MASTER'S THESIS 2020

# Graph Layout Methods for Graph Databases – Performance and Analysis

Erik Danielsson, Lasse Heemann

ISSN 1650-2884
LU-CS-EX 2020-15

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2020-15

# Graph Layout Methods for Graph Databases – Performance and Analysis

**Erik Danielsson, Lasse Heemann**

# Graph Layout Methods for Graph Databases – Performance and Analysis

Erik Danielsson
dat14eda@student.lu.se

Lasse Heemann
dat14lhe@student.lu.se

May 14, 2020

Master's thesis work carried out at Neo4j, Inc.

Supervisors: Jens Oknelid, jens.oknelid@neotechnology.com
Krzysztof Kuchcinski, Krzysztof.Kuchcinski@cs.lth.se

Examiner: Flavius Gruian, Flavius.Gruian@cs.lth.se

## Abstract

Graph databases have become a strong contender against relational databases in the last few years. This is partly due to the efforts of the company Neo4j, who developed one of the most popular graph databases. In addition to the database, Neo4j also developed the Cypher query language used to send requests to the database, as well as two graph drawing tools, Bloom and Neo4j browser. These tools are used when working with the database by visualizing the data in the database. An algorithm called force based drawing is used by these two tools to produce a result, however, there are many more ways of drawing graphs.

We partnered up with Neo4j to explore, and evaluate alternative ways of visualizing the graphs. Mainly, the graph drawing methods layered, spectral, force directed and community based were implemented in Java. Their performances were analyzed by comparing the resulting images, and by experiments such as measuring how fast they could produce a result.

Analysis of the data found in graph databases showed how graph databases mainly consist of sparse data structures, which had great impact on performance. Furthermore, a subset of a graph database often consists of multiple disconnected subgraphs. By solving each subgraph individually, the drawing can be sped up significantly.

Our results found that the spectral method was unable to draw sparse graphs, and therefore unable to produce adequate drawings for graph databases. The force based approach already in use by Neo4j was slow, especially for larger graphs. However, it resulted in drawings that are easy to understand. The community method divides a connected graph into subgraphs of heavily connected nodes. The method also has very good performance, as drawing multiple small graphs is faster than drawing one big graph. Finally, the layered method is able to include contextual information from the database and present them to the user. This method also produced some of the best visual results whilst being comparable to the community methods very fast results.

Our findings were that both the community based and the layered methods were able to present additional information about the database. We also found that both of these methods were able to provide results in a much faster time.

**Keywords**: graph, drawing algorithms, graph databases, Neo4j

# Acknowledgements

We would like to thank Neo4j, as well as our supervisor Jens for the opportunity to work with them. We appreciate the freedom you let us have in our work as well as the help you provided along the way. We would also like to thank Michelle, for her invaluable help in proofreading our work. Lastly we thank Flavius for all his great feedback and comments on our work.

# Contents


**1 Introduction** **9**
- 1.1 Background . . . 9
- 1.2 Contribution Statement . . . 9
- 1.3 Neo4j Graph Database . . . 10
  - 1.3.1 Cypher Queries . . . 10
  - 1.3.2 Datasets and Graph Structure . . . 10
  - 1.3.3 Neo4j Browser . . . 12
- 1.4 Use Cases and Expectations . . . 12

**2 Previous Work** **15**
- 2.1 Layered Method . . . 16
  - 2.1.1 Cycle Removal . . . 16
  - 2.1.2 Layering . . . 17
  - 2.1.3 Node Ordering . . . 18
  - 2.1.4 Coordinate Assignment . . . 19
- 2.2 Spectral Method . . . 20
  - 2.2.1 Matrix Representation . . . 20
  - 2.2.2 Energy Minimization . . . 21
  - 2.2.3 Spectral Decomposition . . . 21
  - 2.2.4 Selecting Vectors . . . 21
- 2.3 Force Directed Method . . . 22
- 2.4 Community Method . . . 23
  - 2.4.1 Louvain . . . 23

**3 Implementation** **25**
- 3.1 Layered Method . . . 25
  - 3.1.1 Layering Algorithms . . . 26
  - 3.1.2 Node Ordering Algorithms . . . 29
  - 3.1.3 Coordinate Assignment Algorithms . . . 30
- 3.2 Spectral Method . . . 31

3.2.1 Matrix Representation . . . . . 31
3.2.2 Eigenvector Solving . . . . . 31
3.3 Force Directed Method . . . . . 32
3.4 Community Method . . . . . 32

**4 Results** **35**
4.1 Layered Method . . . . . 36
4.2 Spectral Method . . . . . 42
4.3 Force Directed Method . . . . . 42
4.4 Community Method . . . . . 42
4.5 Summary . . . . . 44

**5 Discussion** **47**
5.1 Layered Method . . . . . 48
5.1.1 Layering algorithms . . . . . 48
5.1.2 Node Ordering Algorithms . . . . . 50
5.1.3 Coordinate Assignment Algorithms . . . . . 51
5.1.4 Memory Requirements of Brandes and Köpf . . . . . 51
5.1.5 Time Complexity of Layered Methods . . . . . 52
5.2 Spectral Method . . . . . 52
5.2.1 Problems . . . . . 53
5.2.2 Time Complexity . . . . . 54
5.3 Force Directed Method . . . . . 54
5.3.1 Time Complexity . . . . . 54
5.4 Community Method . . . . . 54
5.4.1 Time Complexity . . . . . 54
5.5 Disconnected Subgraphs . . . . . 55
5.6 Query Issues . . . . . 56

**6 Conclusions** **57**
6.1 Further Research . . . . . 58
6.1.1 Alternative Methods . . . . . 58
6.1.2 Layered . . . . . 58
6.1.3 Community Algorithm . . . . . 58

**Bibliography** **61**

**Appendix A Benchmarks** **65**

**Appendix B Graph images** **71**

# Division of Labor

The entire project was conducted together, however tasks were sometimes separated between us. The report was written by both parties, and was mostly divided by the different task division. The tests as well as the analysis of the results was conducted together and thus the discussion section was written together. Feedback and help was exchanged daily.

- Broad research on interesting drawing methods was performed together before diving deeper into specific approaches.
- The structure of the program such as database interaction, graph rendering and benchmarking where implemented by Lasse.
- The Spectral method was implemented by Lasse.
- The majority of the Layered approach was designed and implemented by Erik. Lasse contributed by implementing the Brandes & Köpf algorithm as well as the query based layering.
- For the Community Method Lasse wrote the structure whilst Erik implemented the Louvain algorithm.
- Our version of the Force Based algorithm was implemented by Lasse.

# Chapter 1
# Introduction

## 1.1 Background

A graph $G$ is composed of nodes $N$, and edges $E$ connecting two nodes. This structure can be used to represent a variety of different concepts, such as nodes representing people, and edges representing friendships between them. As graphs can so easily represent different structures, they are widely used in a multitude of different fields. A relatively recent invention has been the use of graph structures in database handling. One of the most prominent developers of this invention is Neo4j, whose graph database has been available since 2007. As part of their database, they are developing a user interface which presents the graph to their customers. The user interface enables the user to filter out specific nodes and color match nodes with similar types. Neo4j's current solution uses force based drawings to represent the data, however, there are many other methods which could prove viable. This thesis aims to evaluate alternative methods for displaying graph database graphs, as well as discuss the various advantages and disadvantages of different algorithms.

## 1.2 Contribution Statement

Not much research has been conducted related to visualizing graph databases. This thesis helps bridge the gap in knowledge, by providing results and practical examples based on real implementations of different graph visualizing methods. Those who are working with graph databases, or with data of similar structures can use this knowledge to make a better decision on what visualization they should implement.

The two new algorithms described in the layered algorithm, the label layering and the query layering, provides new alternatives which might otherwise not be considered.

## 1.3 Neo4j Graph Database

Since 2002, Neo4j Inc. has been developing the Neo4j graph database. Contrary to traditional relational databases, in which data is stored in tables, graph databases store the data in a graph structure. Information entities are represented as nodes, and relations between these entities are represented as edges. One advantage of using the graph structure is that some operations can be performed much quicker. This is because the relationships are stored together with the data in the database. Consequently, finding related entities can be done in a single operation, and retrieving the data can be done in constant time. Common use cases of the database include a multitude of these traversals. Proving a connection between two distant entities would be futile in a traditional database as more than a handful of tables need to be joined, but by using a graph database, the search can span millions of relationships per second.[23]

### 1.3.1 Cypher Queries

Neo4j uses the query language called *Cypher*. Cypher was originally developed specifically for the use with Neo4j databases, but it is now available for anyone to use. After sending a query to the server, the server responds with a data package of records. The contents of the records are dependent on the query. A query like `match x=(:BankAccount) return x` results in each record being a single node, whereas the query `match x=()->()->() return x` results in each record being a path object with three nodes and two corresponding relationships.

### 1.3.2 Datasets and Graph Structure

Each node in the graph has a unique ID, one or more labels, and a set of properties assigned to it. The label shows what type of entity the node represents. Examples of this are: *Person* and *SSN*, or *Network* and *Server*. A *Person* node could have properties such as name and date of birth. The relationships between these nodes also contain labels, such as *Has_Address* which describes the relationship between an *Address* and an *Accountholder*.

Database datasets generally generate sparse graphs, in which the nodes only connect to a small number of other nodes. An example of this, is one person only being connected to a single social security number and a few locations such as their home and workplace. This person would obviously not be connected to unrelated nodes, such as another person's social security number or home address. As a result, these sparse graphs are easier to draw and understand compared to non-sparse graphs where every node is connected to most other nodes. Table 1.1 shows the average number of relationships per node for three datasets provided by Neo4j.

A fully connected graph contains $\frac{n(n-1)}{2}$ edges where $n$ is the number of nodes, whereas the datasets Neo4j provided contain somewhere between $n$ and $11n$ edges.

The specific graph files provided by Neo4j are listed and defined below.

**fraud** The fraud graph represents a banks data about their customers. It contains nodes such as *AccountHolder*, *BankAccount*, and *FinancialInstitute*. 17 labels and 15 types of relationships are found in the graph.

| | Fraud | Network | OMOP |
|---|---|---|---|
| Relationships | $3.0 * 10^5$ | $1.8 * 10^6$ | $5.1 * 10^7$ |
| Nodes | $3.1 * 10^5$ | $8.3 * 10^5$ | $5.9 * 10^6$ |
| Average degree | 0.96 | 2.17 | 8.67 |

**Table 1.1:** Sparseness of datasets

**network** This is a graph that represents a network of web servers.

**OMOP** This graph contains medical data with nodes such as *Person*, *DrugExposure*, and *CareSite*.

Similar to a traditional database, the information often has a specific structure. An example of this is shown in figure 1.1 which describes the labels and relationships present in the fraud dataset. Here, we see that an *AccountHolder* can be related to an *Address* as well as *BankAccounts* etc. However, an *AccountHolder* would never have a relationship to something unrelated, like an *IP*.

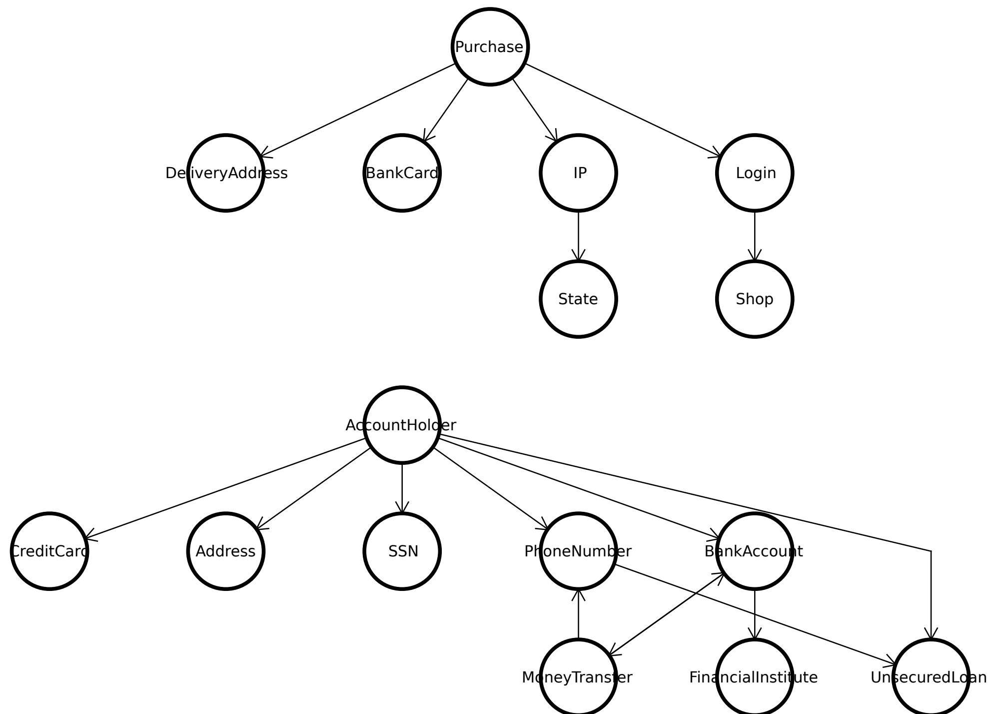



**Figure 1.1:** Labels found in fraud, and the relationships present between them.

### 1.3.3 Neo4j Browser

The Neo4j Browser is a tool developed by Neo4j. It is used to illustrate the data in their database. A variant of the force directed algorithm described in section 2.3 is used to draw the image. Figure 1.2 shows an example of what a user might see in the Neo4j browser. It is the result of the Cypher query `match x=(:Login {firstName:'Kazuko'})-[*]-() return x limit 20` being run on the fraud dataset.

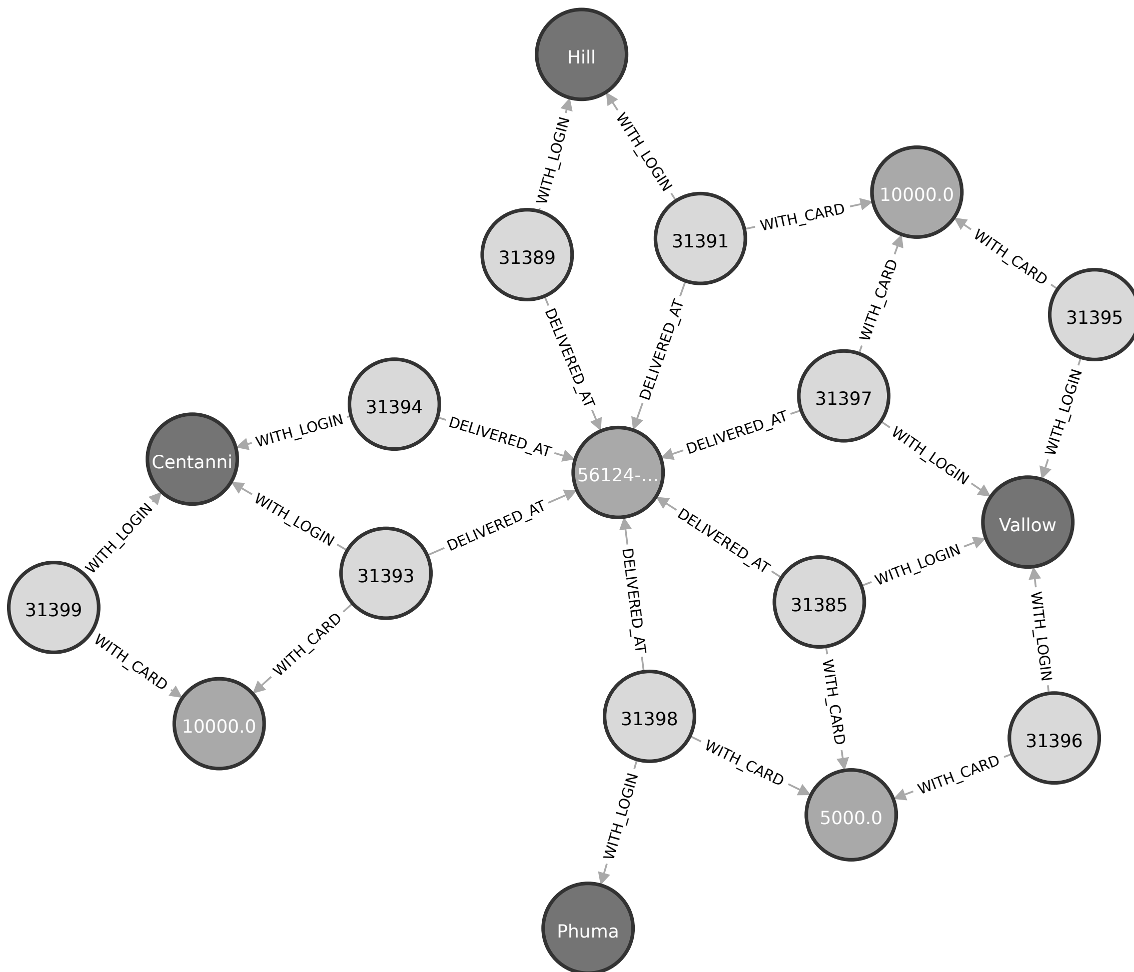



**Figure 1.2:** Neo4j Browser example

## 1.4 Use Cases and Expectations

The purpose of the Neo4j browser is to let a user visualize a part of the data stored in the database. Initially, the user enters a Cypher query and the browser shows an image of the graph. The user can then rearrange or hide nodes. Alternatively, the user might edit the Cypher query and generate a new image. The user can also select a node and show all adjacent nodes that were not already included from the original query. The tool is meant as an interactive editor, where the user responds to a new drawing by modifying the query or selecting which nodes that are shown, expecting a new drawing within a short time.

A commonly used limit for response time was introduced by Jakob Nielsen[17]. Within 0.1 seconds the system is perceived as responding immediately and response times longer than 1 second begin to interrupt the users flow of thought. Since the Neo4j browser is used interactively, render times larger than 1 second are undesirable. Actions such as expanding the graph by adding adjacent nodes should preferably seem instant to the user, and therefore have a response time within 0.1 seconds.

# Chapter 2
# Previous Work

A large amount of research has been done in the field of graph drawing. Different situations require different solutions, therefore many approaches have been developed to achieve specific desired results. A book called *Graph Drawing; Algorithms for the Visualization of Graphs* [19] has summarized the most commonly used approaches. These algorithms often serve as a basis for the graph drawing, which can then be tweaked for each specific case. The algorithms which were considered but not implemented are described below.

**Topology Shape Metrics** This three step process, consists of planarization, orthogonalization, and compaction. During planarization, the graph is modified so that it can be drawn in a plane without any crossing edges. The planar graph is then drawn orthogonal, that is, in a grid with horizontal and vertical edges only. During orthagonalization the number of edge bends is reduced. Finally, the compaction step reduces the size of the graph drawing as much as possible.

**Visibility method** Nodes are drawn as horizontal lines and edges as vertical lines. By constraining edges and nodes to this abstract form, crossing reduction becomes easier. In a final step, the lines representing nodes are replaced with circles as is common in other graph drawings.

**Augmentation approach** This approach also begins with planarization. Then the graph is augmented by finding cycles and inserting edges to create smaller triangular shapes. Finally, special properties of triangular graphs can be used to render an image as described by Kant Goossen [14].

**Divide and Conquer** methods work by dividing the graph into subgraphs recursively and reducing the problem to drawing single nodes and combining two already drawn subgraphs. A modified version of this approach, the community method, that was explored in this paper and described in section 3.4.

**Bundling** The bundling approach improves an existing drawing. Edges which are close to each other are combined into bundles, which reduces visual clutter.

**Constrained Drawing rules** This approach focuses on applying constraints to the graph elements. For example certain nodes should be placed left of others or the angle between two edges from a node should be larger than 30 degrees. These constraints are then taken into consideration whilst generating an image using the force directed approach. Weiqing He and Kim Marriott introduced the approach [12] and it has been implemented in the open source tool WebCola [22].

The following sections describes four approaches which where thought to be suitable for representing graph databases. These were all implemented and evaluated further.

## 2.1 Layered Method

The layered graph drawing is also known as hierarchical graph drawing or Sugiyama style graph drawing (named after its inventor) [18]. This layout algorithm inserts nodes by applying several constraints to the graph. Each node is put in a layer of nodes, where each node shares the same vertical position in the drawing. The general direction of the edges should be downwards, that is, from a node of a higher layer pointing to a node lower layer. Edges that only span between two adjacent layers are also preferable. These edges are called short edges, whilst edges which span over multiple layers are called long.

A layered graph can be altered to adhere to the above mentioned parameters. An edge in the wrong direction can be reversed, and an edge between two layers far apart can be replaced by a series of temporary edges and nodes (hereafter referred to as dummy nodes). Once the simplified graph is drawn, these alterations can be reversed to allow the final graph to represent the original data again.

The algorithm goes through four basic steps: cycle removal, layering, node ordering, and coordinate assignment.

### 2.1.1 Cycle Removal

A cycle in a graph is defined as a non empty sequence of edges, $\{e_1, e_2, \dots, e_n\}$ with a corresponding node sequence $\{n_1, n_2, \dots, n_1\}$, which contains the same node twice. For most layered like approaches, one initial and essential step is to remove all cycles from the graph. This is usually done by either temporarily reversing the direction of some of the edges, or removing them entirely. The appearance of the final result is heavily dependent on the amount of changes made to the graph, as each reversed edge will result in one edge directed against the flow of the other edges in the graph. For this reason, a solution with a small number of reversed edges is preferred. This problem is called the feedback arc set, which is described as finding the minimal set of edges in a cyclic graph that when removed produces an acyclic graph. The feedback arc set is an APX-hard problem.[10] This means that finding an optimal solution to the problem takes polynomial time, but there are algorithms which can find solutions which are some factor C worse than the optimal. This value C has been shown to be equal to 1.3606 by reductions origination from the vertex covering problem. [21]

### Fast & Effective Approximation Algorithm

A solution that reverses few edges is important to get a good looking graph, however, computational complexity needs to be low in order to get a result in an adequate time. Eades et al. [4] developed a very fast solution to the problem with good performance, achieving near optimal results in sparse graphs, which is the usual case when processing graph databases.

By ordering all the nodes of a graph in a sequence $s$, all cycles can be removed by reversing or removing all leftward edges $(n_j, n_i)$, $j > i$. The proposed algorithm uses this fact to simplify the problem. It orders all nodes in a sequence so that the leftward edges are as few as possible. After the sequence has been found and the leftward edges are identified they are reversed. The resulting sequence $s$ will have a set of leftward edges $R(s)$ such that $|R(s)| \leq e/2 - n/6$.

The algorithm used is described in algorithm 1. A source is defined as a node with no edges directed towards it. A sink as a node with no edges directed out of it. $\delta(n)$ denotes the number of edges from $n$, minus the number of edges to $n$.

**Algorithm 1** feedback arc set heuristics

```
1: procedure GR(G : DiGraph; var s : NodeSequence)
2:     s1 ← ∅, s2 ← ∅,
3:     while G ≠ ∅ do
4:         while G contains a sink do
5:             choose a sink n; s2 ← ns2; G ← G − n
6:         while G contains a source do
7:             choose a source n; s1 ← s1n; G ← G − n
8:         choose a node n for which δ(n) is a maximum;
9:         s1 ← s1n ; G ← G − n;
       return s ← s1s2
```

## 2.1.2 Layering

A layer is defined as a set of nodes from the graph $L = N_1, ..., N_n$, $N_1, ..., N_n \in G$. A set of layers such that $\bigcup_{i=1}^{h} L_i = N$ is called a layering.

The task of the layering is to assign each node in the graph to one of these layers. This is arguably the most important part of the layered graph drawing algorithm, as it greatly impacts the final result. Finding such a layering can be done in many different ways, and there are many criteria to take into consideration. Common approaches during this step is to attempt to find the best layering according to a specific measurement. Some popular alternatives for this layering step are described below.

**Minimum height** This algorithm produces a layering by attempting to minimize the height of the graph. The height is determined by the number of layers. A greedy approach for inserting nodes at the topmost layer works as follows. To avoid upward edges, a node can only be inserted if all neighbors with an edge to the node have already been inserted in previous layers. So each layer will contain all nodes where all in edges originate from the above layers. The first layer will contain nodes with no in edges, the second nodes

with only edges from the topmost layer, and so forth. This results in a very short but wide result, with few dummy nodes.

