph in order to convey the information we want to show to the user. Although an additional preprocessing step is necessary for creating this entity graph, combining the true TX-graph and the entity graph might be a better solution. The entity graph was not chosen in the first place, as visualization of ground truth blockchain data was a goal. Despite some limitations and preliminary functionality, the implemented prototype and methodology presented in this thesis, are useful for future blockchain research.

7.2 Future Work

During the implementation of this thesis' methods, several roadblocks and obstacles have been found. Most are related to speed improvements that either stem from initial design decisions or from the used technology stack. Possible speed improvements include a change from internally used Python arrays to NumPy [80] arrays. NumPy, similar to graph-tool, has native implementations of many algorithms and therefore increases the speed of the current implementation [81]. Another possible technology change is a migration from Neo4j to OrientDB [82]. According to the website of OrientDB, it is faster than Neo4j and therefore the proxy server could benefit from this change. However, self reported performance benchmarks that have not been scientifically scrutinized are to be taken with a grain of salt and further benchmarking with the blockchain data is necessary. In relation to the database, loading the complete, currently available, Bitcoin blockchain into the database is yet to be done. Creating a system that constantly updates the database when new blocks are mined would also be a helpful feature.

In terms of algorithmic improvements it is possible to aggregate paths in the graph. After finding paths and pruning nodes that do not contribute information to the system, it is possible that long paths, where all nodes have $deg(n) = 2$, can be collapsed into one single super edge. This would further reduce the node and edge count in the final graph to be visualized. Pruning nodes that have not been expanded by the neighborhood expansion could also performed if a user wishes to. This functionality is not implemented yet but it would result in a drastically reduced graph size. Exact reduction performance needs to be analyzed first and this assumption stems from visually analyzing the system during the work on this thesis. Currently, the system is creating a JSON data string for visualization with D3.js [71]. Returning data in a different format to the client would allow for more versatile visualization clients and could lead to clients using different visualization frameworks. This might be at the expense of the universality of web technologies, but could increase scalability of the visualization client. Using D3.js only between 5,000 and 10,000 nodes are feasibly to visualize. Although the proxy server can handle larger graph sizes internally, graph reduction needs to ensure that the client does not have to visualize too many nodes and edges.

CHAPTER 8

Conclusion

This thesis presented a novel bottom-up/top-down hybrid approach for visualizing and and analyzing Bitcoin. As presented in Chapter 2, the current state of visual analytics of Bitcoin is still an unsolved problem. Especially, following transferred BTCs, given Bitcoin's design choices, through the TX-graph is a difficult task. By combining progressive streaming with top-down aggregation, we showed that it is possible to analyze the TX-graph of the blockchain through visualization of a node-link diagram. User guided graph minimization through aggregation, using heuristics and knowledge about Bitcoin, has proven to be a suitable approach to scale with the dataset size. At the same time, node-link diagrams show graph structures in a visually approachable way. However, force-directed node-link diagrams do not scale well for more than a few thousand nodes [65]. Reducing the amount of data that reaches the rendering stage in the client is an effective solution to this problem. The graph minimization and progressive streaming approach presented in this thesis allows for processing graphs with a size of more then 25,000 nodes and edges. Aggregation of graph motifs and pruning of nodes that do not contribute necessary information allows for visualizing large graphs. While other related work [15, 58, 14] has focused on finding novel visualization types to mitigate the hairball problem [49] and scaling issues of graphs, the methodology presented in this thesis addresses these problems of node-link diagrams. With the approach presented in this thesis it is possible to visualize node-link diagrams of large graphs that exceed current capabilities found in the related work.

Another way of analyzing the Bitcoin TX-graph is to summarize addresses into clusters of entities. The resulting data structure contains clusters of parties participating in the Bitcoin network. However, after loading the blockchain this approach needs a preprocessing step where the whole blockchain is scanned and addresses are clustered according to the TX in which they occur. This makes updating the underlying data storage more computationally expensive [11]. The methodology presented in this thesis operates on the TX-graph as found in the blockchain itself. This saves the processing cost when an update of the blockchain occurs about every 10 minutes. Bitcoin experts have indicated the usefulness of having an entity graph and creating this additional, clustered graph is reserved for future work.