**Minimum width** By instead minimizing width of each layer, the computational complexity for crossing reduction can be made smaller. If fewer nodes exist in each layer, the ordering step could need to process few nodes. Minimizing the width is harder than minimizing the height however, as making one layer short results in additional dummy nodes for layers below.

**Fewest dummy nodes** This method minimizes the number of layers each edge spans across, making them as short as possible. Short edges produce few or no dummy nodes which is detrimental to the speed of other parts in the layered algorithm. Finding such a layering can be done using Integer Linear Programming.

**Coffman-Graham** Coffman-Graham is a widely used job scheduling algorithm. It can be also be used in the layering. It takes a maximum layer width as input, then assigns nodes to the fewest number of layers possible, adhering to constraints such as relationships pointing downwards.

### 2.1.3 Node Ordering

After a layering has been created, the nodes in each layer can be rearranged. This step usually aims to decrease the amount of crossing edges in order to improve readability of the graph. Depending on their order, the amount of crossing edges can vary greatly. For example, if the leftmost node on one layer is connected to the rightmost node on an incident layer, the edge will cross many edges. Placing both nodes to the very left removes all those crossings. Ordering the nodes in the layering to minimize the crossings is NP-Hard, in fact, minimizing the crossings between just two layers is NP-Hard [5]. Many heuristics are available for finding adequate solutions to the crossing problem. These heuristics usually work by iterating through each layer in the graph. The nodes in the previously processed layer are kept in a fixed position, while the order for the layer below it is calculated.

There are three algorithms most commonly used in this ordering step, which are described below. Each of these algorithms was also implemented and discussed in section 3.1.2.

**Median ordering** This method works by assigning a value to each node in the current layer. This value is calculated as the median of the positions of the neighbors in the above layer, which is being kept fixed. The position of the neighbors is simply defined as their order in the layer, the leftmost node has a position of 0, the one to its right has 1, and so forth. After this value has been calculated for each node in the current layer, the nodes are ordered by this value. Both this method and the mean ordering has been proven to always find an ordering with no crossings, if there is one possible [19]. In addition to this, the median ordering has been proven to produce a crossing which is at most three times the optimal[6].

**Mean Ordering** This works in the same manner as the median ordering, however, the average position instead of the median is being calculated as the value.

**Swap adjacent** This algorithms works by iterating through the unfixed layer, and swapping two adjacent nodes if it results in fewer crossings to the above layer. This will find a local minimum configuration of the crossings. However, it might get stuck in undesired positions.

### 2.1.4 Coordinate Assignment

The coordinate assignment step decides the horizontal coordinate of each node. The focus here is to have edges be as straight as possible, since straight edges are easier to follow. Furthermore, keeping the graphs' width as thin as possible is important since layered methods have a tendency to create wide drawings [8]. The Coordinate assignment step is often limited to avoid disrupting the order of nodes decided in previous steps.

An important metric whilst assigning coordinates is the number of bends on the edges. Since long edges are the hardest to follow, it is important to keep the bends on long edges to a minimum. This means that dummy nodes should be removed or placed vertically aligned, wherever possible.

#### Brandes and Köpf Method

The method presented by Ulrik Brandes and Boris Köpf is an efficient solution to the coordinate assignment problem. Their algorithm runs in $O(n)$ time, where n is the number of nodes, including all dummy nodes.

The algorithm aims to align each node vertically with its median neighbor on the layer above it. Edges between two dummy nodes are named inner edges and all other edges are named outer edges. Brandes and Köpf identify three types of conflicts between two edges. Conflicts occur when two edges cross due to the layer ordering. Both edges cannot be perfectly vertical. Thus, the algorithm needs to decide which of the edges becomes diagonal. Inner edges are prioritised in order to limit bends to the beginning and end of long edges.

- Type 2 conflicts occur when two inner edges cross. These conflicts could be changed into type 1 conflicts by swapping nodes. Earlier steps in the Layered approach should already avoid this type of conflict where possible. If such a swapping algorithm where to be implemented here, it would however change some of the node ordering performed in the previous steps. Since type 2 conflicts seldom occur, they are simply ignored in this implementation.

- Type 1 conflicts occur between an inner edge and an outer edge. They are always resolved in favor of the inner edge. This way, edges that span over multiple layers only bend in the beginning and the end wherever possible. These type of conflicts are found and marked in a pre-processing step so that they can easily be ignored later.

- Type 0 conflicts occur between two outer edges. These conflicts are resolved from left to right.

The alignment algorithm then selects which nodes should be aligned and creates blocks. A block is a linked lists of horizontally aligned nodes. During this step, layers are traversed from left to right. The topmost and leftmost node in the block is named the root of the block. Each node in the block gets assigned a position relative to the root block. The blocks are then

compacted in the horizontal compaction step, where each block is placed as far to the left as possible.

Since this method typically has a bias towards the left, as well as the upper neighbors, it is repeated four times and biased in different directions each additional time. Each node gets four horizontal coordinates: the top-left bias, the top-right bias, the bottom-left bias, and the bottom-right bias. Finally, an average median of the four results is used to determine the horizontal coordinate for each node. The average median is calculated by taking the average of the two median values. [1]

## 2.2 Spectral Method

Spectral Drawing Methods where initially introduced by Kenneth M. Hall in 1970 [15]. Hall presented a method using the spectral decomposition of the graph Laplacian to draw the graph. Matrices can be used to represent a system of equations. The eigenvectors then represent the solutions of this system of equations. By building the matrix as a system of equations with the node coordinates as variables, a solution to the equation gives coordinates for the nodes. The equations can be seen as constraints for the coordinates.

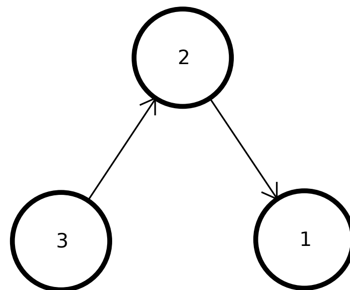



**Figure 2.1:** Spectral example

### 2.2.1 Matrix Representation

Hall used the graph Laplacian to represent the graph. The graph Laplacian is defined as the degree matrix minus the adjacency matrix. In other words, the Laplacian matrix $L$ has the elements described in equation 2.1 where $deg(v_i)$ is the number of edges adjacent to $v_i$.

$$L_{ij} = \begin{cases} deg(v_i), & \text{if } i = j \\ -1, & \text{if } i \neq j \text{and } v_i \text{is adjacent to } v_j \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

Civril, Magdon-Ismail and Bocek-Rivele describe an alternative spectral drawing technique which they call Spectral Distance Embedding [2]. They use a distance matrix where $D_{ij}$ = shortest path between $v_i$ and $v_j$ for $i, j = 1, 2, ...n$. Their approach tries to place nodes so that their Euclidean distance is close to their graph distance. This creates results similar to force based drawing techniques. Matrix 2.2 is the laplacian matrix for the graph shown in figure 2.1.

$$\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix} \tag{2.2}$$

## 2.2.2 Energy Minimization

Hall shows how the Laplacian correlates to an energy function shown in 2.3. The Euclidean distance of each node pair is multiplied by the weight of their corresponding edge $w_{ij}$. Since the relationships in the datasets provided for this thesis where not weighted, the weights were all set to one. The length of edges is minimized, which leads to nodes being placed close to their graph neighbors. In the equation, $X$ is a matrix of size $N \times M$ where $M$ is the amount of eigenvectors or dimensions.

$$E = X^T L X = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij}(x_i - x_j)^2 \tag{2.3}$$

## 2.2.3 Spectral Decomposition

Finding the eigenvectors and eigenvalues of a matrix is a well-researched topic. The power iteration method which Civril et. al. present was implemented in order to find the largest two eigenvectors. For the Laplacian matrix, the Efficient Java Matrix Library (EJML) [9] was used to find the smallest two eigenvectors.

## 2.2.4 Selecting Vectors

The eigenvalues and corresponding eigenvectors are ordered so that $0 = |\lambda_0| \leq |\lambda_1| \leq ... \leq |\lambda_n|$. The Laplacian for the graph in figure 2.1 has the three eigenvalues $\lambda_0 = 0$, $\lambda_1 = 3$, and $\lambda_2 = 3$. The eigenvectors corresponding to those eigenvalues are: $v_0 = [1, 1, 1]$, $v_1 = [1, 0, -1]$ and $v_2 = [1, -2, 1]$.

The least eigenvalue will always be zero and correspond to the vector $(z, z, z)$ which is not useful for drawing the graph. Each other vector can be seen as a dimension in which to draw the graph. Hall uses the vectors corresponding to the smallest eigenvalues except for the zero vector. Civril et al. use the largest two eigenvalues instead. As can be seen in figure 2.1, node **1** gets the horizontal coordinate 1 from the **first** element of $v_1$ and the **first** element of $v_2$ specifies the vertical coordinate. The same is repeated for node 2 and 3. The vectors are normalized and scaled based on the size of the nodes.

## 2.3 Force Directed Method

The force directed graph drawing approach was initially introduced by Peter Eades in 1984. Each node is assigned a position. Then the position is incrementally improved based on a force calculation in the graph. Eades presents an attractive force between adjacent nodes and a repulsive force between all nodes [3].

Fruchterman and Reigold present improved calculations for these forces [11]. The forces are based on the distance between two nodes $d$ and a user specified optimal distance $k$. For each pair of nodes, the two forces are calculated as follows.

$$f_r(d) = \frac{-k^2}{d} \tag{2.4}$$

$$f_a(d) = \frac{d^2}{k} \tag{2.5}$$

Yifan Hu presents a method for improving the speed at which the model converges. The adaptive cooling scheme shown in algorithm 2 performs larger or smaller steps depending on how much the graph is improving [13].

**Algorithm 2** Adaptive Cooling Scheme

1: **procedure** FORCE_DIRECTED((V,E) : Graph)
2: converge=false, step = initial_step
3: **while** converge = false **do**
4: $x_0 \leftarrow x$
5: engergy$_0$ $\leftarrow$ engergy,
6: **for** i ∈ V **do**
7: $f \leftarrow 0$
8: **for** (i,j) ∈ E and (j,i) ∈ E **do** $f \leftarrow f + \frac{f_a}{\|x_j - x_i\|}(x_j - x_i)$
9: **for** j ∈ V where j ≠ i **do** $f \leftarrow f + \frac{f_r}{\|x_j - x_i\|}(x_j - x_i)$
10: $x_i \leftarrow x_i + \text{step} * (\frac{f}{\|f\|})$
11: energy $\leftarrow$ energy + $f^2$
12: step $\leftarrow$ update_steplength(step energy, energy$_0$
13: **if** $\|x_0 - x\| < K_{tol}$ **then** converge = true
14: **procedure** UPDATE_STEPLENGTH(step: float, energy:float, energy$_0$: float)
15: **if** energy < energy$_0$ **then**
16: progress $\leftarrow$ progress + 1
17: **if** progress $\geq 5$ **then**
18: progress $\leftarrow$ 0
19: step $\leftarrow \frac{\text{step}}{t}$
20: **else**
21: progress $\leftarrow$ 0
22: step $\leftarrow t$step

## 2.4 Community Method

The Neo4j open source community has implemented a number of graph algorithms to be used with Neo4j. Among these three algorithms for community detection, namely Louvain, Label Propagation, and Weakly Connected Components. These algorithms inspired the development of the Community Drawing Method, that is described in detail in section 3.4.

### 2.4.1 Louvain

Modularity is a way of measuring how well a community of nodes fits together. A community densely packed with edges between nodes in the community, and few edges to other nodes, will receive a high modularity score. The Louvain algorithm uses greedy heuristics to try and maximize the total modularity score of the graph. Every node will start off in its own community at first. Then, each node will attempt to move into the community of each of its neighbors. The node will stay in the community of its neighbors with the highest increase in the graph's modularity. If no such increase can be found, it will stay in its original community. This step is repeated until no more nodes can be moved. After this step, each community is merged into one big node, and all edges represented as self edges or edges to the other community nodes. The first step can then be re-attempted to see if any communities should be merged into bigger ones.

The modularity compares the fraction of edges within the communities, with what the fraction would be if the edges were randomly distributed. A community with a higher number of edges within the community, than that of a random distribution, will have a positive modularity. The change in modularity within a community, when a given node moves into given community, is calculated as follows:

$$\Delta Q = \left[ \frac{\sum_{in} + 2k_{i,in}}{2m} - \left(\frac{\sum_{tot} + k_i}{2m}\right)^2 \right] - \left[ \frac{\sum_{in}}{2m} - \left(\frac{\sum_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2 \right] \tag{2.6}$$

$i$ the given node, moving into the community.

$m$ The number of edges in the graph.

$\sum_{in}$ sum weight of edges within the community.

$\sum_{tot}$ sum weight of edges originating from nodes in the community.

$k_{i,in}$ sum weight of edges between $i$ and the community.

$k_i$ sum weight of edges originating from node $i$.

This change in modularity also has to be calculated for the removal of $i$ from its original community.

# Chapter 3
# Implementation

The software written to evaluate the different algorithms consist of 4 main parts. First there is a `DBHandler` class, that sends a cypher query to the neo4j server and decodes the answer into a graph object to be handled by the algorithms.

Second there are the different graph algorithms. These take a graph object and, through different algorithms, assign each node in the graph with a position in the image.

Third `GraphRenderer` takes the graph object and renders an image file. Compared to neo4j bloom and the neo4j browser, these images are much simpler. The nodes and edges are drawn in black and white, and each node is labled with its ID and label. Images generated this way can be compared in appendix B.

Finally the main program, `GraphDraw`, connects the previous parts together. This program was later extended to run the experiments as described in chapter 4.

## 3.1 Layered Method

The layered method was developed to operate in a modular way, with different configurations of algorithms to choose from. This was achieved using the command pattern. An abstract class was created for each step in the algorithm, for example a "Layering algorithm" to solve the layering step. Each implemented algorithm was then made to extend its corresponding abstract class, so that they could easily be interchanged. The abstract classes included a command to perform its specific purpose, which was overridden by the specific algorithms. For example, the minimum height algorithm described in section 2.1.2 was implemented. This extends the abstract interface "Layering Algorithm", providing a layering when "getLayerAssignment" is called. The layered method is therefore constructed using parameters of these three abstract algorithms. For example, with the parameters: (new LongestPathLayering(), new MeanOrdering(), new CoordinateAssignerBrandesKopf()) the algorithm will run using these specified methods.

### Cycle Removal

The heuristic method described in section 2.1.1 was implemented and used in the label layering algorithm described in section 3.1.1. The cycle removal is usually used as the first step in the layered algorithm. In practice, different layering algorithms suggest different cycle removal techniques. The heuristic was only performed on the set of labels in the graph and never performed on the entire set of nodes. Since the predefined layering will follow the specified layering, it can simply skip the cycle removal phase and remove all upward edges. The query layering, however, does not take the direction of edges into consideration, as it aims to show a layering similar to the query written. If the query represents edges in a reverse order, such as `()<-()`, the edges in the result should also represent this.

To keep edges pointing downwards, and to remove cycles, the determined edges are reversed temporarily by editing the inward, and outward edges of both related nodes. The edge is also marked with a boolean value, to show that it has been reversed. For the final image, all revered edges need to go back to their original orientation. This will be easy when each edge is marked. During implementation, using lists for sinks and sources proved useful, as finding an arbitrary sink or source often means iterating over much of the graph. Only the nodes connected to any removed edges will need to be checked if they become a sink or source. Another important aspect of this implementation is that it will remove all nodes and their edges from the graph it is working on. Lists of removed edges and nodes can be kept to avoid creating a deep copy of the graph.

## 3.1.1 Layering Algorithms

The task for the layering algorithms is to assign every node in the graph to a layer. An abstract class for these algorithms was created with an abstract method for finding a layering. This method is implemented by all layering algorithms, and returns a list of layers represented by lists of nodes. All edges between two layers need to be directed from the above layer to the one below. The algorithms must therefore either find a solution, or temporarily reverse the edges in the wrong direction. The algorithm described in section 2.1.1 can be used to help achieve this.

### Label Layering

The databases often contain the same types of relationships. This is shown in Figure 1.1 where the relationships between different labels are presented. In this data set, each of these edges represent a multitude of relationships. We can see that the relationships present are almost always the same, a *Creditcard* is always related to an *Accountholder*, but never anything else. Many data sets will have these kind of connections present, the Label Layering algorithm was designed to portray this information to the user.

In databases with these well established relationships, a layering can be created so that each layer contains only nodes with the same label. This layering method will present the nodes in a structure that shows the user what types of relationships are found. Edges originating from one layer will therefore only point to one, or few other other layers, instead of many different ones. It will also create a representation of the structure of the database. The general direction of the edges is downwards, therefore as long as the layers are in a proper

order, a clear hierarchy can be created. To draw a graph for a dataset such as the one in described figure 1.1, we can put all of the nodes in the layer with the corresponding label. In this case, the *AccountHolders* at the top and their related *Addresses*, *SSNs*, etc. below them. In fact, the figure 1.1 was actually generated using this algorithm.

### Removing Label Cycles

Similar to how an acyclic graph is needed in most layering algorithms, this approach needs the connections between all labels to be acyclic. Any cycle between the labels would mean that at least one of the labels will have edges directed towards a higher level layer. An example of a label cycle that does not occur in an acyclic graph can be seen in figure 1.1. Some *Moneytransfer* nodes have a relationship to a *BankAccount* and vice versa. As both of these relationships cannot point downwards, one of them needs to be reversed. A label graph can be created where each node represents one layer of nodes, and each edge represents all connections between two layers. By utilizing the same heuristic approach described in 2.1.1 on the label graph, the label graph can be made acyclic by reversing a small number of edges. It is worth noting that this will yield a low number of label edges to reverse, which will result in few edges directed upwards.

### Finding a Layering

Each layering can then be scored by calculating the amount of dummy nodes that would be inserted. This is easily achieved by multiplying the number of connections between the layers, and the distance between the layers. $Sum(La) = \sum_{i=1}^{h} \sum_{j=i+1}^{h} (j-i-1) * \sum E(i,j)$. Two different methods were implemented for finding a layering. The first method implemented is a naive approach that effectively attempts to score all acceptable permutations. As an alternative to this, a faster heuristics that attempts random configurations which are then modified to find local minimal solutions was also created.

### Finding All Permutations

The number of permutations grows as a factorial of the number of layers. One way to limit this number is to only allow permutations for which all edges are pointing to a lower layer. Such a permutation surely exists as long as no cycles are present in the label graph. By iterating over all edges in the label graph, each layer L can be assigned a set of layers R(L) with an edge to L. These layers would have to be placed above L, and therefore assigned to an earlier position in the layering.

All allowed permutations can be found by gradually building a layering.

Each time multiple labels can be inserted, a new layering is created for each possible layer. This will recursively build each possible permutation, as every possible outcome is chosen once. To clarify: at the first step, only labels for which $R(L_1) = \emptyset$ can be inserted. The second layer $L_2$ can only have the label of the first layer as a requirement: $R(L_2) \subseteq \{L_1\}$. Layer three can only have layer one, layer two, or both: $R(L_3) \subseteq \{L_1, L_2\}$ and so forth. The proposed algorithm is described in algorithm 3.

**Algorithm 3** layering permutations

```
 1: procedure LP(noReq : Set of L; La : Layering; Al : Set of all layerings)
 2:     if noReq = ∅ then
 3:         Al ← Al+La
 4:     for L ∈ noReq do
 5:         copyLa ← La+L
 6:         copyNoReq ← noReq - L
 7:         for L2, where L2 has an edge to L do
 8:             if R(L2) ⊆ copyLa then
 9:                 copyNoReq ← copyNoReq+L2
10:         LP(copyNoReq, copyLa, Al)
```

### Randomized Heuristics

Algorithm 3 can easily be modified to produce only one randomized permutation by choosing an arbitrary layer L from noReq to add, instead of branching on each alterative. If each layer is chosen in a uniformly random manner, multiple layerings will produce a uniform distribution of the possible layouts. Each layout can then be repeatedly altered to find a local minimal score. Two incident layers can be swapped if the resulting layering is both valid and has a lower score than the original. The number of layers is very small, therefore these alterations can be quickly made until no more changes are possible.

## Predefined Layering

This layering is a specific variant of label layering. Each layer still consists of the nodes with a specific label. However, instead of finding the best layer ordering, the user instead specifies which order they want. This way there could be a large amount of edges that need to be reversed. However the user might use context which is not stored in the graph to order the layers and thus create a better drawing for their specific needs.

## Query Result Layering

The structure of a Cypher query proved useful for layering step. Many queries return series of nodes connected by edges, referred to as paths. The edge direction of the path might be defined but does not have to be. For example, the query `match p=(:Shop)-(:Login)-[*2]-() return p limit 50` finds 50 paths starting from a node labeled *Shop*, via a *Login* and then 2 further steps over nodes with any label.

A natural approach here is to assign layers to nodes depending on how far into the path they are. This way, the login nodes will be in the layer below the shop. A node which is two steps away from the shop is placed in the layer below that, and so on. Figure 3.1 shows the result of applying this layering on the query `match x=(start:Login)-[*2..4]-(end:Login) where ID(start)=31683 and ID(end) =31681 return x limit 10`. The query finds 10 paths between the Login nodes with ID 31683 and ID 31681. `[*2..4]` specifies that the paths should be between 2 and 4 steps

long. A path of length 2 is found on the right where the logins share a relation to a shop. On the left, the two logins are related to purchases that share IP and delivery address.

The crossing reduction step, performed after layering, naturally places related purchases close to each other and unrelated purchases apart.

This layering algorithm builds a predecessor tree where each node references a list of neighbors which precede it in the path. Login node 31681 at the bottom of figure 3.1 has the predecessors Purchase 31670, Purchase 31660, and Shop 30777. This tree is then traversed recursively. When all predecessors of a node have been assigned to a layer, the node is assigned to the lowest possible layer. The predecessors of Login 31681 are assigned the layers 3, 3, and 1, therefore Login 31681 is assigned to layer 4.

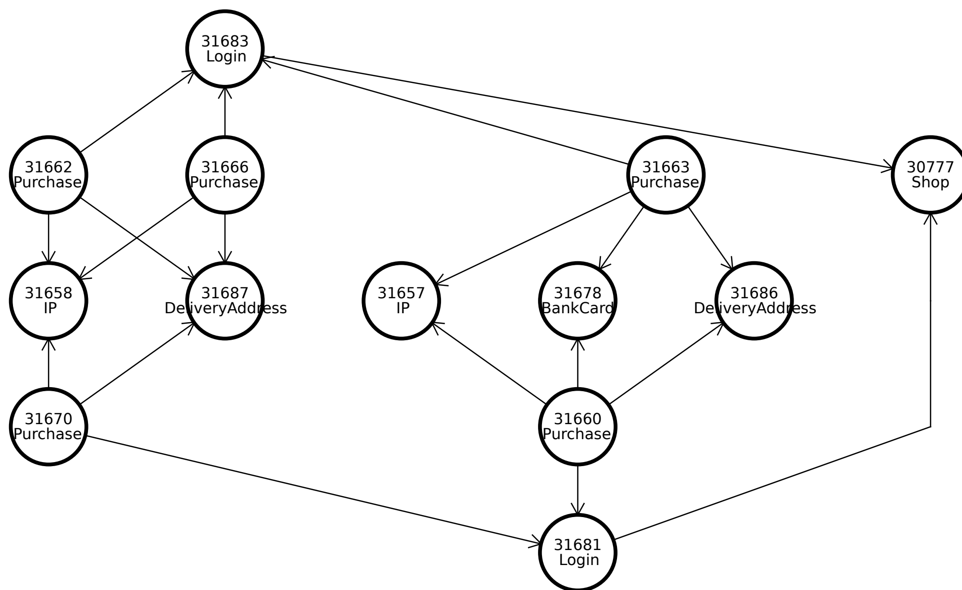



**Figure 3.1:** Query result layering example

### 3.1.2 Node Ordering Algorithms

The node ordering algorithm receives the layering created by the layering algorithm, and reorders the nodes in each layer. The mean ordering, median ordering, and swap adjacent algorithms described in section 2.1.3 were implemented. The mean and median algorithms were implemented by iterating through each layer of the input, starting with the second layer, and calculating either the mean or median value of the above neighbors for each node. The nodes in the layer were then ordered by this associated value. The first layer was left in an arbitrary order as there are no neighbors to a higher layer. The swap neighbors algorithm also begins the iteration on the second layer. The edge crossings between each two adjacent nodes are considered. If swapping the two nodes decreases the number of crossings to the layer above, the nodes change places. As long as at least one set of nodes are swapped, the layer is iterated through again. This iteration repeated until no decrease in crossings is possible.

The crossings is calculated by saving the positions of the neighbors to the above layer for each of the two nodes considered. The indexes for each neighbor to the above layer from the two nodes are saved as two separate lists. The number of crossings can then be calculated by considering each pair of indexes from the two lists. If an index from the left node is *higher* than an index from the right node, a crossing is present. For example, the graph in figure 3.2 would have the lists [2,4] and [1,3], the algorithm will find a crossing on the edge between the left node and node 2, and two crossings between the left node and node 4. The number of crossings after swapping would instead be the number of indexes in the left list that are *lower*. As the number of indexes that are lower *or* higher is equal to the product of the number of edges in the lists, the number of crossings only has to be calculated once. Swapping the nodes is therefore only desirable if the number of crossings is greater than half of the product of the sizes of the lists. In the example, three crossings are present. That is greater than half of the product 4/2, so the swapping is performed. After the swap the number of crossings is reduced to one (calculated by 4-3=1).

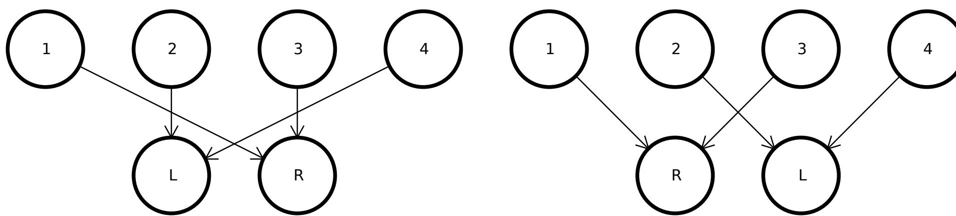



**Figure 3.2:** Before and after swapping the neighbors L and R

In order to analyze the data, two additional crossing algorithms were implemented. The *No ordering* algorithm leaves the layers in their arbitrary order, so that the improvement of using an ordering algorithm can be measured. The *mean and swap ordering* algorithm was also created which first performs the mean ordering, then attempts the swap algorithm to potentially decrease the crossings further.

### 3.1.3 Coordinate Assignment Algorithms

The coordinate assignment algorithms are responsible for assigning an X coordinate for each of the nodes in the graph, including the dummy nodes. The Y coordinate is decided by the layer number of each node, and can therefore be assigned at any point after the layering step. The X coordinate assignment should preferably not change the order of the nodes in the previous step, as this could result in an increase in crossings. But as can be seen later in this section, there are efficient algorithms that break this rule. Assigning coordinates without altering the order of the nodes has few measurable impacts on the final result. However, they can still be very meaningful to the user. Minimizing the number of bends in the edges can reduce clutter in the final result. It is also a good idea to keep edges in strict vertical lines, as it is easier to follow parallel lines.

### Brandes and Köpf

The algorithm described in section 2.1.4 was implemented following the pseudo-code in the paper by Brandes and Köpf [1]. HashMaps were used to store predecessor, root, sink, shift, and align variables to give $O(1)$ access to them. Problems appeared later with the recursive function `place_block`. The function performs a depth first walk of the graph visiting every node, including the dummy nodes. This quickly lead to stack overflow errors. The jvm arguments `-Xmxs` and `-Xms` were used to increase the memory to 8GB and help mitigate the problem.

### Mean Coordinate Assignment

The Brandes and Köpf algorithm of assigning coordinates can create very good results, however, the memory complexity can become troublesome with larger graphs. As an alternative mean coordinate assignment was implemented as a way of exploring whether a simple and faster approach could present acceptable results. The goal of the mean coordinate assignment is to position each node close to the mean X coordinate of the neighbors above it. Nodes are placed as close to the mean as is possible, whilst not being placed too close to previously placed nodes. Dummy nodes only have one in edge, and one out edge, so these are positioned first. This will keep the edges as straight as possible while minimizing the risk of occupying the space of another node.

After the dummy nodes have been placed, the nodes are processed in order of number of input edges. This priority was achieved by sorting the layer using a comparator. In order to locate the closest valid position, a list of all assigned positions in this layer can be kept. Any new assignment then iterates through the list to find its best position, and appends the list.

It is important to note that using this assignment will not take the order of the nodes into consideration, meaning that no node ordering will have any impact.

# 3.2 Spectral Method

## 3.2.1 Matrix Representation

Both Hall's Laplacian method as well as Distance Embedding, described in 2.2, were implemented. The implementations use a `DMatrixSparseCSC` object from the EJML library to store the matrix representations. EJML was chosen to provide robust and efficient matrix arithmetic algorithms. Sparse graphs lead to the matrix consisting of mostly zeroes. Storing $n \times n$ values for $n$ nodes quickly requires a large amount of memory. An efficient solution is to use the `DMatrixSparseCSC` from EJML, which only stores non-zero values and their locations in the matrix. This solution comes at a performance drawback however, as element access becomes somewhat slower.

## 3.2.2 Eigenvector Solving

Two algorithms where used for finding the Eigenvectors of a matrix. First, the Power Method was used. This method, described by Civril et al. [2], finds the eigenvectors corresponding to

the largest two eigenvalues.

Thereafter, a version using the algorithms from EJML was implemented. EJML provides eigenvector decomposition methods for their `DMatrixRMaj` matrix classes. EJML does not provide eigenvector decomposition functionality for their sparse data structures however. In order to use the algorithms, the `DMatrixSparseCSC` was converted to a `DMatrixRMaj`, which is a dense storage option. Thereby limiting the maximum amount of nodes able to be stored and processed.

## 3.3 Force Directed Method

The force directed approach was implemented based on the adaptive cooling algorithm by Yifan Hu presented in section 2.3. Each force calculation was done separately for the x and y coordinates to simplify the problem.

The force calculations created a few large values which lead to errors in further calculations. This issue was mitigated by limiting the updated position of each node to the left of its rightmost neighbor, below its uppermost neighbor and so on. In order to allow the graph to grow in size when necessary, each node was allowed to break the previous rule as long as it stayed close to its original position.

## 3.4 Community Method

The Community Drawing Method was inspired by existing algorithms mentioned in section 2.4.1. Community detection algorithms distribute the nodes of a graph into smaller groups, each containing nodes related to each other in some way.

The community method in this paper separates the graph into smaller subsets using one of three approaches. The first approach uses the Louvain algorithm as it is implemented by the Neo4j community. A second query is issued to the Neo4j server. This divides the graph and returns which community each node belonged to. A problem with this approach was that the server performs the Louvain calculation on the entire database, even when only a subset should be rendered. Thereby often placing every visible node into the same subset.

A second approach performs a random walks algorithms and assigns groups based on how reachable a node is. The algorithm performs ten random walks from each node, walking three steps each time. If a node is encountered five times starting from another, they are put into the same community. The choice of numbers was made arbitrarily and does not work for larger graphs. The random walk algorithm serve as an example of bad community assignment, however, they can be used to compare with a better solution.

Lastly, a custom implementation of the Louvain algorithm was used. This algorithm only analyzed the part of the graph that was returned from the Cypher query. Furthermore, it was performed within the program and did not require accessing the server. The Louvain algorithm itself is described in section 2.4.1.

After the division step, each subgraph is drawn on its own by using any other drawing algorithm presented in this paper. A graph is created with a node for each subgraph, and edges for relations between these. This graph is also drawn using any drawing algorithm.

In a last step, each node is placed in the original graph based on the position of its community node, and its position in that community.

# Chapter 4
# Results

To evaluate the performance of the drawing methods, the processing time for this step was recorded. Time was measured by storing `System.currentTimeMillis()` before calling the drawing method and comparing it to `System.currentTimeMillis()` after the drawing method had finished. The measured time thus neither includes fetching data from the server, nor the time to creating an image from the data.

The time required to receive the data from the database was not included in the measurements due to two reasons. Firstly one of the chosen queries was very slow to run in neo4j, which would skew the resulting measurements. Secondly neo4j applies a series of optimizations, including caching, which greatly affect performance between different runs of the same Cypher query.

Since the render methods performance scales linearly with the number of nodes and edges, the impact on the total time would be minimal. In a real world system the rendering step might be more complicated, however the impact of this was deemed to be out of scope.

By measuring time using the system clock, the presented results depend on hardware and system load. The neo4j server was running on the same machine, however the impact of this was mitigated in that the drawing method did not start execution before the server had returned the entire query result. Furthermore care was taken to keep the system load equal throughout the tests, by not using the PC for different tasks whilst running the benchmarks. Each query/drawing method combination was ran 10 times in order to mitigate any errors and the results presented in this paper are the average values of those 10 experiments.

The experiments were run on a desktop computer running Linux Mint 18.2. The computer uses a Intel Core i5-4690K Processor and has 16GB of RAM. The processor is specified to run at 3.5GHz with a turbo frequency of 3.9GHz, during the benchmarks the system was observed running between 3.7GHz and 3.8GHz without encountering thermal throttling. The program is run with OpenJDK 1.8.0_222. The JVM arguments `-enableassertions`, `-Xmx=8G`, and `-Xms=8G` were specified. Both `-Xmx=8G`, and `-Xms=8G` specify for Java to use 8GB of ram memory for the heap and initialize the heap with 8GB of ram. 8GB were chosen

so that the Neo4j server, which is hosted on the same machine, can use the remaining memory. Neo4j server community version 3.5.12 was used, since it was the newest version at the start of the project.

Images generated with each drawing method are presented in appendix B.

### Query Choice

The Cypher query, that is chosen as input to the program, defines the graph that should be drawn. Different queries will result in different graphs returned from the graph database. Two objectives where set for the queries. Firstly they should represent any structures that is common in graph databases. A query that does not include any edges might result in a very fast drawing, but this is quite uncommon when working with databases. Secondly the query should be scaleable. Each drawing method was tested with different sizes of graphs, so that the results could be compared to the theoretical complexity of the methods.

Two different Cypher queries were used to benchmark the different methods. The query `match e=()-() return e limit X`, hereafter referred to as the edge query, simply finds all edges in the graph. With the limit command, the number of edges returned is limited to at most X matches. By using different values for X the edge query satisfies the scaleability objective.

The edge query does represent the structure of the underlying graph well, in that it randomly picks edges and connected nodes evenly from the dataset. The performance results are misleading however which is discussed in section 5.6.

The query `match p=(:Login firstName:"Kazuko")-[*1..X]-() return p` finds all paths in the graph which start at the specific node that has firstName Kazuko and the label Login. This query will hereafter be referred to as the Kazuko query. The Kazuko query gives a better representation of how neo4j queries are used compared to the edge query. A common user scenario is to select an interesting node and expand the graph to include neighbors of that node, followed by the neighbors of neighbors and so on. This query guarantees a connected graph, which avoids some of the issues discussed in section 5.6. Since the value of X can still be changed this query also satisfies the scaleability objective.

## 4.1 Layered Method

In order to evaluate the different algorithms for each step in the layered method, each step was evaluated separately. This way, the speed of each algorithm could be compared. A specific algorithm was chosen for each step as the default. Each tested algorithm was then used with the default algorithm for the other steps not currently being evaluated. The default algorithms were chosen as the most typical algorithms to be run, and were marked by the solid line in each figure. For the layering assignment, both the **label layering** and the **query path layering** were evaluated, as both are suitable and their performances vary greatly. The **median ordering** algorithm was chosen as ordering and the **Brandes & Köpf** algorithm was chosen as coordinate assigner. The performance of the algorithms in each step could impact the performance of the other steps. Consequently, the total computation time was included rather than the time for each specific step. For example, if a layering algorithm decreases the computation time for the ordering step, this information will be represented.

## Layering

Figure 4.1 presents the running time of different layering algorithms. The predefined layering performance naturally varies based on the label order chosen. To represent this variance, two different label orders were tested: one using the alphabetical order of the layers and another with a random order. For the label layer algorithm, a layering was found using the randomized heuristics with 200 attempts.

In figure 4.2, the amount of created dummy nodes is presented. This figure can serve as a performance indicator for the layering algorithms since fewer dummy nodes are often preferable. In this chart, the predefined layering algorithms achieved a very similar result to the label layering and were therefore represented with a single line.

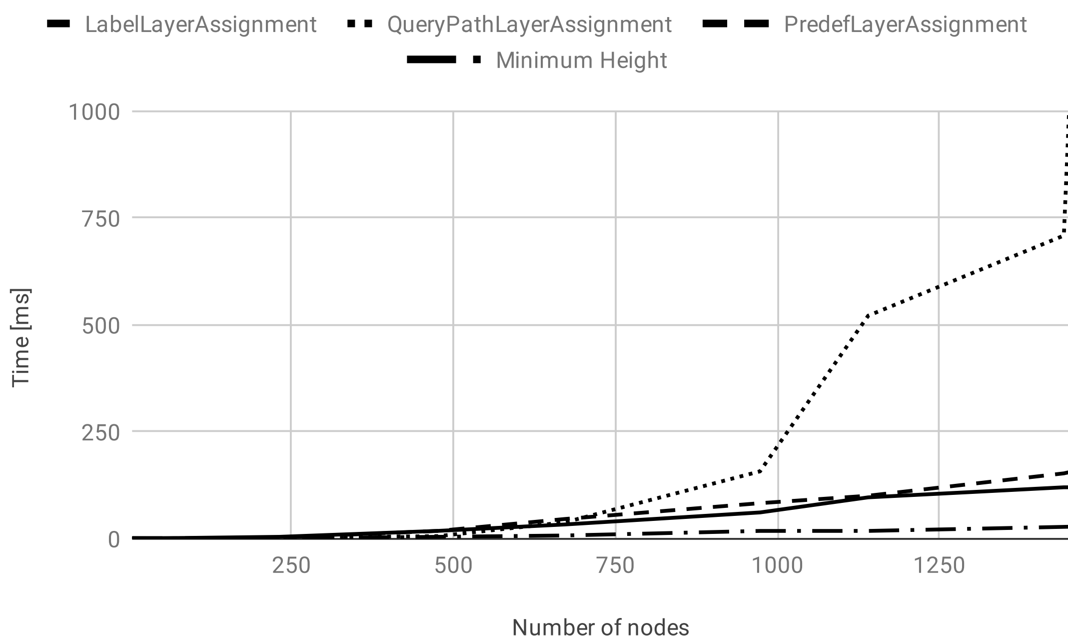



**Figure 4.1:** Computation time for different layering algorithms

## Crossing Reduction

In figure 4.3 and 4.4, the computation time of different crossing reduction algorithms are presented.

The number of crossings in each ordering algorithm are presented in figure 4.5 and 4.6. These figures where created using the same dataset and query as the timing figure. The figures present to what degree each ordering algorithm is able to decrease the total number of crossings between edges present in the graph. Fewer crossings means that the resulting graphs are less cluttered and easier to interpret.

In some of these tests, the mean and median ordering achieved near identical results, and are therefore represented as a single line.

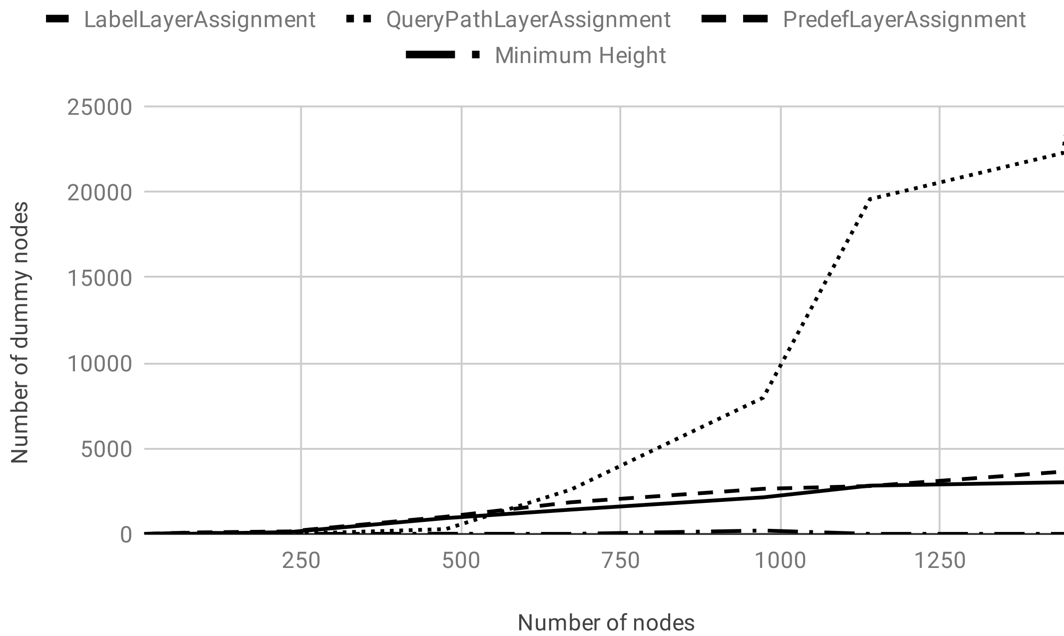



**Figure 4.2:** Number of dummy nodes for different layering algorithms

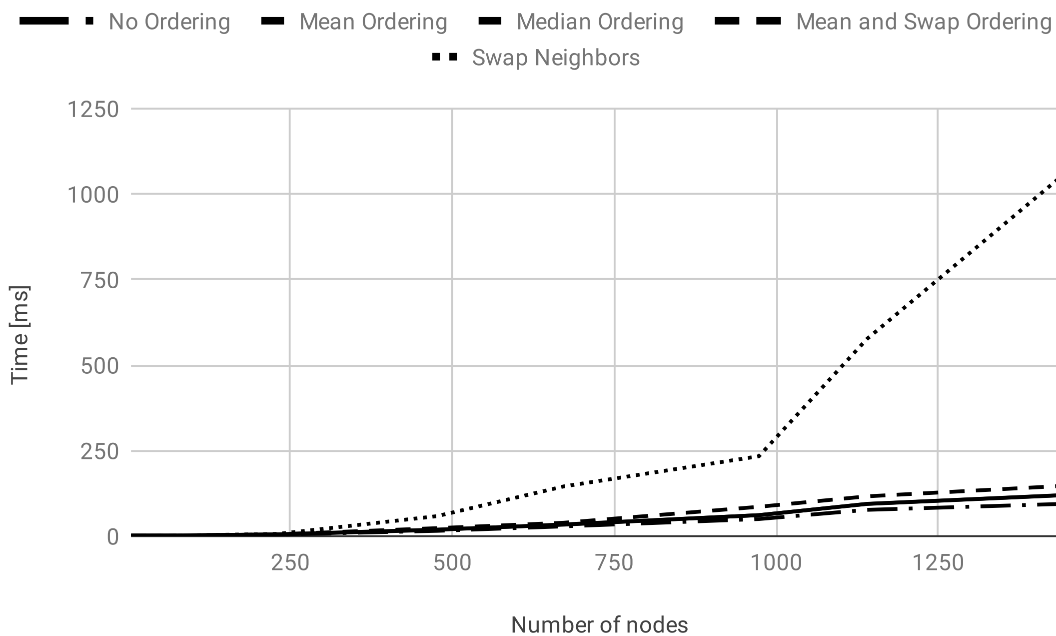



**Figure 4.3:** Computation time for different node ordering algorithms using the **label layering**.

## Coordinate Assignment

The performance of the three coordinate assignment algorithms is presented in figures 4.7 and 4.8. The performance is shown relative to the number of nodes returned from the query.

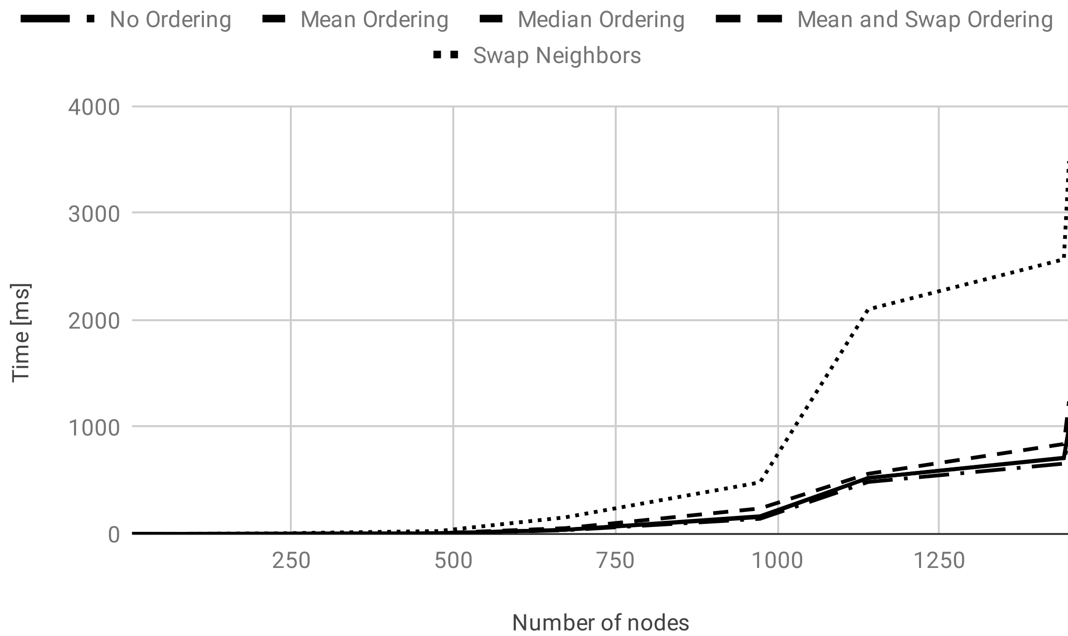



**Figure 4.4:** Computation time for different node ordering algorithms using the **query path layering**.

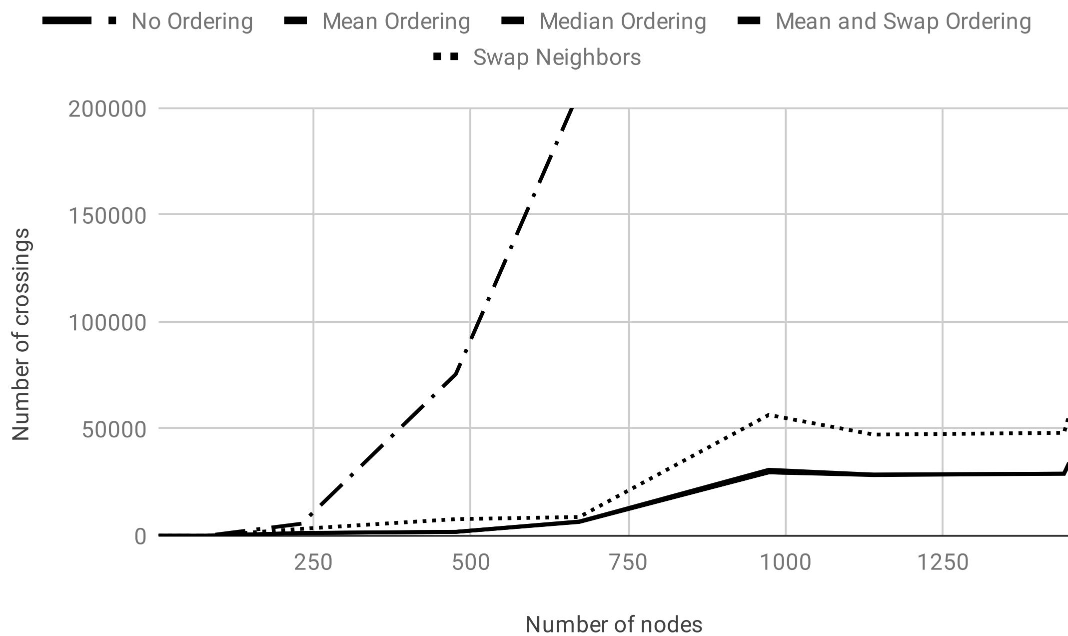



**Figure 4.5:** Number of crossings for different node ordering algorithms using the **label layering**.

The actual number of nodes processed, however, also includes the created dummy nodes which are shown in figure 4.2.