One of the two main contributions of this work is to tackle the scaling problem of big data visualization through a combination of progressive streaming and aggregation. Especially cryptocurrencies as a fairly new phenomenon have not yet readily available solutions to visualize their data. The second contribution is the implementation of a proof-of-concept that allows experts to query the blockchain for analysis. One possible query is to find paths, and therefore the flow of BTCs, from one address to another one. The second visualization type is to see how a node is embedded in the TX-graph. This shows how an address interacts with other addresses and it allows to analyze spending patterns. The benchmark results have shown that not all graph minimization steps provide a good cost-performance trade off. Especially motif detection as currently implemented is inefficient and a bottleneck. Further analysis in collaboration with Bitcoin researchers is necessary to find better suitable motifs for graph minimization and visual mapping of the client visualization. Nonetheless, the system already achieves graph reduction by an order of magnitude and it is considered, with some minor additions, a valuable contribution to Bitcoin visualizations by Bitcoin experts.

List of Figures

List of Tables

List of Algorithms

Glossary

Bitcoin Name of the first decentralized currency system based on cryptography ix, xi, xiii, 3, 5–9, 11–14, 16, 17, 20, 23, 26, 28–32, 34–36, 44, 46, 47, 53, 75, 79, 81–83, 85–88, 91, 95

blockchain The central data structure in which the financial information of Bitcoin is saved xi, 6, 7, 11–13, 15, 16, 21, 23–26, 28, 31, 33, 34, 46, 47, 49, 53, 54, 58, 73, 76, 79, 81–83, 85–88, 95

client-server architecture A system architecture that splits work and functionality between a client and a server. Several different approaches exist depending on how much work is done in which component. 23

coinbase transaction A special transaction in each newly created block in the blockchain that generates new BTCs. 32

Cypher A query language designed for graph databases. It is declarative and supports traditional query functionality as well as graph related pattern matching. 47, 49, 95

DDoS Distributed denial of service. An attack on a service or server by flooding it with fake requests and exhausting all its resources 12

Neo4j An implementation of a no-SQL database for graphs. It supports Cypher as a query language and graph theoretical queries such as shortest paths. 37, 47, 49, 51, 52, 83, 89

Acronyms

API Application Program Interface 47, 50

BTC bitcoin xi, 5–7, 13, 14, 21, 32, 33, 35–38, 42, 51, 62, 69, 71, 76, 85, 86, 88, 89, 95

DOM Document Object Model 50

FIFO first in, first out 29

JSON JavaScript Object Notation 39, 40, 44–46, 50, 52, 55, 67, 74, 75, 78, 83, 91

OGM Object Graph Mapper 47, 51, 52

p2p peer-to-peer 5, 6

SQL Structured Query Language 47, 95

SVG Scalable Vector Graphics 50

TX transaction xi, 7, 8, 12, 14, 17, 25, 33–38, 46, 47, 50, 51, 62, 77, 82, 85, 87, 88

TX-graph transaction graph 9, 11, 13–15, 24, 25, 28, 29, 34, 35, 42, 44, 73–76, 79, 82, 85, 86, 91

USD US dollars 5, 8

Bibliography

[1] Charles Arthur. Tech giants may be huge, but nothing matches big data. https://www.theguardian.com/technology/2013/aug/23/tech-giants-data, August 2013. [Online; accessed 14-May-2018].

[2] Manuel Castells. The information age: Economy, society, and culture. volume i: The rise of the network society, 1996.

[3] Mark Mulcahy. Big data statistics & facts for 2017. https://www.waterfordtechnologies.com/big-data-interesting-facts/, February 2017. [Online; accessed 14-May-2018].

[4] Bob Landstrom. The laws of technology: Driving demand in the data center. https://www.interxion.com/blogs/2014/11/the-laws-of-technology-driving-demand-in-the-data-center/, November 2014. [Online; accessed 14-May-2018].

[5] Alicia Key, Bill Howe, Daniel Perry, and Cecilia Aragon. Vizdeck: self-organizing dashboards for visual analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 681–684. ACM, 2012.