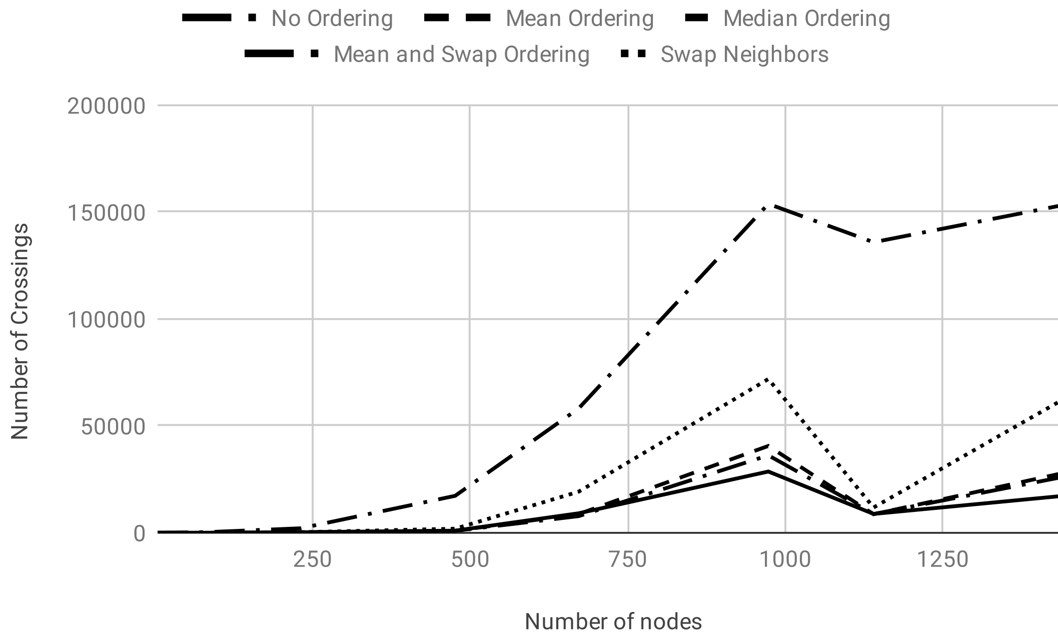



**Figure 4.6:** Number of crossings for different node ordering algorithms using the **query path layering**.

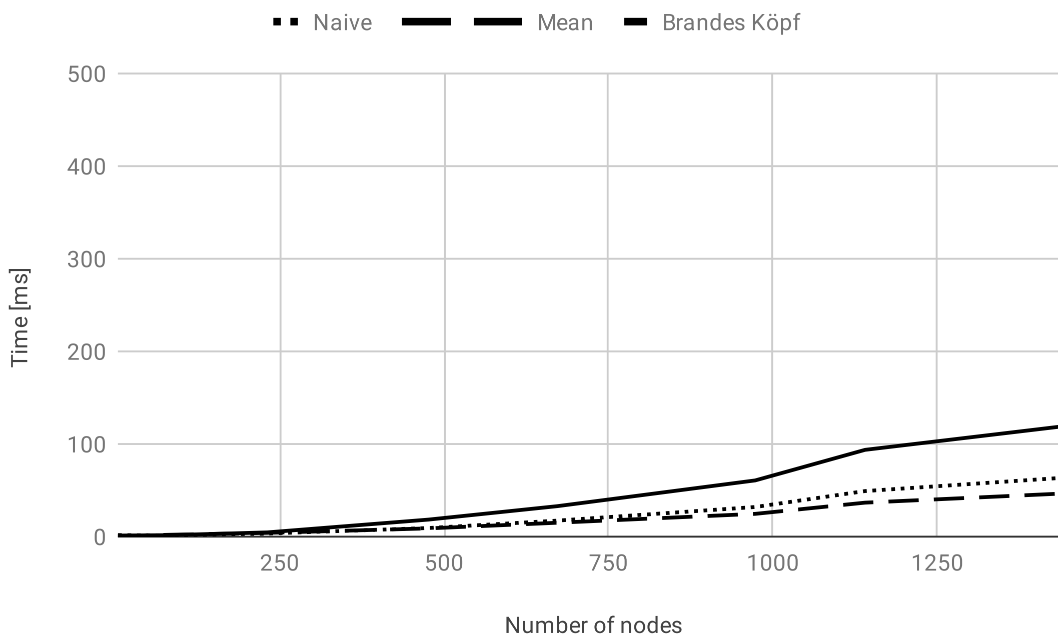



**Figure 4.7:** Computation time for different coordinate assignment algorithms using the **label layering**

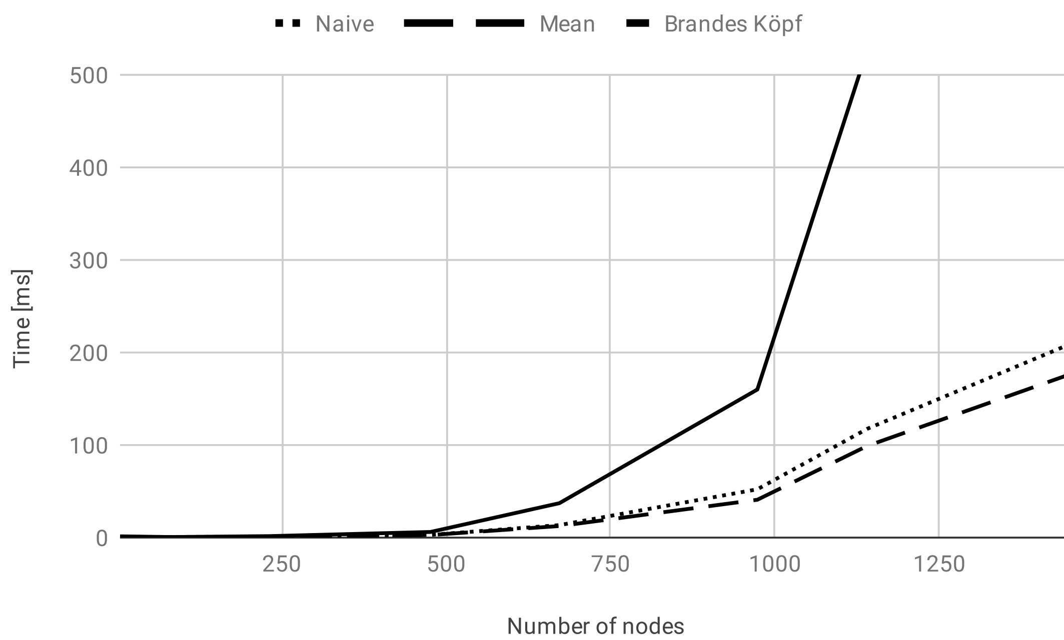



**Figure 4.8:** Computation time for different coordinate assignment algorithms using the **query path layering**

## 4.2 Spectral Method

The spectral algorithms were benchmarked like the previous methods, using the Kazuko query. The results are presented in figure 4.9. Note, however, that the result for spectral distance embedding is most likely erroneous, as described in section 5.2.1.

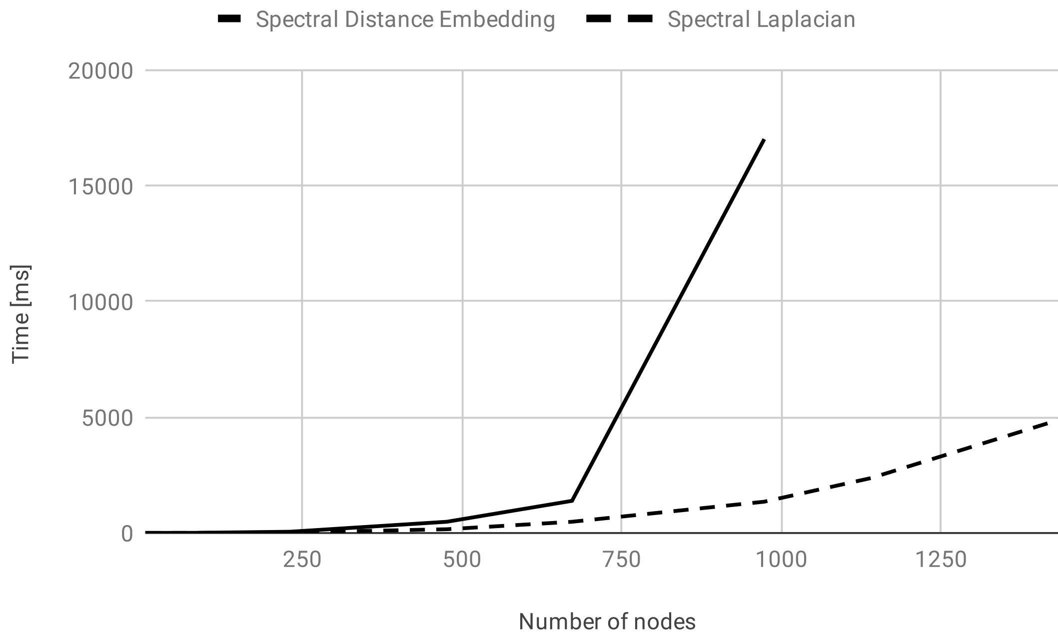



**Figure 4.9:** Computation time for different spectral algorithms

## 4.3 Force Directed Method

Only one version of this algorithms was implemented. The results are shown together with the other methods in figure 4.11 and 4.12.

## 4.4 Community Method

The Kazuko query was used to compare the community methods. A simple graph drawing method came to be used for comparing community versions. The circular method places all nodes on a circle in any order, requiring only one calculation of position for each node. Figure 4.10 presents the results of this benchmark. Four versions of the community algorithm where compared against each other:

**Circular-Circular-LouvainImp** This version uses the Louvain algorithm described in section 2.4.1 to divide the graph. Then the circular method draws each subgraph. The

subgraphs are arranged using the circular method. The version was selected for comparison to show performance of a fast drawing method paired with a good, but slow, division algorithm.

**Force-Force-LouvainImp** This version also uses the Louvain algorithm. However the subgraph drawing and arrangement are performed using the force directed algorithm described in section 2.3. The version was selected for comparison to show performance of a slow drawing method paired with a good division algorithm. It is also a good candidate for comparison with the force directed method being performed on the entire graph.

**Circular-Circular-RandomWalk** This version uses the circular method for subgraph drawing and arrangement. However they are combined with the random walks algorithm described in 3.4. By using the random walks algorithm the performance of a fast drawing algorithm combined with fast division is presented.

**Force-Force-RandomWalk** This version uses the random walk algorithm for divison. Whereafter the force based method is used to draw and arrange the subgraphs. This version presents the impact of selecting a bad division algorithm when using slow subgraph drawing methods.

Overall, Circular-Circular-RandomWalk is fastest, whereas Force-Force RandomWalk is the slowest.

Notably, the LouvainImp versions of the method showed an unexplained behaviour. At around 1000 nodes around one third of the benchmarks took up to 1800ms to run whilst the others took around 480ms. The figure 4.10 shows the average of those benchmarks. At 1400 nodes, the different benchmarks returned results at speeds within 3 percent of the average. The behaviour occurred repeatably at different days and on different computers.

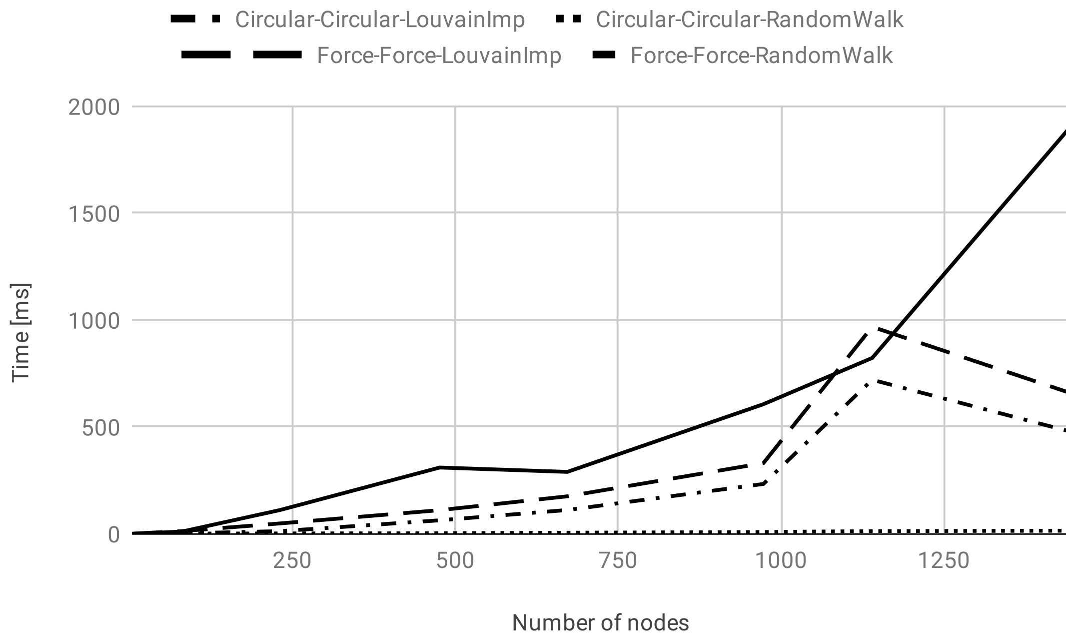



**Figure 4.10:** Computation time for community algorithms

# 4.5 Summary

For each method, representative candidates were chosen based on performance and visual result.

**Layered (LabelLayer)** This configuration was chosen to represent the label layering versions of the layered approach. It uses median ordering for crossing reduction, and Brandes and Köpf for coordinate assignment.

**Layered (Query)** This configuration was chosen to represent the query based layering versions of the layered approach. It uses median ordering for crossing reduction and Brandes and Köpf for coordinate assignment.

**Force Directed** Only one configuration of the force directed approach was implemented.

**Community algorithm** This configuration represents the community method. It uses force based for subgraph drawing, as well as subgraph arrangement. It uses the LouvainImp division algorithm. It represents the the performance compared to pure force based solutions.

**Spectral** The Laplacian version was used due to the implementation errors of spectral distance embedding discussed in 5.2.1.

These were compared to each other using the Kazuko query and the results are presented in figure 4.11. Figure 4.12 shows the same graphs being compared using the edge query.

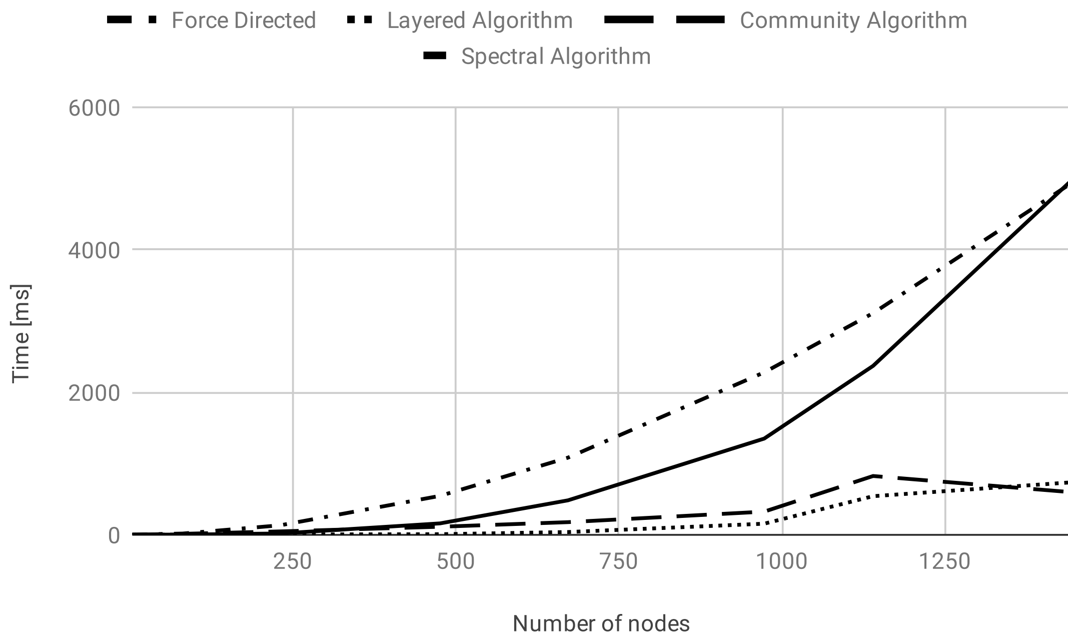



**Figure 4.11:** Computation time for different drawing algorithms using the Kazuko query

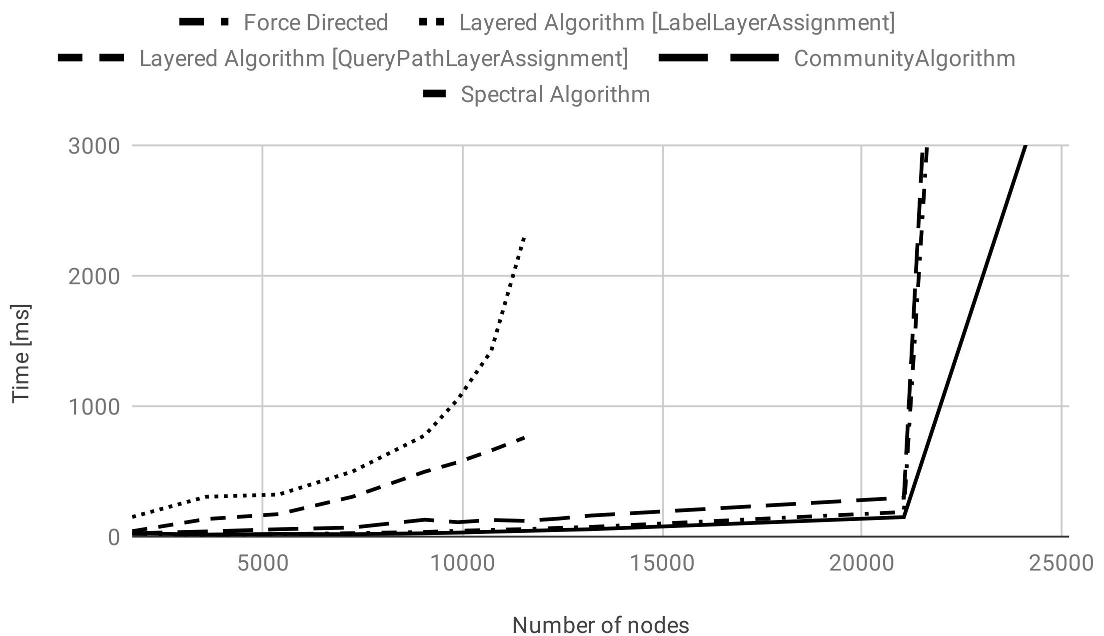



**Figure 4.12:** Computation time for different drawing algorithms using the edge query

# Chapter 5

# Discussion

The most important factor for the performance of the drawing methods turned out to be the sparseness of graph databases. Since the amount of edges is usually much larger than the amount of nodes, drawing algorithms should be evaluated on how many calculations they require for each edge. This is where algorithms, like the spectral algorithm, show their strength. However, in the case of graph databases, the number of edges is generally close to the number of nodes. This results in node calculations being much more relevant.

Furthermore, the graphs often contain similar patterns, such as a person being connected to their phone number and social security number. When searching for person-ssn relationships, the graph database will return a large set of disconnected subgraphs–each containing one person and one social security number. Drawing many smaller graphs is much faster than drawing a single large graph. The number of nodes and edges becomes much less relevant than the amount of subgraphs in these cases.

Finding large connected subgraphs in the graph database turned out to be problematic in itself. The limit function in Cypher only returns whichever elements were encountered first, rather than a set of elements which are closely related. Thereby repeating the previous situation of returning a large amount of disconnected subgraphs. When selecting a Cypher query specifically to return nodes connected to a single point as shown in the Kazuko query, the database became the slowest part of the entire process. Moreover, detail is quickly lost when viewing hundreds of nodes and edges. This leads to the conclusion that connected graphs are only really useful when looking at small examples, such as a Kazuko query with $X \leq 5$. Meanwhile, larger graphs should be used to analyze structure of groups of nodes rather than specific details.

Since Neo4j uses the graph visualization interactively, there is the requirement of performing the drawing within relatively short time. However, as long as the largest connected component contains less than 600 nodes, any of the methods will create a drawing within one second. Drawings with less then 100 nodes will be drawn within 0.1 seconds by every method as well.

# 5.1 Layered Method

## 5.1.1 Layering algorithms

The layering algorithms are arguably the most vital parts of the layered approach. Creating the layers differently will directly impact what information is portrayed to the observer. The other algorithms in the Layered method only serve to clarify this information. Because of this, there is not a single "best" layering; it depends on what information the observer wishes to receive. The query path layering and the label layering are designed to showcase and explore how the layering can be made specifically for the data in graph databases.

In addition to its informational aspect, the layerings also have a big impact on the time complexity of the algorithm. While each algorithm will have its own complexity, the layerings created will have an impact on both the ordering algorithm and the coordinate assignment. This is why the results included the total time for each algorithm, instead of the computation time for each individual step.

One way the layerings affect the graph is the number of dummy nodes created. As dummy nodes are created for each layer an edge passes, long edges can create a multitude of dummy nodes. The number of dummy nodes created for the different layerings is shown in figure 4.2.

The label layering algorithm finds a layering that attempts to both minimize the long edges, and maximize the number of edges directed to a lower layer. The minimum height layering puts the node in the fewest amount of layers, and therefore creates nearly zero dummy nodes. The query path layering sorts nodes into layers based directly on the user's input. This leads to a readable image for the user, however, it can create a large amount of dummy nodes as seen in figure 4.2.

The query path layering is obviously dependent on the query itself. If the query result contains no cycles, there will be no long edges at all, and thus no dummy nodes will occur. The query analyzed in the results chapter encounters a problem leading to a large amount of dummy nodes. The problem is illustrated in figure 5.1 and figure 5.2. In the first figure, the algorithm was only given paths of length 4 or shorter, and the drawing contains only two dummy nodes marked in black. In the second figure, however, the algorithm was also given the paths $S - C - B - A - D - E$ and $S - A - D - C - B - E$. The algorithm places node $A$ in the layer below node $B$ as one of the new paths dictate, which leads to the image shown in figure 5.2. The second drawing contains five times as many dummy nodes as the previous drawing. A second problem becomes visible when looking at the drawings: the algorithm is forced to ignore one of the given paths in favour of the other. A way to reduce the impact of this is to avoid undirected queries.

### Label Layering

The Label layering can provide a quick understanding of the structure in the graph, as well as the relationships between the labels. Compared to the other algorithms, the label layering requires some initial time to start. This is because the order of the labels has to be decided first, which requires some processing time. In the test for figure 4.1, the randomized heuristics for this was used. The number of layers in the test graph was seven for all but the first few queries. This means that an optimal solution could probably be found for these instances by

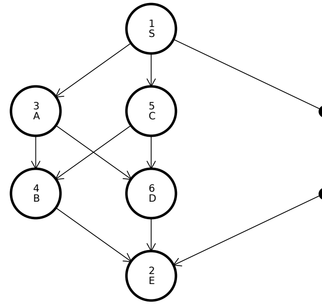



**Figure 5.1:** Query layering example with short paths

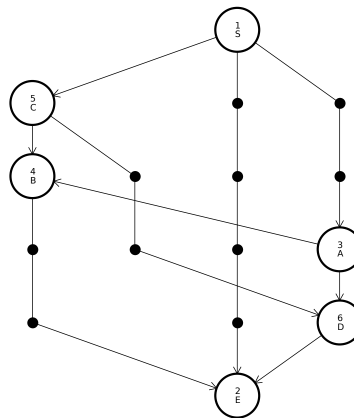



**Figure 5.2:** Query layering example with long paths

trying every permutation. However, the randomized heuristic was still used as it is a lot faster, and it better represents what would be used in other instances. The number of attempts used in the heuristic could be changed for a faster or slower result, with potential for a worse or better final result. In this case, 200 attempts were made. Some research is needed on what number is suitable in each situation, as it surely varies on the number of labels and possible other variables. A few prior experiments showed that in many cases as few as 100 attempts could yield the same solution as if 10000 were used.

In figure 4.2, we see that the algorithm produces relatively few dummy nodes in these tests. This helps explain why the time in figure 4.1 only rises slowly with the number of nodes. After the initial step is completed, the label layering as well as the other steps will be completed relatively fast.

## Query Path Layering

The Query Path Layering approach described in Section 3.1.1 gave good visual results whilst being reasonably fast. This approach combines the benefits of layered algorithms with drawing based on the inherent structure of a Cypher query. Cypher queries are naturally read by

humans from left to right and when read, they will give the reader some form of understanding about the expected result. The query path layering uses this knowledge by arranging the nodes into layers based on the query written. For example, a query finding paths 2 steps away from node ***A*** can have the following structure: node ***A*** at the top, neighbors of ***A*** at the layer below ***A***, and the remaining nodes two layers below ***A***. This way the structure remains clear to the user, and the result will be instantly recognizable.

By trying to avoid crossings, the layered algorithm then groups the nodes in each layer based on how interconnected they are.

### 5.1.2 Node Ordering Algorithms

The node ordering algorithms only serve to increase the clarity of the graph. Without them, the graphs would be a swarm of edges going in every direction. There are multiple ways of measuring the improvement the ordering algorithms bring to the graph–one of them is the number of crossings. A high number of crossings indicates that the edges often go in different directions; overlapping edges can be hard to read. As shown in figure 4.5 and 4.6, the number of crossings without the use of an ordering algorithm is significantly higher than when they are used. This is natural, as putting each node into each layer without forethought will naturally lead to nodes being placed far away from the nodes they are related to. This is why the mean and median orderings are successful at removing so many of the crossings. A single node placed at the opposite side of a lone parent would cross almost every other edge in the layer. By simply moving it close to given parent, most crossings are removed.

Moreover, this is why the swapping algorithm takes a considerable amount of time compared to the mean and median orderings. A node originally placed far away from its parent would have to swap places with a multitude of other nodes. Comparatively, the mean and median algorithm only need to calculate the position of each node once. The swapping algorithm will repeatedly iterate through the nodes in each layer until no swaps between adjacent nodes are preferable. Consequently, the resulting time can vary greatly on the original order of the nodes. This has no impact on the mean and median orderings. In the best case scenario, the swapping algorithm is performed on an already ordered graph. In which case, each pair of nodes and their edges only needs to be processed once. It could therefore prove to be useful when this is run after the mean or median orderings to further decrease the number of nodes. Figure 4.3 shows how the time for running the swapping algorithm after the mean ordering results in only a very slight increase in time.

The ordering algorithms all perform their orderings from the top towards the bottom. This means that sometimes the orderings might put the nodes in an order that seems correct when considering the above layer, but not the one below. This is can be seen in the swapping algorithm where two nodes are swapped; there is a slight crossing decrease in the above layer, and yet a big increase in layer below. This is displayed in table A.2 in the appendix. In the *Query path layering - Mean and swap*, the swapping actually increases the amount of crossings in one of the instances (from 8660 to 8767). This is usually not a problem, however, as the next layer will be based on this new configuration.

In regards to number of crossings, the mean and median ordering algorithms performed exceptionally well, on average reducing the number of crossings by 93% each in the label layering. In the query path layering, the median ordering outperformed the mean ordering slightly, thus reducing crossings by 90% compared to 87%.

Using the swapping algorithm after the mean algorithm yielded minor improvements. On average, it improved the crossing reduction by 2% and 7% on the label layering and query path respectively.

### 5.1.3 Coordinate Assignment Algorithms

The coordinate assignment algorithms have no impact on the structure of the graph. Thus, they are hard to evaluate. Some metrics can be used, such as the width, height, or number of bends in the edges. However, any of these metrics can be improved at the expense of the others. As shown in figure 4.8 and figure 4.7 the coordinate assignments can have a large impact on the final computation time. The Brandes Kopf algorithm is shown to take about three times longer than the naive and mean alternatives. The cause of this becomes quite apparent when the figure is compared to the number of dummy nodes shown in figure 4.2. The Brandes Kopf method is naturally more dependent of the total number of nodes, including the Naive and Mean algorithms. The naive algorithm will process all nodes and dummy nodes once, placing them greedily. The mean algorithm prioritizes placing dummy nodes first by calculating their position based on just one parent. The Brandes and Köpf algorithm performs multiple loops and depth first searches through the set of nodes (dummy nodes included). Its complexity is $O(n)$ [1], however, the constant factor before n is considerable as shown in the slower results.

The Brandes and Köpf algorithm can be run using any ordering algorithm, and will provide very good visual results while still being reasonably fast. Even at 1000 nodes, which is far more than what could be understood in an image, the result is provided in about 1/5 of a second under our testing conditions. The mean coordinate assignment is useful because it is able to very quickly create a good result limited to the mean ordering. It is less flexible, however, as other orderings would simply get overwritten in the final step. If the median ordering is desired, the algorithm could simply be modified slightly. The same would be possible with other solutions. If the ordering desired is known and not subject to change, the mean coordinate assignment could be the best alternative for fast results.

### 5.1.4 Memory Requirements of Brandes and Köpf

The algorithm presented by Brandes and Köpf used more memory than any other algorithm discussed in this paper. The other algorithms were able to draw graphs with large amounts of nodes using less than 500 MB of RAM memory. The Brandes and Köpf algorithm, however, quickly ran into stack overflow exceptions. The stack size was increased to 8GB as described in section 4. As seen in figure 4.12, both layered versions still crashed with stack overflow errors above 10000 nodes. By dividing the graph into disconnected subgraphs, as was done with other methods in this paper, the problem could have been reduced. However it would not have removed the problem since the graph is not guaranteed to be composed of disconnected subgraphs.

### 5.1.5 Time Complexity of Layered Methods

**Cycle removal** The cycle removal step described in section 2.1.1 processes each node (N) once, with a complexity of $O(N^2)$.

**Layering algorithms** The label layering has multiple steps with different complexity. To find the best layering, random configurations are made a set number of times; each of which takes $O(L)$ time, where $L$ is the number of labels. The layers in these configurations are then reordered as long as the number of dummy edges produced is lowered. As each reorder decreases the number of dummy nodes significantly, it will only be performed a handful of times. Calculating the number of created dummy nodes takes $O(E)$ time. Each node is also processed a few times for actions, such as determining what labels are present, inserting labels into the final layering, etc., taking $O(N)$ time. The entire label layering algorithm therefore has a total complexity of $O(L + E + N)$.

The path layering only requires one iteration through each path returned by Neo4j and a post order traversal of the graph. Since both of these steps are $O(N)$, the layering performs quite fast. As a prerequisite to later steps of the layered algorithm, no edges may exist which point from a lower layer to one above. Finding these edges has a time complexity of $O(N * W)$ where W is the maximum number of nodes in a layer. Since a layer can contain at most N nodes, the time complexity becomes $O(N^2)$.

**Ordering algorithms** The mean and median algorithms both order the nodes in each layer by their calculated value. This value is for each node determined by its edges to its neighbors on the above layer. Since each edge and each node is processed once, the time complexity becomes $O(N + E)$.

The swap ordering considers each possible pair of nodes at most once, meaning the complexity has a worst case scenario of $O(N^2)$

**Coordinate assignment algorithms** The mean coordinate assignment calculates each nodes optimal x coordinate in the same way as the mean ordering. This takes $O(N + E)$ time. After this, the nodes need to be put at a distance from each other node in the layer. The worst case scenario for this would be a single layer containing all nodes; each node would need to compare its location with each other node in the layer. The worst case complexity therefore takes $O(N^2)$ time.

The Brandes and Köpf algorithm finds horizontal coordinates in $O(N)$ as described by its authors [1]. However, the algorithm performs expensive walks through the graph structure for each node and ghost node. This causes the overall running time to be relatively slow.

## 5.2 Spectral Method

The spectral algorithm gave unexpectedly bad results. Previous articles praise the efficiency of the approach [2] [16]. Yet, the benchmarks in this project show comparatively bad results. Figure 4.9 shows the exponential nature of the approach. The spectral approach is highly dependent on the number of nodes in the graph. It is, however, entirely independent of

the number of edges in the graph. This makes the spectral approach very efficient for dense graphs, but impractical for the typical Cypher query.

## 5.2.1 Problems

### Implementation Errors

The results shown in 4.9 indicate that something is wrong with this implementation of spectral distance embedding. The power method does not converge quickly enough to find eigenvectors in less than 30 seconds and the benchmark is canceled. This was not investigated further since spectral graph drawing was deemed a bad fit for sparse graphs.

### Sparse Matrix

When two nodes in a graph have the same relation to the rest of the graph, the corresponding rows and columns in the Laplacian and distance matrices become close to equivalent. The only difference is their relative connection. This results in the situation where the two nodes get assigned the same coordinates. This situation often occurs in sparse graphs, such as the ones Neo4j normally works with.

### Sparse vs Dense

Dense matrices have the definite drawback of using $n^2$ memory space, which could be troublesome but could be overcome depending on the use-case. A more important problem arose due to the usage of a matrix library. EJML stores the matrix data into a single dimensional array of doubles. The amount of variables that can be stored in this array is limited by the maximum value of an integer in Java. Since the array size is calculated by multiplying the number of rows with the number of columns, we get the limit $numRows = numCols \leq \sqrt{Integer.Max} \leq 46341$. This limit could be avoided by implementing a specialized dense matrix storage and linear algebra methods.

The sparse approach works well when using the Laplacian approach because the Laplacian is as sparse as the graph. Spectral distance embedding, however, creates a dense matrix even when the graph is sparse. The lookup time of EJML sparse matrix is $O(N)$ due to the way the data is stored. This impacts performance of every piece of code which needs to do many lookups. This could potentially be mitigated by customizing algorithms to use the background structure of the sparse matrix. Even worse though, insertion time of the sparse matrix is also $O(N)$ which means that all code that fills each element of a sparse matrix results in $O(N^3)$ time complexity.

Spectral distance embedding thus requires a dense matrix storage, unless a custom matrix storage solution is implemented. Using the dense storage option limits the number of nodes to 46341.

The Laplacian, however, gains a lot by using sparse storage options.

### 5.2.2 Time Complexity

The time complexity is dictated by finding the eigenvalues. This depends on the time complexity of matrix multiplication, so the time complexity is at least $O(n^2)$. The results in 4.11 are comparable to the force based method because of this. Still, the spectral complexity is entirely independent on the amount of edges in a graph, resulting in it being described as a fast solution.

## 5.3 Force Directed Method

The force directed method discussed in this paper is a basic implementation with no optimization techniques applied. However, the community algorithm can be seen as an improvement on the force directed method.

### 5.3.1 Time Complexity

During each step of the force directed approach, two calculations were performed: the repulsive force and attractive force for each pair of nodes and each edge, respectively. The complexity of this is $O(n^2 + e)$. For sparse graphs, such as the ones discussed in this paper, the node complexity dominates the execution time whereby the complexity of one step becomes $O(n^2)$. In other graphs, however, the calculation of the attractive force can become dominating.

The steps are repeated a number of times until the sum of forces no longer decreases. In practice, the number of steps was low. Overall, the time complexity of the force based algorithm can be seen as $O(n^2)$.

## 5.4 Community Method

The quality of the community algorithm is strongly dependent how well the graph is split into communities. Nodes in the same community will be placed close towards each other in the final graph. Example drawings of different community algorithms are presented in Figure B.19 to B.22 in the Appendix. The Circular and Force based algorithm were only modified slightly to be able to arrange the subgraphs. The distance between the subgraph representing nodes was increased. There is room for improvement here; when using force based, the repulsive force could be defined by how many nodes each subgraph contains, for example.

### 5.4.1 Time Complexity

The performance of the community algorithm is also dependent on the community division. Still, a visually good community division is not always good from the performance standpoint. The time complexity can be divided into three parts:

**Dividing the graph** The RandomWalk algorithm only performs a constant amount of walks and only walks a constant depth, so it results in a quick complexity of $O(n)$. The Louvain algorithm can be performed in $O(nlogk)$ where k is the average degree according to a paper by Vincent Traag [20].

**Drawing each subgraph** The complexity of drawing each subgraph depends on the algorithm used, and how many nodes the subgraph contains. With a drawing algorithm that has time complexity of $g(x)$, the total complexity becomes $O(max(g(n_0), g(n_1), \ldots, g(n_k)))$. When $g(x)$ is worse than linear, any large subgraph will dominate the result. Take for example, three subgraphs of size 3, 3 and 10 and set $g(x) = x^2$. The sum expands to $3^2 + 3^2 + 10^2 = 9 + 9 + 100$. The best case scenario would be n subgraphs of size 1 when only taking this step into consideration.

**Arranging subgraphs** Each subgraph is represented as a node in a new graph, then any drawing algorithm may be used to draw this metagraph. The complexity depends on which drawing algorithm is used and the structure of the metagraph.

The complete complexity of the community algorithm is

$$O(max(f(n), g(n_0), g(n_1), \ldots, g(n_k), h(k))) \tag{5.1}$$

where $f(x)$ is the complexity of dividing the graph, $g(x)$ is the complexity of drawing a single subgraph, $h(x)$ is the complexity of arranging the subgraphs, $n_i$ is the amount of nodes of a single subgraph, and $k$ is the number of subgraphs.

Assuming the same algorithm is used for drawing each subgraph and arranging the subgraphs, that is $g(x) = h(x)$, the most efficient division would result in $\sqrt{n}$ subgraphs with $\sqrt{n}$ nodes each. This would lead to a total complexity of

$$O(max(f(n), g(n_0), g(n_1), \ldots, g(n_{\sqrt{n}}), h(\sqrt{n}))) \tag{5.2}$$

This can be seen in figure 4.10. RandomWalk creates a bad division with too many subgraphs and $h(x)$ dominates the resulting time complexity. As a result the Circular-RandomWalk solution with $h(x) = x$ is very fast and the Force-RandomWalk with $h(x) = x^2$ is very slow.

The two algorithms, using LouvainImp for division, are close to each other; the choice of $h(x)$ does not affect the results as much. Instead, the complexity $f(x)$ of running LouvainImp itself dominates the result.

When different algorithms are chosen for subgraph drawing ($g(x)$) and arranging ($h(x)$), close attention should be placed on the division algorithm. If it creates a large amount of subgraphs, $h(x)$ will determine the performance. If it instead creates few large subgraphs, or subgraphs of different size, $g(x)$ will be the limiting factor.

## 5.5 Disconnected Subgraphs

Figure 4.12 shows a problem with the versions of the layered method that use the Brandes and Köpf algorithm; it cannot handle graphs which are larger than 10000 nodes. The comparison to other algorithms in that figure is still flawed. The Layered method is the only method that does not divide the graph into disconnected subgraphs, which is an implementation fault rather than a problem with the method. In the case of a single connected subgraph with

more than 10000 nodes, the Brandes and Köpf algorithm would still require large amounts of memory and potentially break.

In the same figure we can also see that drawing each subgraph on its own, and then arranging them in a grid proves a good strategy. When using this strategy the choice of drawing algorithm no longer affects the run time. The community algorithm is designed to benefit from this, even when there are no disconnected subgraphs.

## 5.6 Query Issues

Selecting representative queries for the experiments was difficult. Take for example the query `(:Person)->(:SSN)`, Neo4j will return many disconnected graphs, each containing only one person and their SSN. By drawing each of these disconnected graphs on their own, the drawing time complexity becomes almost linear. The query `(:Shop)->(:Person)->(:SSN)` in contrast could result in all of the previously disconnected graphs become connected to a singular Shop node. Suddenly the different SSNs positions are depending on each other and the graphs can no longer be drawn separately. The results of this are visible in figure 4.12. The graph contains a large amount of disconnected subgraphs for measurements under 20000 nodes, leading to minimal drawing time. Then above 20000 nodes the subgraphs merge together and the time complexity skyrockets.

Drawing disconnected graphs on their own was not implemented for the layered algorithm which leads to the poor performance seen in this figure. This should not be seen as an indicator of performance however, since this feature could easily be implemented for any drawing algorithm.

Overall the edge query shown in the figure gives little information about the performance of any drawing algorithm, but rather shows the disconnected structure of query results. Similar problems where encountered with most other queries that did not specifically enforce a connected graph.

# Chapter 6
# Conclusions

After all drawing methods had been finalized, we created appendix B. The appendix shows the resulting image using all the different configurations on the same graph. These images show the visual characteristics of the algorithms, rather than the performance measured in chapter 4. A visual comparison obviously depends on the viewer, who themselves should decide which algorithms are suitable for their needs.

The layered method gave the best representation of the database content. Especially the layered methods that use label layering or query path layering. This is simply a product of the drawing method considering a broader context than other methods. Furthermore, the layered method contains a specific step for reducing the amount of edge crossings, which the other methods do not.

The force based method and the community method using force based for drawing subgraphs gave comparable results with the force based method generally being better. With a good division algorithm, and improved subgraph arrangement, the community algorithm could become equally readable whilst showing the information gained from community grouping.

The spectral algorithms gave very bad results with large amounts of nodes sharing the same coordinates. This is the result of the sparseness of graph databases.

Performance-wise, the layered and community methods performed as well as expected based on their low time complexities. Both spectral and force based have polynomial time complexities. This is also present in the experimental data namely figure 4.11.

When the amount of nodes to be drawn is low, the drawing method should be chosen based on the visual representation that is desired. The same is applicable for graphs comprised of disconnected subgraphs that have a low amount of nodes. In those cases, the graph should be divided into subgraphs, which are each drawn by themselves and then combined into a larger picture. If, instead, the graph is connected a variant of the community algorithm could be used to gain similar performance.

When none of the above are applicable, the complexity of the drawing algorithms become relevant. Here, the layered methods provide similar results to the community algorithm.

# 6.1 Further Research

## 6.1.1 Alternative Methods

For this project, the methods that seemed useful for Neo4j's use-case were selected and investigated. However, there are more possible approaches that have not yet been tested with the type of data common in graph databases. Some of these are mentioned in 2.

## 6.1.2 Layered

This thesis explores some of the more common methods of performing the layered graph drawing, though there are many more ways of altering the layered algorithm. The methods evaluated were chosen based on their fit for graph databases, as well as their prevalence. The most important part of the algorithm, the layering step, was focused on two custom layering algorithms. This was done to explore how the layered graph drawing could be used on graph databases in ways that would not be possible with other structures. The already established layering algorithms could naturally also be used to create a more generic graph. If more time were allotted for this project, some other layering algorithms could have been implemented to compare with the label and query layerings. Some examples of algorithms which could be implemented are described in 2.1.2. The linear programming approach used to minimize the number of dummy nodes could prove very effective if used with the Brandes & Köpf algorithm, as it is heavily dependant on the dummy nodes.

Another interesting algorithm we could have implemented was developed by Eiglsperger et al. The algorithm is able to decrease the processing time heavily, to an almost linear complexity [7]. It is based on two facts: the dummy nodes are usually placed in a long line with the same X coordinate, and the dummy nodes usually outnumber the real nodes greatly. As long edges are replaced by a multitude of dummy nodes in a line, the line can instead be replaced by a single segment. In the node ordering step, this entire segment can be moved, but not the dummy nodes it represents.

A potential improvement could be made to the node ordering algorithms. Two of the layering algorithms, predefined layering and query based layering, break the rule of edges being directed downward. The node ordering algorithms arrange lower layers based on the upper layers. This leads to upper layers dictating the position of lower layers. It can be desired to have source nodes dictate placement of other nodes, for example in tree structures. In our case however edges can be directed both up and down. There is potential to improve the node ordering by traversing the layers from the middle out or in another order entirely.

## 6.1.3 Community Algorithm

Since the community algorithm is modular, multiple different ideas could be incorporated. Different drawing algorithms can be used to draw subgraphs as well as to arrange them. It could be possible to optimize each to suit specific requirements. By selecting a different division algorithm, different community types or structures could be found. It could also be possible to write new division algorithms that find specific elements in a graph. For example, the algorithm could find nodes with only one neighbor and place them into a community.

Another approach could be to perform the community method recursively by drawing each subgraph using the community algorithm again. This would require a good division algorithm, as it should divide into reasonable subgraphs and be very fast.

# Bibliography


[1] Ulrik Brandes and Boris Köpf. Fast and simple horizontal coordinate assignment. Graph Drawing Lecture Notes in Computer Science, page 31–44, 2002.

[2] Ali Civril, Malik Magdon-Ismail, and Eli Bocek-Rivele. SDE: Graph drawing using spectral distance embedding. In Patrick Healy and Nikola S. Nikolov, editors, Graph Drawing, pages 512–513, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[3] P. EADES. A heuristic for graph drawing. Congressus Numerantium, 42:149–160, 1984.

[4] Peter Eades, Xuemin Lin, and William F Smyth. A fast and effective heuristic for the feedback arc set problem. Information Processing Letters, 47(6):319–323, 1993.

[5] Peter Eades and Sue Whitesides. Drawing graphs in two layers. Theoretical Computer Science, 131(2):361–374, 1994.

[6] Peter Eades and Nicholas C Wormald. Edge crossings in drawings of bipartite graphs. Algorithmica, 11(4):379–403, 1994.

[7] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. In International Symposium on Graph Drawing, pages 155–166. Springer, 2004.

[8] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of sugiyama's algorithm for layered graph drawing. Graph Drawing Lecture Notes in Computer Science, page 155–166, 2005.

[9] Efficient Java Matrix Library. `http://ejml.org`. Accessed: 2019-11-01.

[10] feedbackarcapx. `http://www.nada.kth.se/~viggo/wwwcompendium/node20.html#GT11`. Accessed: 2020-01-29.

[11] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. Software: Practice and Experience, 21(11):1129–1164, 1991.

[12] Weiqing He and Kim Marriott. Constrained graph layout. In International Symposium on Graph Drawing, pages 217–232. Springer, 1996.

[13] Yifan Hu. Efficient and high quality force-directed graph drawing. Mathematica Journal, 10:37–71, 01 2006.

[14] Goossen Kant. Algorithms for drawing planar graphs. PhD thesis, Rijksuniversiteit te Utrecht, 1993.

[15] Hall Kenneth M. An r-dimensional quadratic placement algorithm. Management Science, 17(3):219, 1970.

[16] Yehuda Koren. On spectral graph drawing. In International Computing and Combinatorics Conference, pages 496–508. Springer, 2003.

[17] Jakob Nielsen. Usability engineering. Morgan Kaufmann an imprint of Academic Press, a Harcourt Science and Technology Company, 1993.

[18] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man, and Cybernetics, 11(2):109–125, 1981.

[19] Ioannis G Tollis, Giuseppe Di Battista, Peter Eades, and Roberto Tamassia. Graph drawing: Algorithms for the visualization of graphs. Prentice Hall, 1999.

[20] Vincent A Traag. Faster unfolding of communities: Speeding up the louvain algorithm. Physical Review E, 92(3):032801, 2015.

[21] kann Viggo. On the approximability of NP-complete optimization problems. PhD thesis, Royal Institute of Technology Stockholm, 1992.

[22] Webcola. `https://ialab.it.monash.edu/webcola/`. Accessed: 2020-01-29.

[23] What is a Graph Database? `https://neo4j.com/developer/graph-database/`. Accessed: 2020-01-26.