[6] Selan Dos Santos and Ken Brodlie. Gaining understanding of multivariate and multidimensional data through visualization. *Computers & Graphics*, 28(3):311–325, 2004.

[7] Christian Tominski. *Event based visualization for user centered visual analysis.* PhD thesis, University of Rostock, 2006.

[8] Annika Baumann, Benjamin Fabian, and Matthias Lischke. Exploring the bitcoin network. In *WEBIST (1)*, pages 369–374, 2014.

[9] Hiroki Kuzuno and Christian Karam. Blockchain explorer: An analytical process and investigation environment for bitcoin. In *Electronic Crime Research (eCrime), 2017 APWG Symposium on*, pages 9–16. IEEE, 2017.

[10] Damiano Di Francesco Maesa, Andrea Marino, and Laura Ricci. Detecting artificial behaviours in the bitcoin users graph. *Online Social Networks and Media*, 3-4:63 – 74, 2017.

[11] Fergal Reid and Martin Harrigan. An analysis of anonymity in the bitcoin system. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom)*, pages 1318–1326. IEEE, 2011.

[12] Stefano Bistarelli and Francesco Santini. Go with the-bitcoin-flow, with visual analytics. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, page 38. ACM, 2017.

[13] Dan McGinn, David Birch, David Akroyd, Miguel Molina-Solana, Yike Guo, and William J Knottenbelt. Visualizing dynamic bitcoin transaction patterns. *Big data*, 4(2):109–119, 2016.

[14] Christoph Kinkeldey, Jean-Daniel Fekete, and Petra Isenberg. Bitconduite: Visualizing and analyzing activity on the bitcoin network. In *EuroVis 2017-Eurographics Conference on Visualization, Posters Track*, page 3, 2017.

[15] Giuseppe Di Battista, Valentino Di Donato, Maurizio Patrignani, Maurizio Pizzonia, Vincenzo Roselli, and Roberto Tamassia. Bitconeview: visualization of flows in the bitcoin transaction graph. In *Visualization for Cyber Security (VizSec), 2015 IEEE Symposium on*, pages 1–8. IEEE, 2015.

[16] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[17] A history of bitcoin. `https://www.smithandcrown.com/a-history-of-bitcoin/`. [Online; accessed 20-March-2018].

[18] Krzysztof Okupski. Bitcoin developer reference. *Eindhoven*, 2014.

[19] Address reuse. `https://en.bitcoin.it/wiki/Address_reuse`. [Online; accessed 20-March-2018].

[20] Jameson Lopp. What the data tells us about bitcoin in 2017. `https://www.coindesk.com/bitcoin-2017-stats/`, January 2018. [Online; accessed 23-January-2018].

[21] Svenja Schröder, Markus Huber, David Wind, and Christoph Rottermanner. When signal hits the fan: On the usability and security of state-of-the-art secure mobile messaging. In *First European Workshop on Usable Security (EuroUSEC 2016)*, 2016.

[22] Jeffrey M Stanton, Kathryn R Stam, Paul Mastrangelo, and Jeffrey Jolton. Analysis of end user security behaviors. *Computers & security*, 24(2):124–133, 2005.

[23] Cryptocurrency market capitalizations. `https://coinmarketcap.com`, Constantly Updated. [Online; accessed 29-January-2018].

[24] Bitcoin stats. `https://blockchain.info/stats`, Constantly Updated. [Online; accessed 29-January-2018].

[25] Bitcoin charts. https://blockchain.info/charts, Constantly Updated. [Online; accessed 29-January-2018].

[26] Katie Richard. Data visualization: Processing images at 1250mb/s. https://wiredcraft.com/blog/data-visualizations-images/, July 2015. [Online; accessed 14-May-2018].

[27] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: Real-time visual querying of big data. In *Computer Graphics Forum*, volume 32, pages 421–430. Wiley Online Library, 2013.

[28] Niklas Elmqvist and Jean-Daniel Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, 2010.

[29] Stef Van Den Elzen and Jarke J Van Wijk. Multivariate network exploration and presentation: From detail to overview via selections and aggregations. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2310–2319, 2014.

[30] Bryan McDonnel and Niklas Elmqvist. Towards utilizing gpus in information visualization: A model and implementation of image-space operations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1105–1112, 2009.