# Appendices

# Appendix A
# Benchmarks

**Table A.1:** Layered algorithm Time

| Full name | Running time [ms] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Nodes: | 4 | 81 | 231 | 476 | 672 | 973 | 1140 | 1442 | 1450 |
| Edges: | 3 | 80 | 250 | 683 | 1117 | 1623 | 1875 | 2220 | 2236 |
| | | | | | | | | | |
| LabelLayerAssignment - MedianOrdering - BrandesKopf | 1.2 | 1.9 | 4.6 | 18.5 | 33.5 | 61.6 | 96.9 | 120.4 | 120.4 |
| QueryPathLayerAssignment - MedianOrdering - BrandesKopf | 0.8 | 1.2 | 2.2 | 6.2 | 38.5 | 157.7 | 521.7 | 709.1 | 996.2 |
| PredefLayerAssignment - MedianOrdering - BrandesKopf | 0.2 | 1.4 | 2.8 | 18.4 | 46.3 | 83.3 | 100.7 | 153 | 155.7 |
| PredefLayerAssignment - MedianOrdering - BrandesKopf | 0.5 | 1.9 | 4.5 | 11.7 | 28.2 | 70.2 | 92.1 | 103.6 | 108.2 |
| LongestPathLayerAssignment - MedianOrdering - BrandesKopf | 1.8 | 1.1 | 1.4 | 4.6 | 7.7 | 18.4 | 18.4 | 27.9 | 28.2 |
| | | | | | | | | | |
| LabelLayerAssignment - null - BrandesKopf | 1.4 | 2.1 | 4.6 | 15.3 | 28.4 | 49.8 | 76.7 | 94.1 | 95.7 |
| LabelLayerAssignment - MeanOrdering - BrandesKopf | 1.3 | 1.5 | 5.3 | 19 | 33.4 | 61.3 | 93.9 | 119.4 | 120.5 |
| LabelLayerAssignment - MedianOrdering - BrandesKopf | 1.6 | 2 | 4.6 | 18.3 | 34.1 | 61.2 | 94.3 | 119.8 | 121.3 |
| LabelLayerAssignment - Mean and Swap Ordering - BrandesKopf | 1.6 | 1.3 | 5.3 | 23.2 | 39 | 85.6 | 116.2 | 146.7 | 152 |
| LabelLayerAssignment - SwapNeighbors - BrandesKopf | 0.6 | 1.6 | 6.5 | 58.2 | 145.3 | 233.4 | 576.6 | 1054.9 | 1017 |
| | | | | | | | | | |
| QueryPathLayerAssignment - null - BrandesKopf | 0.8 | 1 | 1.4 | 5.2 | 34.3 | 141.2 | 485.9 | 657.2 | 935.2 |
| QueryPathLayerAssignment - MeanOrdering - BrandesKopf | 1.4 | 0.9 | 1.8 | 6 | 38.4 | 165.3 | 522.7 | 712.5 | 1000.6 |
| QueryPathLayerAssignment - MedianOrdering - BrandesKopf | 0.8 | 0.7 | 1.6 | 6 | 37.8 | 157.4 | 523.5 | 708 | 992 |
| QueryPathLayerAssignment - Mean and Swap Ordering - BrandesKopf | 0.6 | 1 | 1.6 | 6.7 | 53.2 | 237.9 | 562.4 | 840.7 | 1239.3 |
| QueryPathLayerAssignment - SwapNeighbors - BrandesKopf | 0 | 0.6 | 5.5 | 26.1 | 154.3 | 481.5 | 2100.8 | 2568.7 | 3487 |
| | | | | | | | | | |
| LabelLayerAssignment - MedianOrdering - Naive | 1.6 | 1.2 | 3.3 | 9.2 | 17.2 | 31.9 | 49.1 | 63.5 | 62.9 |
| LabelLayerAssignment - null - MeanCoord | 0.6 | 1.9 | 4 | 8.8 | 14.9 | 24.5 | 36.6 | 46.5 | 46.2 |
| LabelLayerAssignment - MedianOrdering - BrandesKopf | 1.1 | 1.5 | 4.5 | 18.3 | 32.8 | 60.6 | 93.6 | 119.1 | 122.3 |
| QueryPathLayerAssignment - MedianOrdering - Naive | 0.6 | 0.6 | 0.8 | 2.9 | 13.4 | 52.1 | 117.4 | 207 | 406.9 |
| QueryPathLayerAssignment - null - MeanCoord | 0.8 | 0.4 | 0.9 | 2.7 | 12.4 | 40.8 | 98.9 | 174.7 | 370.4 |
| QueryPathLayerAssignment - MedianOrdering - BrandesKopf | 1.3 | 0.5 | 1.5 | 6 | 37.1 | 159.9 | 526.3 | 716.4 | 999.5 |

**Table A.2:** Layered algorithm crossings

| Full name | Number of crossings | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Nodes: | 4 | 81 | 231 | 476 | 672 | 973 | 1140 | 1442 | 1450 |
| Edges: | 3 | 80 | 250 | 683 | 1117 | 1623 | 1875 | 2220 | 2236 |
| | | | | | | | | | |
| LabelLayerAssignment - MedianOrdering - BrandesKopf | 0 | 0 | 1257 | 1783 | 6423 | 29778 | 28595 | 29029 | 33496 |
| QueryPathLayerAssignment - MedianOrdering - BrandesKopf | 0 | 0 | 233 | 982 | 9110 | 28636 | 8805 | 17282 | 17903 |
| PredefLayerAssignment - MedianOrdering - BrandesKopf | 0 | 91 | 768 | 80215 | 197135 | 442268 | 546342 | 919360 | 930962 |
| PredefLayerAssignment - MedianOrdering - BrandesKopf | 0 | 188 | 7889 | 31109 | 76619 | 175906 | 221904 | 330364 | 336136 |
| LongestPathLayerAssignment - MedianOrdering - BrandesKopf | 0 | 30 | 4156 | 44668 | 144709 | 309577 | 345502 | 416936 | 451025 |
| | | | | | | | | | |
| LabelLayerAssignment - null - BrandesKopf | 0 | 12 | 5619 | 75588 | 207878 | 315651 | 531858 | 820574 | 742796 |
| LabelLayerAssignment - MeanOrdering - BrandesKopf | 0 | 0 | 1308 | 1844 | 6651 | 30858 | 28605 | 29034 | 34104 |
| LabelLayerAssignment - MedianOrdering - BrandesKopf | 0 | 0 | 1257 | 1783 | 6423 | 29622 | 28595 | 29029 | 33496 |
| LabelLayerAssignment - Mean and Swap Ordering - BrandesKopf | 0 | 0 | 1238 | 1771 | 6536 | 29621 | 28338 | 28977 | 33894 |
| LabelLayerAssignment - SwapNeighbors - BrandesKopf | 0 | 0 | 3215 | 7753 | 8751 | 56498 | 47284 | 48126 | 56122 |
| | | | | | | | | | |
| QueryPathLayerAssignment - null - BrandesKopf | 0 | 280 | 2087 | 17269 | 58153 | 153813 | 135890 | 153328 | 169394 |
| QueryPathLayerAssignment - MeanOrdering - BrandesKopf | 0 | 0 | 276 | 919 | 8733 | 40597 | 8660 | 28009 | 33867 |
| QueryPathLayerAssignment - MedianOrdering - BrandesKopf | 0 | 0 | 233 | 982 | 9110 | 28636 | 8805 | 17282 | 17903 |
| QueryPathLayerAssignment - Mean and Swap Ordering - BrandesKopf | 0 | 0 | 230 | 909 | 7773 | 36196 | 8767 | 26197 | 32724 |
| QueryPathLayerAssignment - SwapNeighbors - BrandesKopf | 0 | 0 | 251 | 1889 | 19212 | 71893 | 11813 | 62570 | 64237 |

**Table A.3:** Layered algorithm dummy nodes

| Full name | Number of dummy nodes | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Nodes: | 4 | 81 | 231 | 476 | 672 | 973 | 1140 | 1442 | 1450 |
| Edges: | 3 | 80 | 250 | 683 | 1117 | 1623 | 1875 | 2220 | 2236 |
| | | | | | | | | | |
| LabelLayerAssignment - MedianOrdering - BrandesKopf | 0 | 12 | 105 | 922 | 1419 | 2137 | 2821 | 3018 | 3022 |
| QueryPathLayerAssignment - MedianOrdering - BrandesKopf | 0 | 0 | 0 | 280 | 2594 | 7967 | 19570 | 22266 | 24086 |
| PredefLayerAssignment - MedianOrdering - BrandesKopf | 1 | 84 | 144 | 1005 | 1850 | 2636 | 2799 | 3638 | 3686 |
| PredefLayerAssignment - MedianOrdering - BrandesKopf | 1 | 222 | 395 | 721 | 1358 | 2508 | 2906 | 2938 | 3018 |
| LongestPathLayerAssignment - MedianOrdering - BrandesKopf | 0 | 71 | 1 | 8 | 1 | 197 | 2 | 2 | 2 |

**Table A.4:** Community algorithm time

| Full name | Running time [ms] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Nodes: | 4 | 81 | 231 | 476 | 672 | 973 | 1140 | 1442 | 1450 |
| Edges: | 3 | 80 | 250 | 683 | 1117 | 1623 | 1875 | 2220 | 2236 |
| | | | | | | | | | |
| Circular-Circular-LouvainImp | 0.9 | 2.6 | 14.8 | 64.8 | 114.7 | 239.4 | 514.4 | 415.8 | 453.3 |
| Circular-Circular-RandomWalk | 0.8 | 2.2 | 3.6 | 2.9 | 5.1 | 8.9 | 11.6 | 13.8 | 14.1 |
| Force-Force-LouvainImp | 1.1 | 11.9 | 50.3 | 113.5 | 177.9 | 324.3 | 780.6 | 618.8 | 631 |
| Force-Force-RandomWalk | 0.4 | 11.6 | 109 | 306.1 | 290.8 | 585.1 | 836.1 | 1853.8 | 1820.9 |

**Table A.5:** Drawing method comparison

| Full name | Running time [ms] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Nodes: | 4 | 81 | 231 | 476 | 672 | 973 | 1140 | 1442 | 1450 |
| Edges: | 3 | 80 | 250 | 683 | 1117 | 1623 | 1875 | 2220 | 2236 |
| | | | | | | | | | |
| Force Directed | 1.2 | 26.2 | 125.2 | 543.3 | 1098.2 | 2283.3 | 3119.5 | 4943.1 | 4998.5 |
| Layered Algorithm [LabelLayerAssignment - MO - BK] | 2.4 | 5.8 | 11.7 | 29.6 | 39.7 | 67.9 | 98.8 | 121.4 | 121.4 |
| Layered Algorithm [QueryPathLayerAssignment - MO - BK] | 1.3 | 0.9 | 2.5 | 7.4 | 40 | 159.5 | 526.2 | 713.8 | 990.5 |
| CommunityAlgorithm [Force-Force-LouvainImp] | 0.7 | 15.7 | 49.3 | 108.6 | 176.4 | 330.5 | 631.9 | 612.7 | 615.3 |
| Spectral Algorithm [Laplacian - DenseSolver] | 1.8 | 4.3 | 19.2 | 161.6 | 489.6 | 1359.5 | 2407.1 | 4967 | 5161.8 |
| | | | | | | | | | |
| **Note:** Both Layered algorithms above use the MedianOrdering and BrandesKopf algorithms | | | | | | | | | |

# Appendix B
# Graph images

These images are the result of running different layout algorithms on the cypher query and same database. These images are the result of the query

```
match x=(:Login {firstName:"Kazuko"})-[*]-() return x limit 20
```

**Figure B.1:** LayeredAlgorithm[LabelLayerAssignment-MeanandSwapOrdering-BrandesKopf]

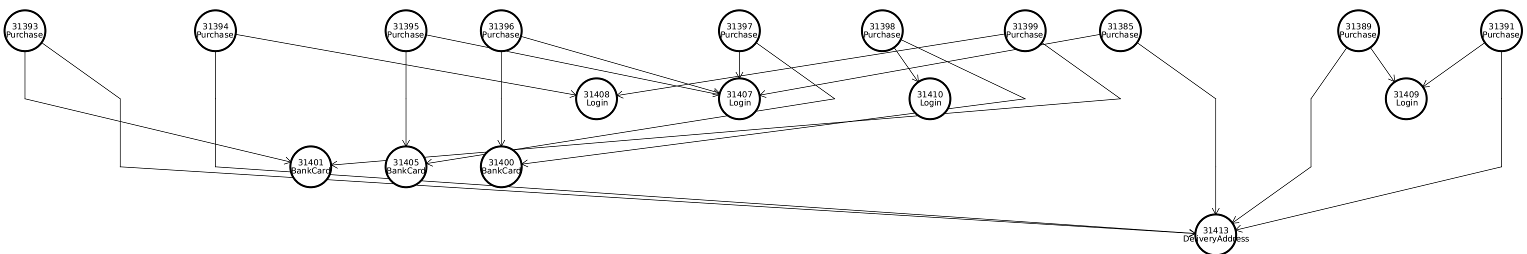



**Figure B.2:** LayeredAlgorithm[LabelLayerAssignment-MeanOrdering-BrandesKopf]

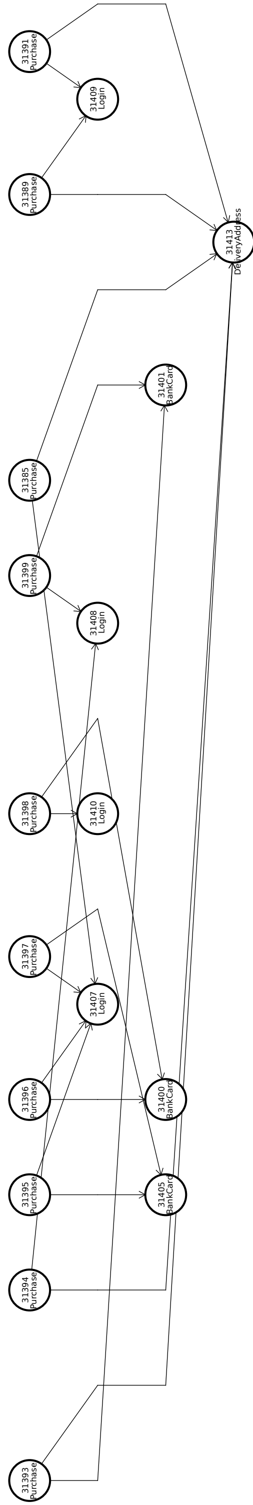



**Figure B.3:** LayeredAlgorithm[LabelLayerAssignment-MedianOrdering-BrandesKopf]

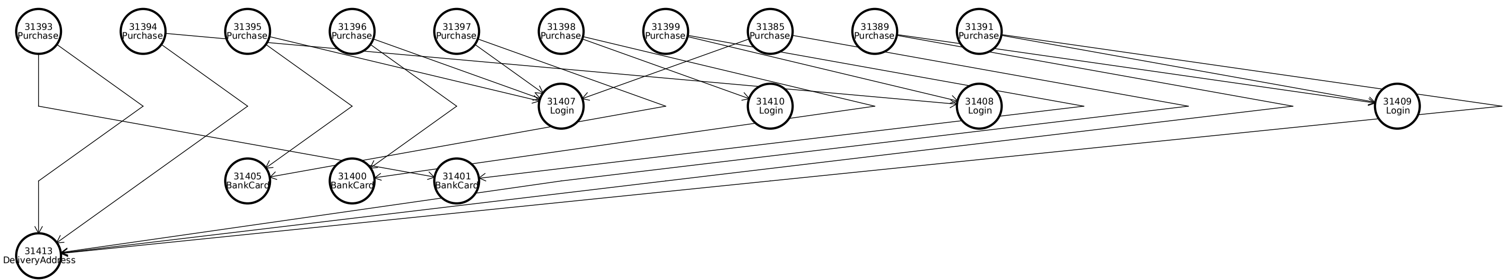



**Figure B.4:** LayeredAlgorithm[LabelLayerAssignment-MedianOrdering-Naive]

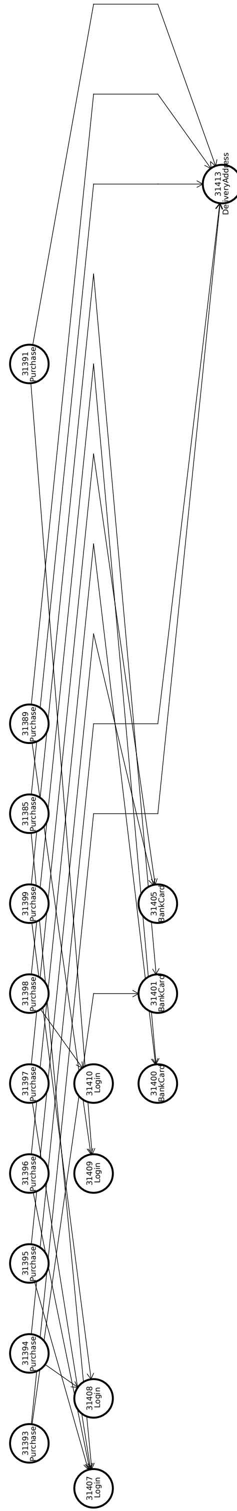



**Figure B.5:** LayeredAlgorithm[LabelLayerAssignment-null-BrandesKopf]

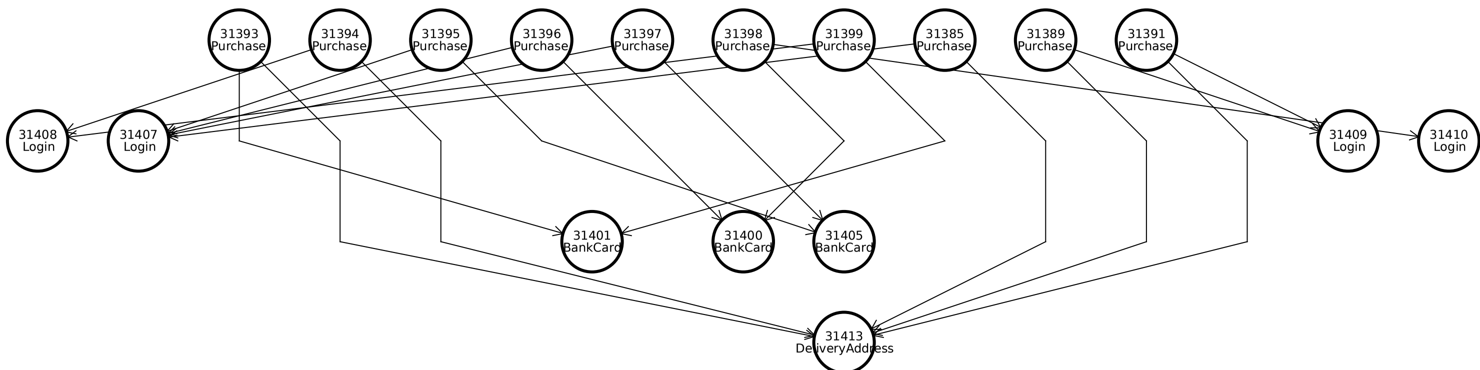



**Figure B.6:** LayeredAlgorithm[LabelLayerAssignment-null-MeanCoord]

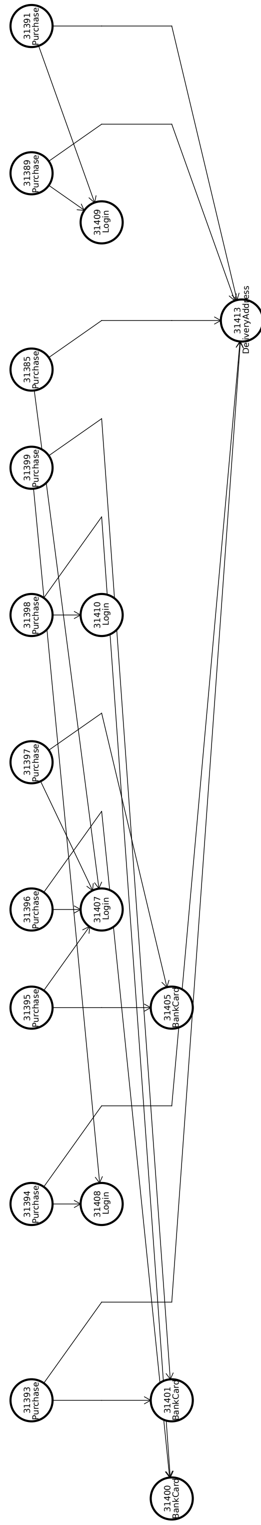



**Figure B.7:** LayeredAlgorithm[LabelLayerAssignment-SwapNeighbors-BrandesKopf]

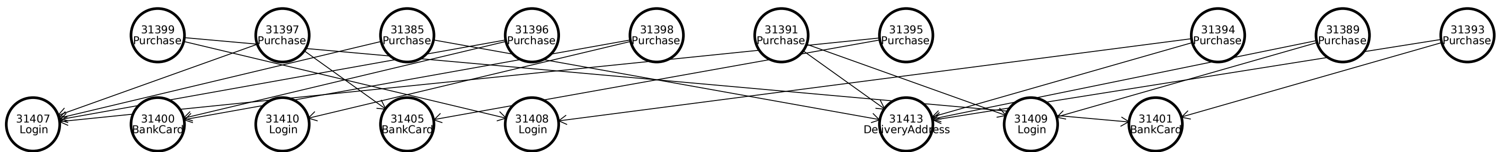



**Figure B.8:** LayeredAlgorithm[LongestPathLayerAssignment-MedianOrdering-BrandesKopf]

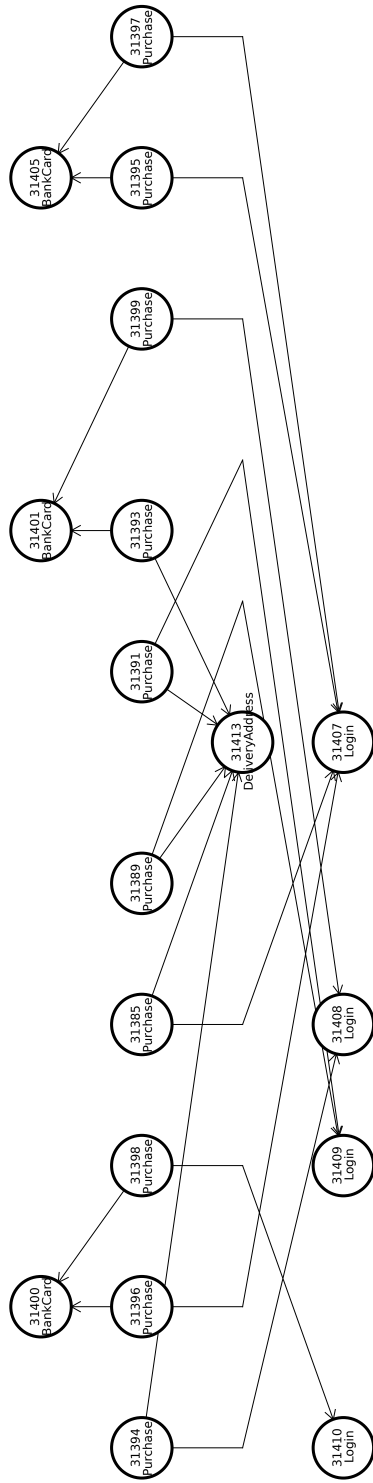



**Figure B.9:** LayeredAlgorithm[PredefLayerAssignment-MedianOrdering-BrandesKopf]

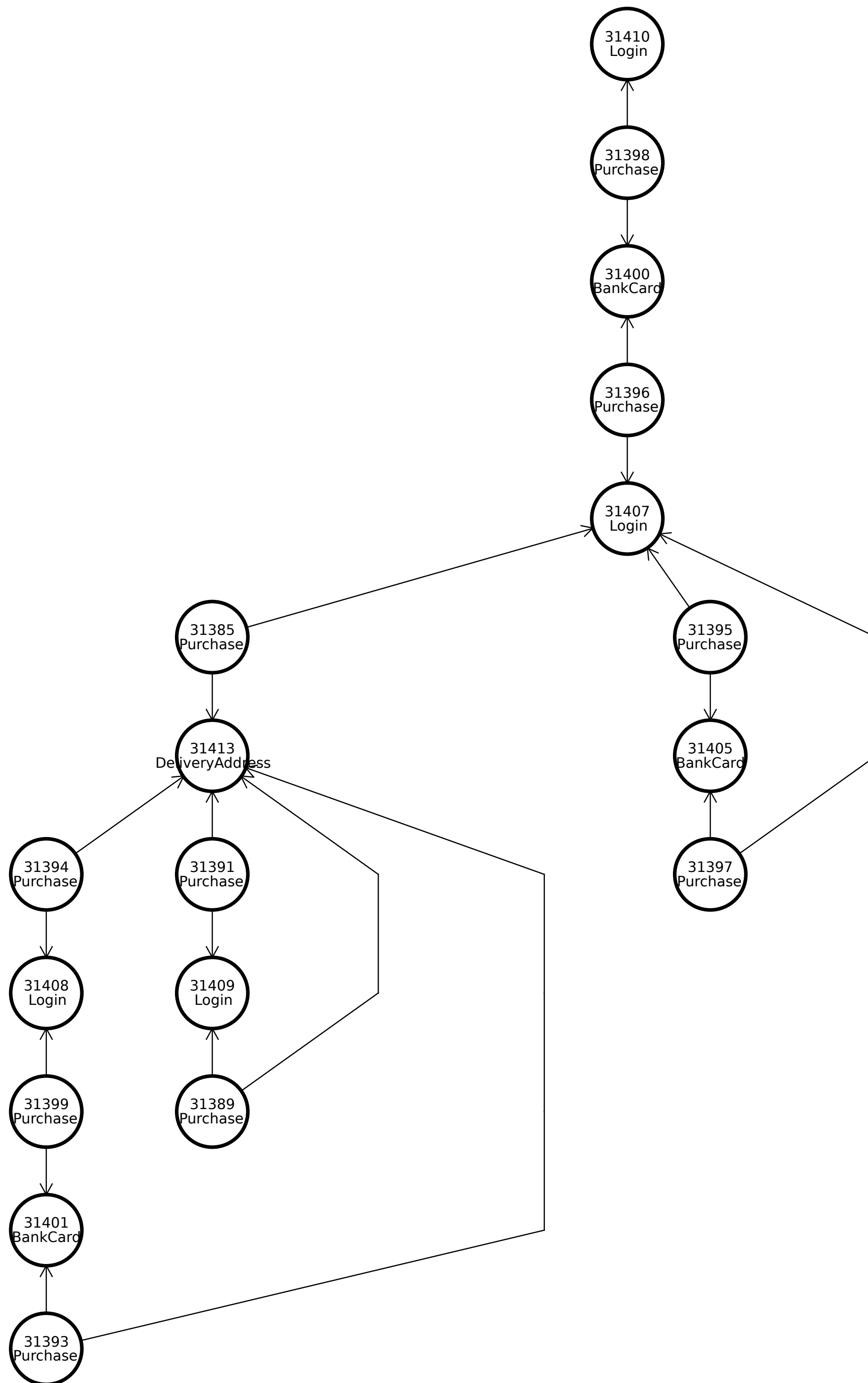



**Figure B.10:** LayeredAlgorithm[QueryPathLayerAssignment-MeanandSwapOrdering-BrandesKopf]

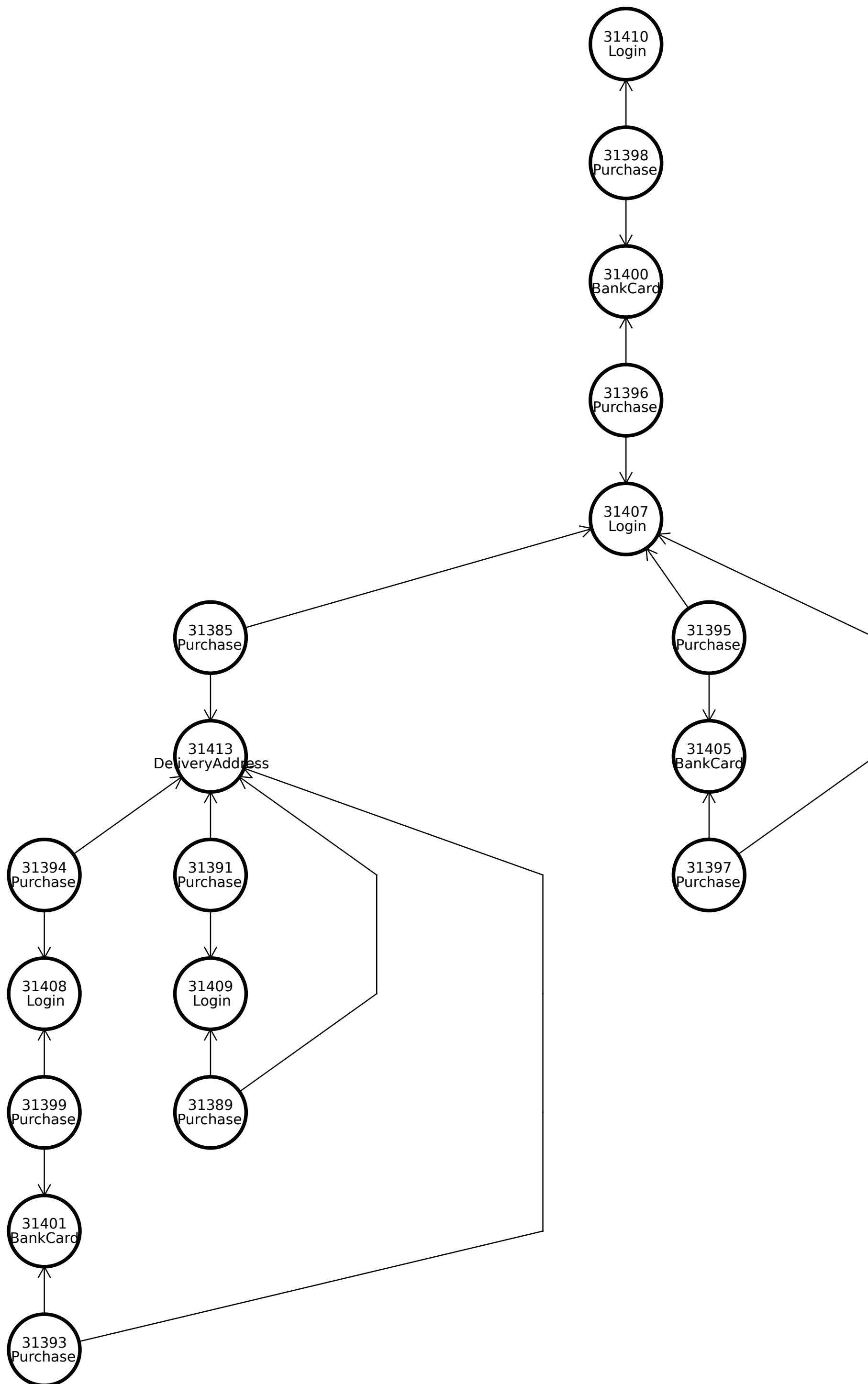



**Figure B.11:** LayeredAlgorithm[QueryPathLayerAssignment-MeanOrdering-BrandesKopf]

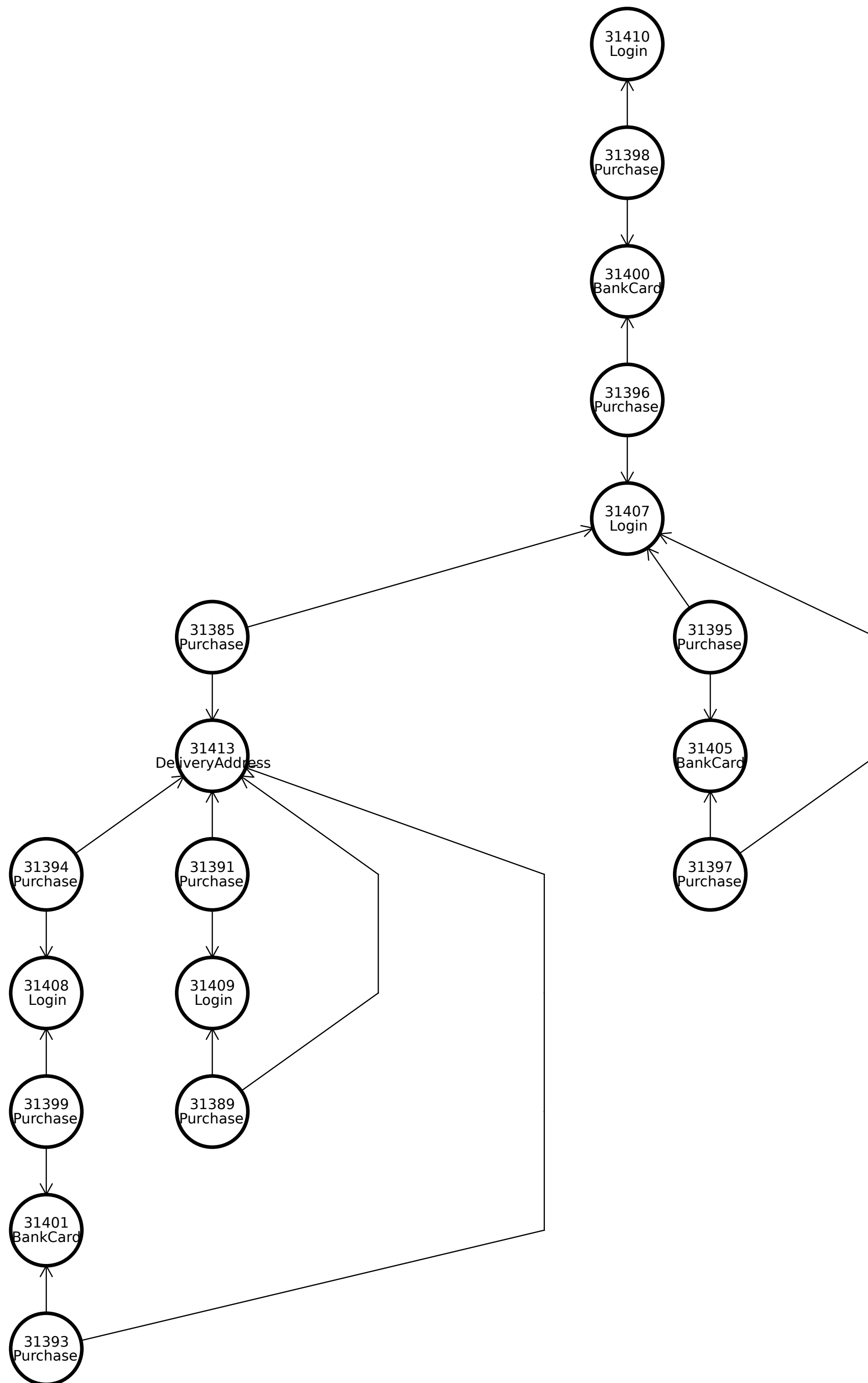



**Figure B.12:** LayeredAlgorithm[QueryPathLayerAssignment-MedianOrdering-BrandesKopf]

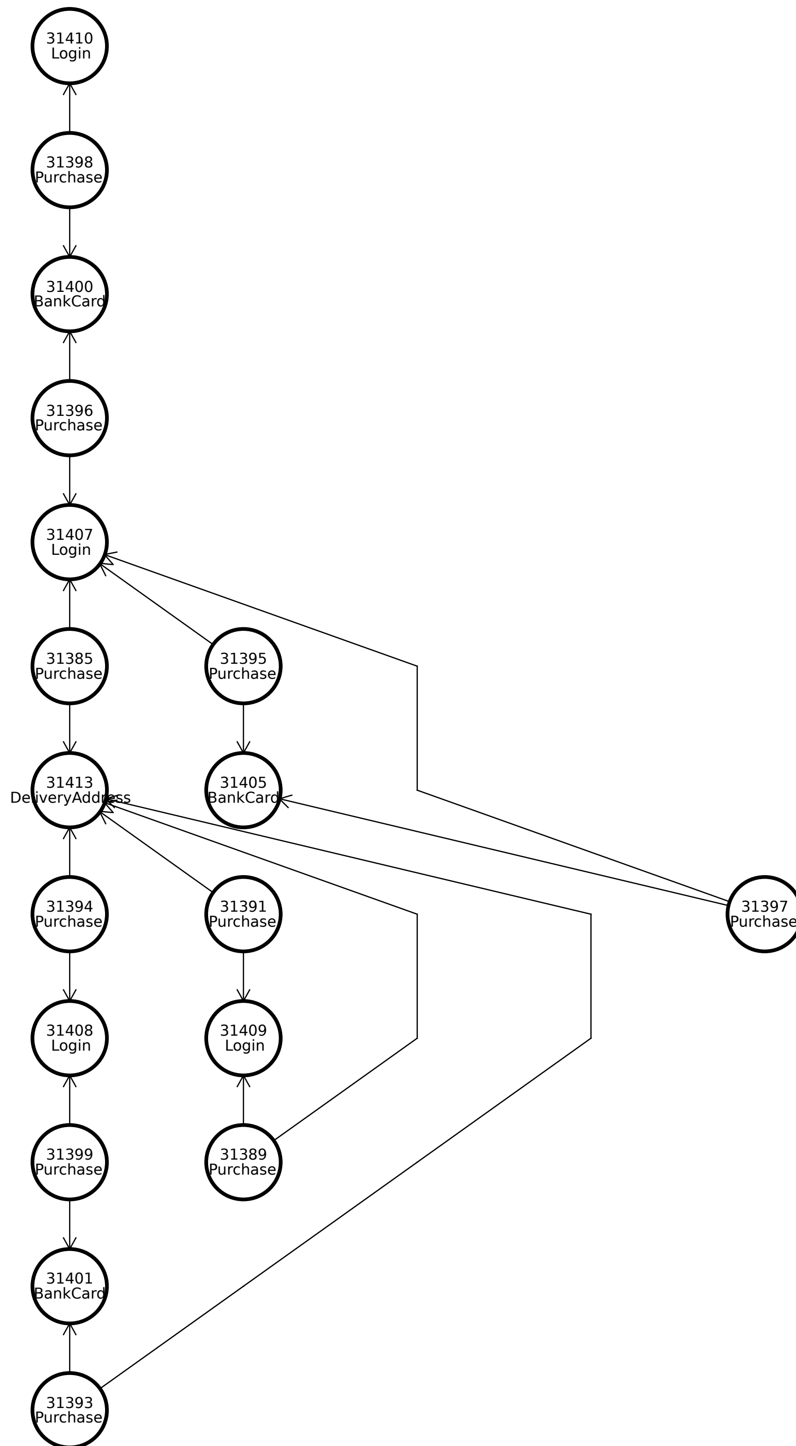



**Figure B.13:** LayeredAlgorithm[QueryPathLayerAssignment-MedianOrdering-Naive]

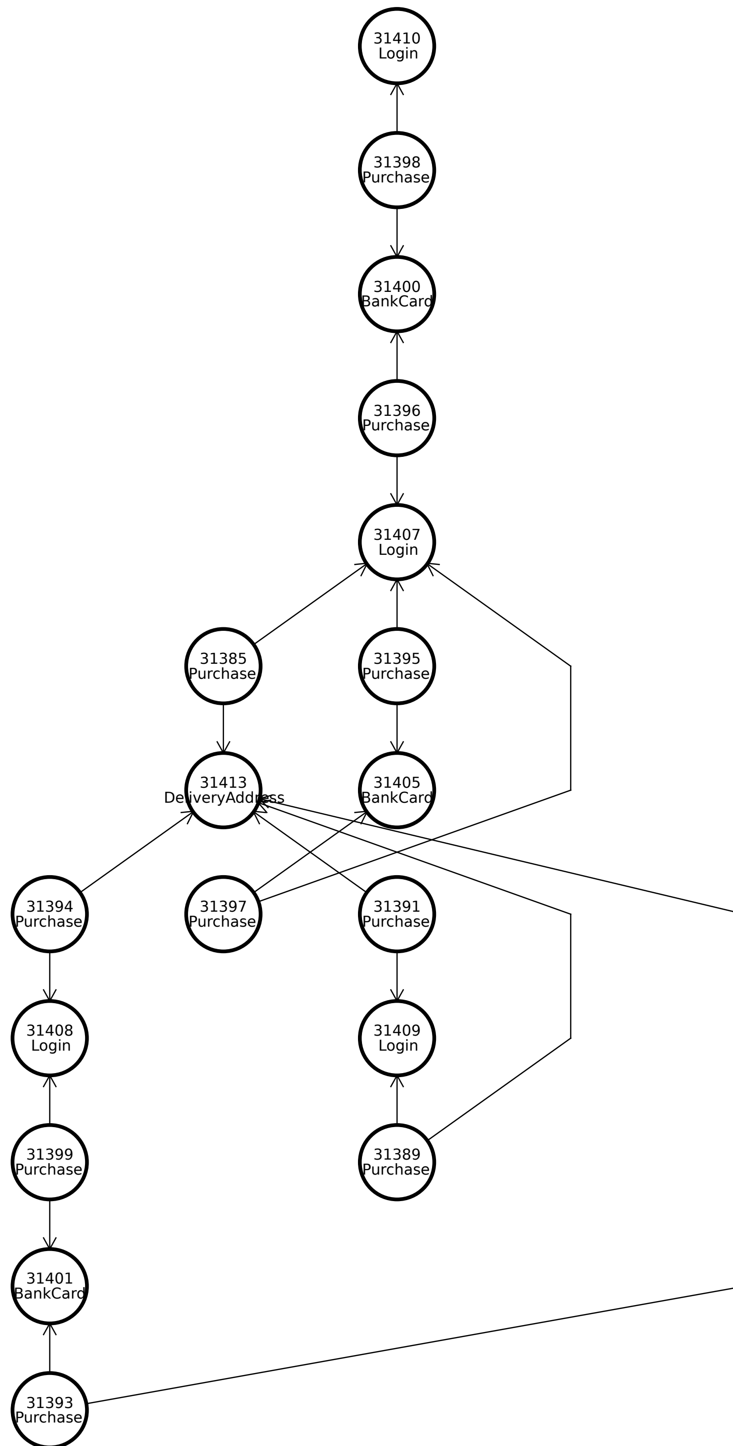



**Figure B.14:** LayeredAlgorithm[QueryPathLayerAssignment-null-BrandesKopf]

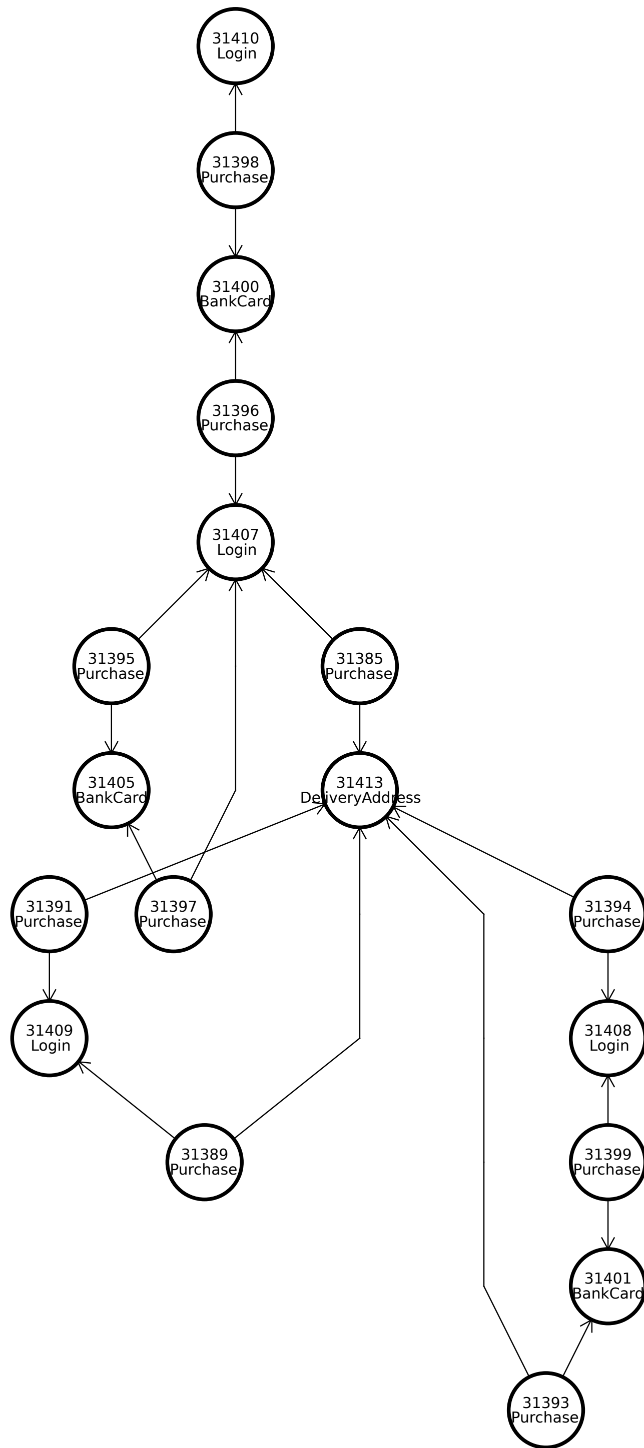



**Figure B.15:** LayeredAlgorithm[QueryPathLayerAssignment-null-MeanCoord]

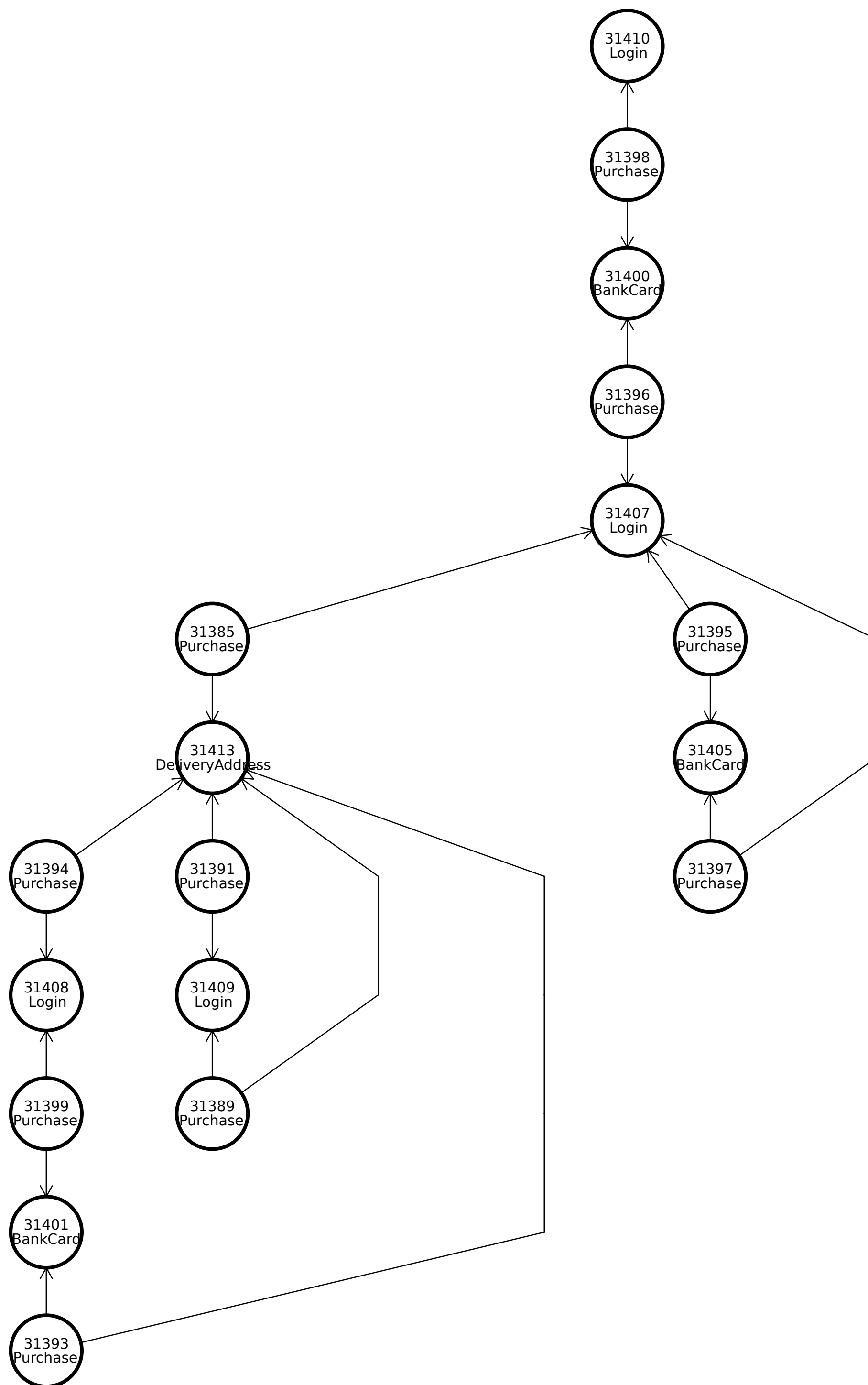



**Figure B.16:** LayeredAlgorithm[QueryPathLayerAssignment-SwapNeighbors-BrandesKopf]

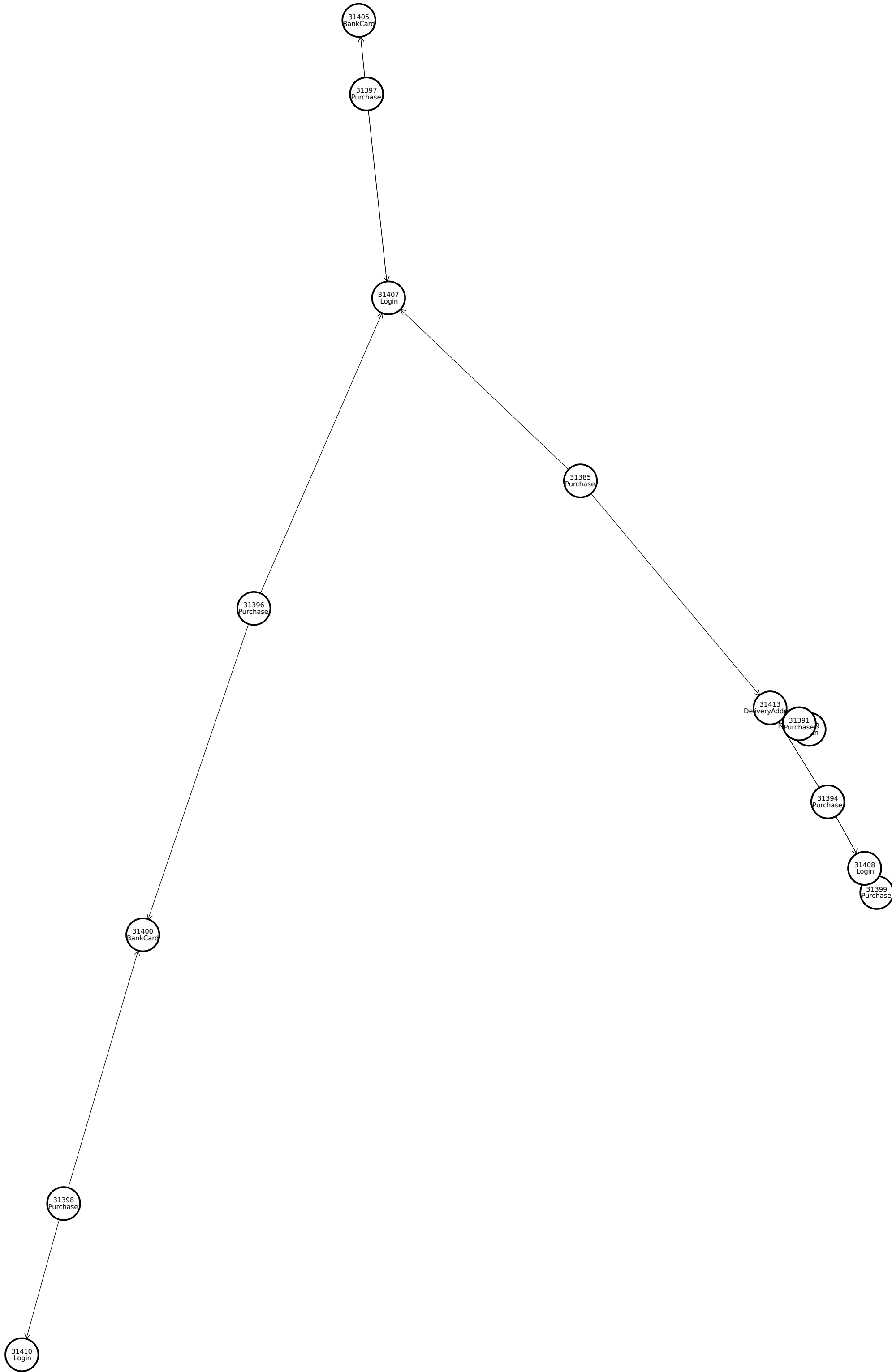



**Figure B.17:** SpectralAlgorithm[Laplacian-DenseSolver]

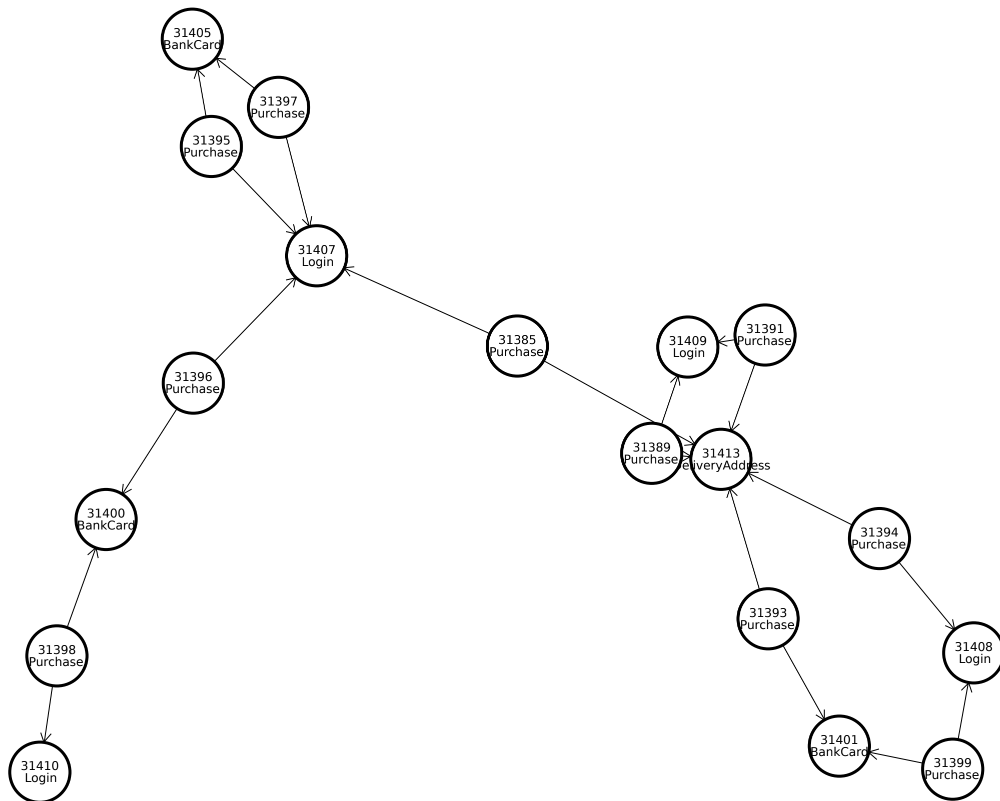



**Figure B.18:** ForceDirected

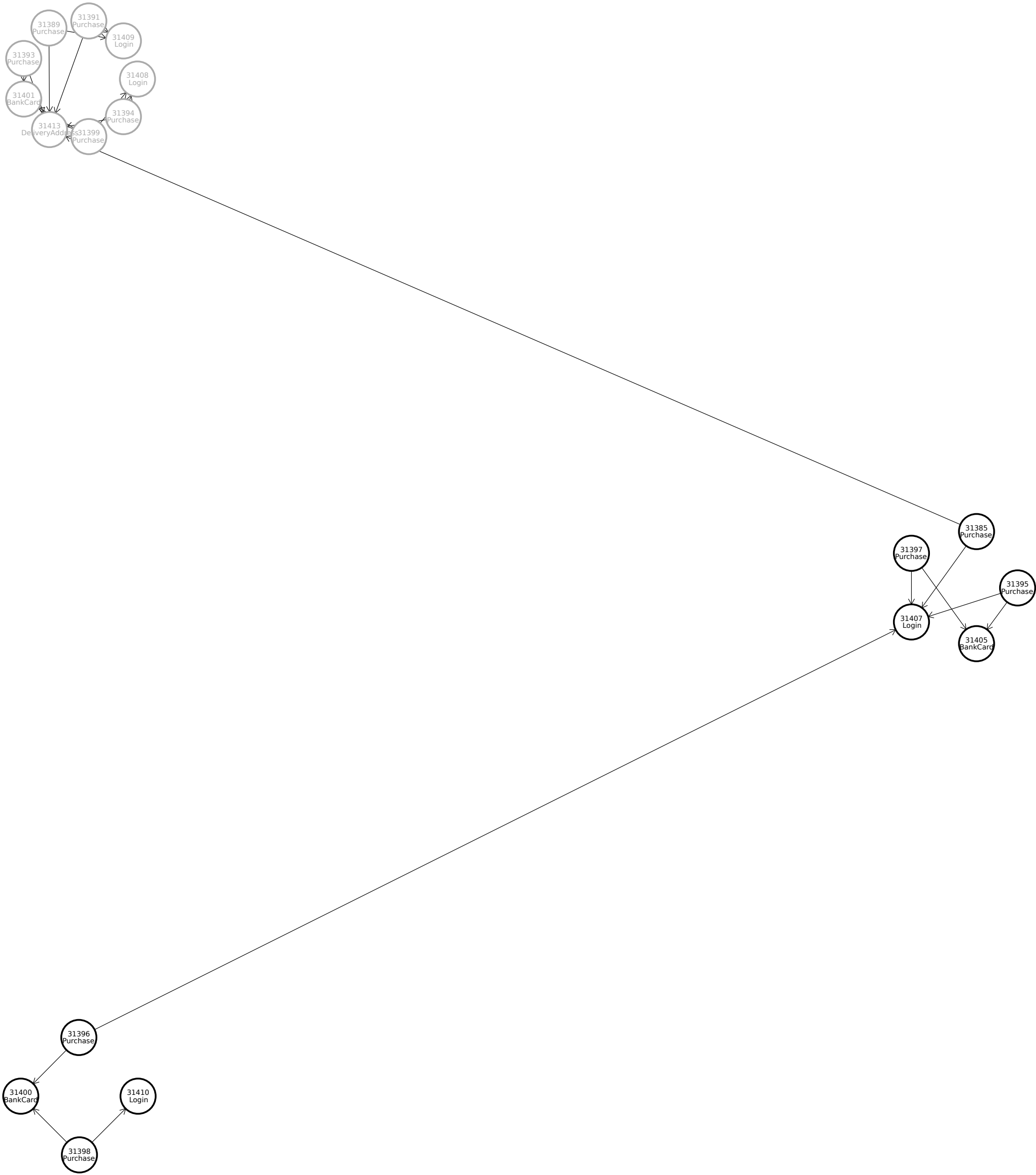



**Figure B.19:** CommunityAlgorithm[Circular-Circular-LouvainImp]

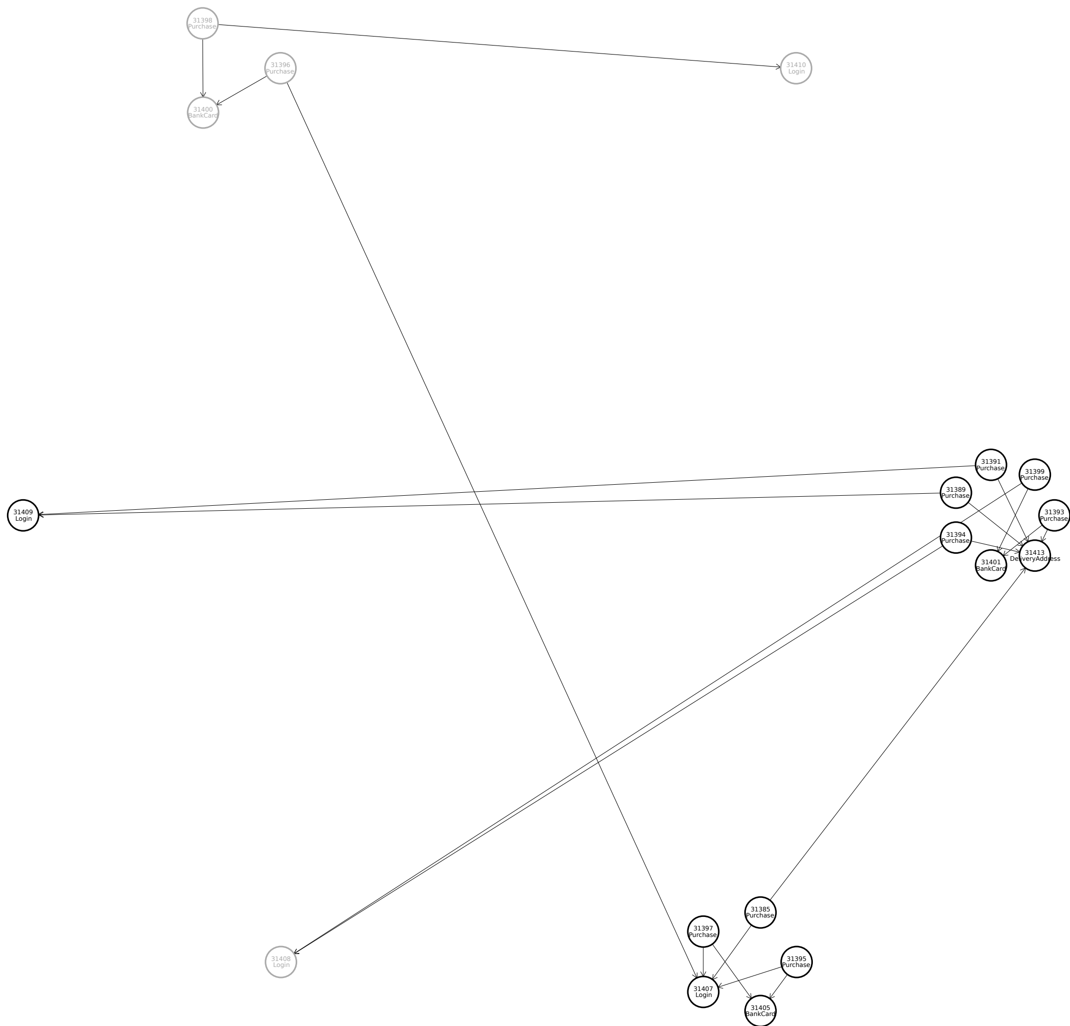



**Figure B.20:** CommunityAlgorithm[Circular-Circular-RandomWalk]

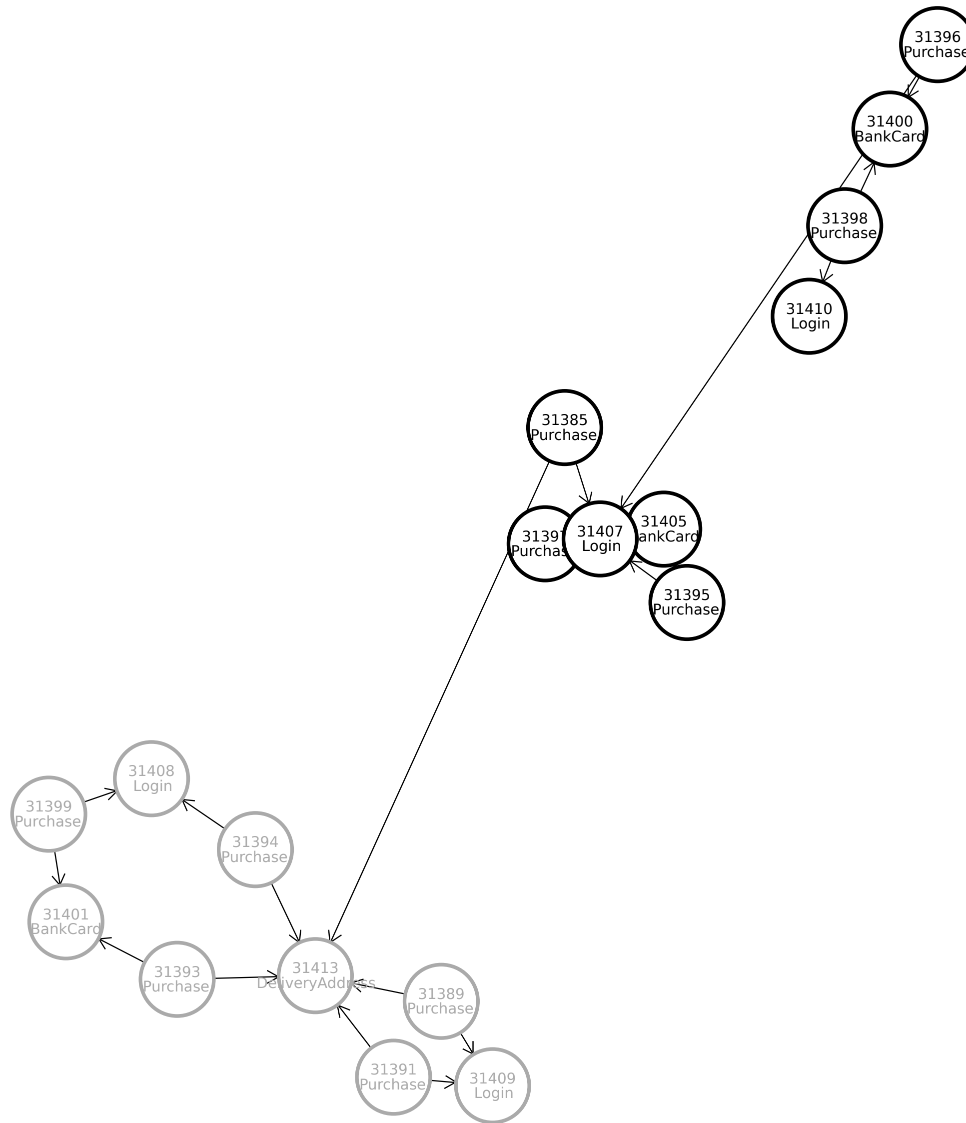



**Figure B.21:** CommunityAlgorithm[Force-Force-LouvainImp]

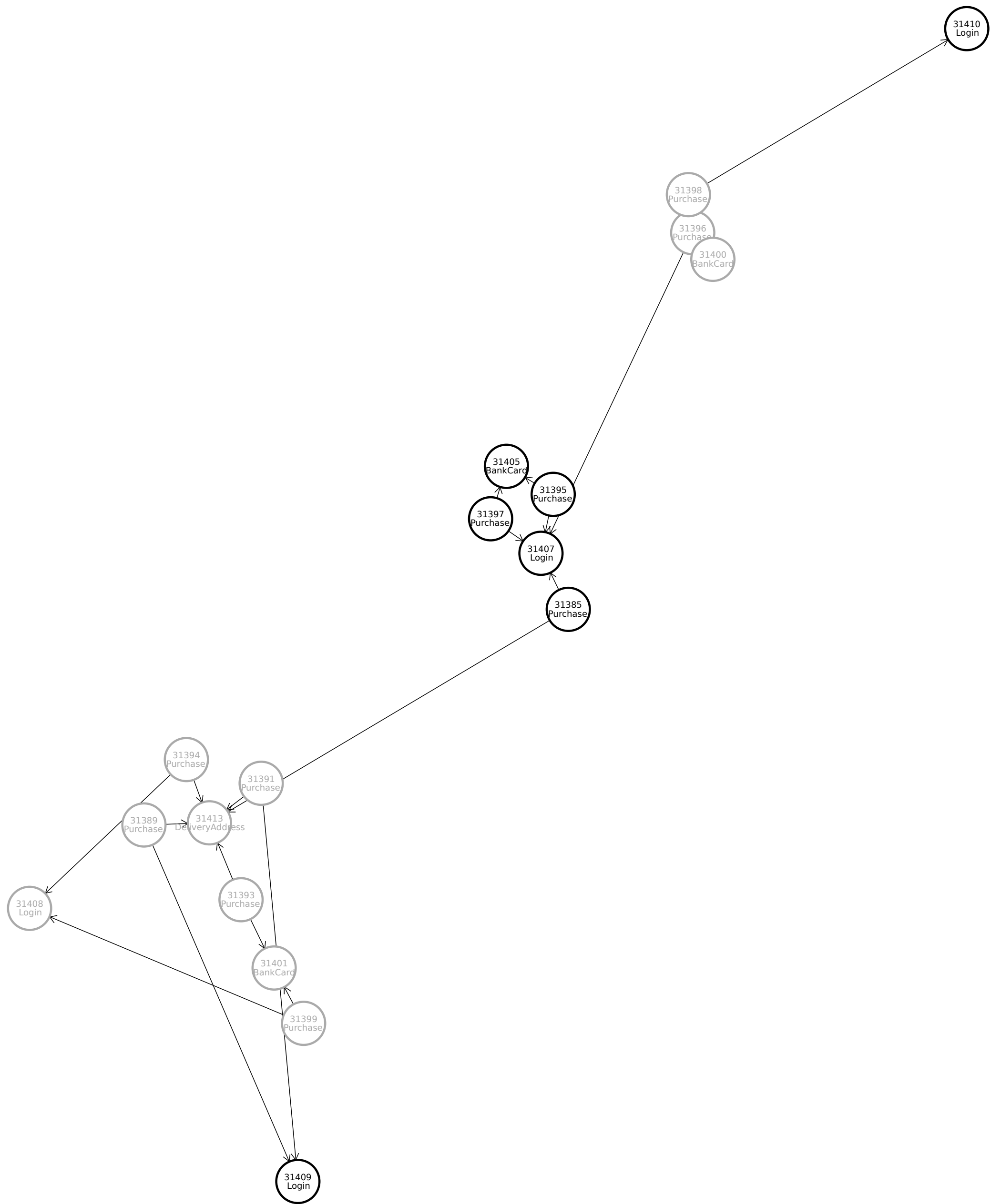



**Figure B.22:** CommunityAlgorithm[Force-Force-RandomWalk]

**EXAMENSARBETE** Graph Layout Methods for Graph Databases - Performance and Analysis
**STUDENTER** Erik Danielsson, Lasse Heemann
**HANDLEDARE** Krzysztof Kuchcinski (LTH)
**EXAMINATOR** Flavius Gruian (LTH)

# Evaluating graph drawing algorithms for graph databases

POPULÄRVETENSKAPLIG SAMMANFATTNING **Erik Danielsson, Lasse Heemann**