[31] Danyel Fisher. Incremental, approximate database queries and uncertainty for exploratory visualization. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 73–80. IEEE, 2011.

[32] Chad A Steed, Daniel M Ricciuto, Galen Shipman, Brian Smith, Peter E Thornton, Dali Wang, Xiaoying Shi, and Dean N Williams. Big data visual analytics for exploratory earth system simulation analysis. *Computers & Geosciences*, 61:71–82, 2013.

[33] Jaegul Choo and Haesun Park. Customizing computational methods for visual analytics with big data. *IEEE Computer Graphics and Applications*, 33(4):22–28, 2013.

[34] Daniel Keim, Huamin Qu, and Kwan-Liu Ma. Big-data visualization. *IEEE Computer Graphics and Applications*, 33(4):20–21, 2013.

[35] Leishi Zhang, Andreas Stoffel, Michael Behrisch, Sebastian Mittelstadt, Tobias Schreck, René Pompl, Stefan Weber, Holger Last, and Daniel Keim. Visual analytics for the big data era—a comparative review of state-of-the-art commercial systems. In *Visual Analytics Science and Technology (VAST), 2012 IEEE Conference on*, pages 173–182. IEEE, 2012.

[36] Rae Earnshaw. *Visual Analytics and Big Data*, pages 25–46. Springer International Publishing, Cham, 2018.

[37] Alper Sarikaya and Michael Gleicher. Using webgl as an interactive visualization medium: Our experience developing splatterjs. In *Proceedings of the Data Systems for Interactive Analysis Workshop. IEEE*, 2015.

[38] Martijn Tennekes and Edwin de Jonge. Top-down data analysis with treemaps. In *IMAGAPP/IVAPP*, pages 236–241, 2011.

[39] Ben Sneiderman. The eyes have it: A task by data type taxonomy for information visualization. In *Proceedings IEEE Symposium on Visual Languages*, volume 96, 1996.

[40] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[41] Shai S Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon. Network motifs in the transcriptional regulation network of Escherichia coli. *Nature genetics*, 31(1):64, 2002.

[42] David A Bader and Kamesh Madduri. Snap, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[43] Yuanyuan Tian, Richard A Hankins, and Jignesh M Patel. Efficient aggregation for graph summarization. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 567–580. ACM, 2008.

[44] Eric A Bier, Maureen C Stone, Ken Pier, William Buxton, and Tony D DeRose. Toolglass and magic lenses: the see-through interface. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 73–80. ACM, 1993.

[45] Tomer Moscovich, Fanny Chevalier, Nathalie Henry, Emmanuel Pietriga, and Jean-Daniel Fekete. Topology-aware navigation in large networks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2319–2328. ACM, 2009.

[46] Niklas Elmqvist, Pierre Dragicevic, and Jean-Daniel Fekete. Color lens: Adaptive color scale optimization for visual exploration. *IEEE Transactions on Visualization and Computer Graphics*, 17(6):795–807, 2011.

[47] Michael J McGuffin and Igor Jurisica. Interaction techniques for selecting and manipulating subgraphs in network visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):937–944, 2009.

[48] Ethan Kerzner, Alexander Lex, Crystal Lynn Sigulinsky, Timothy Urness, Bryan William Jones, Robert E Marc, and Miriah Meyer. Graffinity: Visualizing connectivity in large graphs. In *Computer Graphics Forum*, volume 36, pages 251–260. Wiley Online Library, 2017.

[49] My 15 minutes of fame as a b-list gamergate celebrity. `http://www.brianckeegan.com/2014/10/my-15-minutes-of-fame-as-a-b-list-gamergate-celebrity/`. [Online; accessed 20-March-2018].

[50] Daniel Archambault, Tamara Munzner, and David Auber. Grouseflocks: Steerable exploration of graph hierarchy space. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):900–913, 2008.

[51] Frank Van Ham and Adam Perer. "search, show context, expand on demand": supporting large graph exploration with degree-of-interest. *IEEE Transactions on Visualization and Computer Graphics*, 15(6), 2009.