Graph databases have become popular in recent years, being used by a multitude of tech companies. But how do you display the data in a meaningful way? We evaluated several approaches, and came up with new ways of visualizing these databases.

Neo4j, a rising tech company, was one of the first to deliver a stable working graph database. They currently provide services to prominent clients worldwide, such as IBM, Volvo, and Comcast. We partnered up with Neo4j to explore alternative ways for their clients to display their data.

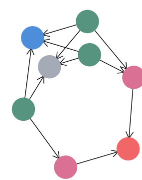
Traditional databases use tables to store data–think of information in address books or excel sheets. The table structure is practical when working with a single table, but things get messy when you need to see information from different places. A graph database instead represents the data as nodes with relationships to one another. For example two people (nodes) connected by a friendship (relationship) in a social media application. Using this structure, related data can be found much faster.

Neo4j currently uses the Force Directed method (shown in Figure 1a), to draw representations of the data. The method is known to have one major flaw; it cannot handle large graphs well. With large amounts of data, the Force Directed method becomes too slow to be usable.

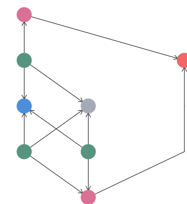
For our master's thesis, we evaluated several alternative methods for drawing graphs. Our hopes were that a different algorithm could produce faster results, or bring something else to the table. We implemented methods such as the *Layered* and *Community*, as well as the *Force Directed* method as a comparison. The performance of the algorithms were analyzed by comparing the resulting images, and by experiments, such as measuring how quickly they could produce a result.

Figure 1: Two methods of drawing the same graph

(a) Force Directed



(b) Layered



During the implementation, we found new ways to perform the layered method (shown in figure 1b). The first method uses additional information, stored in each node, to group the nodes intuitively. The second method gives the user a drawing that better correlates to the given input.

We also found that the community method is able to present groups of strongly related nodes in the graphs. This can be useful in situations such as finding a group of friends in a social media platform.

Overall, our results showed that the Layered and Community method were faster and better visualized than the Force Directed approach. The methods we present help highlight useful approaches to visualizing data for further research.