[52] Alexander Lex, Christian Partl, Denis Kalkofen, Marc Streit, Samuel Gratzl, Anne Mai Wassermann, Dieter Schmalstieg, and Hanspeter Pfister. Entourage: Visualizing relationships between biological pathways using contextual subsets. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2536–2545, 2013.

[53] Robert Pienta, Minsuk Kahng, Zhiyuan Lin, Jilles Vreeken, Partha Talukdar, James Abello, Ganesh Parameswaran, and Duen Horng Chau. Facets: Adaptive local exploration of large graphs. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 597–605. SIAM, 2017.

[54] Matthias Lischke and Benjamin Fabian. Analyzing the bitcoin network: The first four years. *Future Internet*, 8(1):7, 2016.

[55] Paolo Tasca, Adam Hayes, and Shaowen Liu. The evolution of the bitcoin economy: Extracting and analyzing the network of payment relationships. *The Journal of Risk Finance*, 19(2):94–126, 2018.

[56] Kevin Liao, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. Behind closed doors: measurement and analysis of cryptolocker ransoms in bitcoin. In *Electronic Crime Research (eCrime), 2016 APWG Symposium on*, pages 1–13. IEEE, 2016.

[57] Shaun Giudici. Exploratory analysis of bitcoin data. `http://www.bitcoinlinks.net/tag/transaction-visualizations`. [Online; accessed 14-May-2018].

[58] Petra Isenberg, Christoph Kinkeldey, and Jean-Daniel Fekete. Exploring entity behavior on the bitcoin blockchain. In *Posters of the IEEE Conference on Visualization*, 2017.

[59] Bitcoin links - tx visualizations. http://www.bitcoinlinks.net/tag/transaction-visualizations. [Online; accessed 19-February-2018].

[60] Global bitcoin nodes distribution. https://bitnodes.earn.com. [Online; accessed 19-February-2018].

[61] Bit bonkers! http://bitbonkers.com. [Online; accessed 19-February-2018].

[62] Bitnodes. https://bitnodes.earn.com/nodes/network-map/. [Online; accessed 19-February-2018].

[63] Blocks wizb. https://blocks.wizb.it. [Online; accessed 19-February-2018].

[64] Bitcoin transaction visualization. http://bitcoin.interaqt.nl. [Online; accessed 19-February-2018].

[65] Unconfirmed bitcoin transaction visualization. http://dailyblockchain.github.io. [Online; accessed 19-February-2018].

[66] The bitcoin big bang. https://www.elliptic.co/. [Online; accessed 19-February-2018].

[67] Protecting the integrity of digital assets. https://www.chainalysis.com. [Online; accessed 19-February-2018].

[68] Block seer. https://www.blockseer.com. [Online; accessed 19-February-2018].

[69] Edward F Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Theory, 1959*, pages 285–292, 1959.

[70] Proof of work. https://en.bitcoin.it/wiki/Proof_of_work. [Online; accessed 26-March-2018].

[71] Michael Bostock et al. D3. js. *Data Driven Documents*, 492:701, 2012.

[72] Alexandra Mai. *Visualization of Bitcoin Transactions for Forensic and Security Analysis*. Wien.

[73] Neo4J Developers. Neo4j. *Graph NoSQL Database*, 2012.

[74] Robin Edwards. Neomodel. https://neomodel.readthedocs.io/en/latest/. [Online; accessed 31-March-2018].

[75] Tiago P. Peixoto. The graph-tool python library. *figshare*, 2014.

[76] NetworkX Developers. Networkx. https://networkx.github.io. [Online; accessed 31-March-2018].

[77] Twisted Developers. Twisted. https://twistedmatrix.com/trac/. [Online; accessed 31-March-2018].

[78] Crossbar.io Technologies. Autobahn. `https://autobahn.readthedocs.io/en/latest/`. [Online; accessed 31-March-2018].

[79] Neo4J Bolt Driver Developers. Neo4j bolt driver for python. `https://github.com/neo4j/neo4j-python-driver`. [Online; accessed 31-March-2018].

[80] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[81] Renato Candido. Pure python vs numpy vs tensorflow performance comparison. `https://realpython.com/numpy-tensorflow-performance/`. [Online; accessed 14-May-2018].

[82] OrientDB Developers. Orientdb, 2012.