Comparison and implementation of graph visualization algorithms using JavaFX

Author: Dennis Örnberg
Date: 2016-12-04

Examiner: Patrick Lambrix
Advisor: Valentina Ivanova
Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances. The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.
Abstract

Graph drawing is an important area in computer science and it has many different application areas. For example, graphs can be used to visualize structures like networks and databases. When the graphs are really big, however, it becomes difficult to draw them so that the user can get a good overview of the whole graph and all of its data. There exist a number of different algorithms that can be used to draw graphs, but they have a lot of differences. The goal of this report was to find an algorithm that produces graphs of satisfying quality in little time for the purpose of ontology engineering, and implement it using a platform that visualizes the graph using JavaFX. It is supposed to work on a visualization table with a touch display. A list of criteria for both the algorithm and the application was made to ensure that the final result would be satisfactory. A comparison between four well-known graph visualization algorithms was made and “GEM” was found to be the best suited algorithm for visualizing big graphs. The two platforms Gephi and Prefux were introduced and compared to each other, and the decision was made to implement the algorithm in Prefux since it has support for JavaFX. The algorithm was implemented and evaluated, it was found to produce visually pleasing graphs within a reasonable time frame. A modified version of the algorithm called GEM-2 was also introduced, implemented and evaluated. With GEM-2, the user can pick a specific number of levels to be expanded at first, additional levels can then be expanded by hand. This greatly improves the performance when there is no need to expand the whole graph at once, however, it also increases the amount of edge crossings which makes the graph less visually pleasing.
Acknowledgements

I would like to thank my supervisor Valentina Ivanova and my examiner Patrick Lambrix for helping me get started in this area of research that I initially knew very little about, and for giving me advice whenever I felt like I got stuck. Being able to use ADIT’s multi-touch display to test the touch functionality was also extremely helpful, perhaps even necessary to make everything work.
# Table of contents

1 Introduction 
   1.1 Background............................................................................................................. 1  
   1.2 Purpose................................................................................................................ 1  
   1.3 Problem.............................................................................................................. 2  
   1.4 Method................................................................................................................ 2  
   1.5 Additional Application Requirements............................................................... 3  
   1.6 Limitations......................................................................................................... 3  
   1.7 Outline............................................................................................................... 4  

2 Background and Requirement Analysis  
   2.1 Graph Drawing Theory............................................................................................ 5  
      2.1.1 Basics, Measures and Distances................................................................. 5  
      2.1.2 Algorithms.................................................................................................... 7  
         2.1.2.1 KK (Kamada / Kawai)......................................................................... 7  
         2.1.2.2 FR (Fruchterman / Reingold)............................................................. 8  
         2.1.2.3 GEM (Frick / Ludwig / Mehldau)..................................................... 9  
         2.1.2.4 DH (Davidson / Harel)..................................................................... 10  
      2.1.3 Analysis of Graph Visualization Algorithms........................................... 11  
         2.1.3.1 Related work....................................................................................... 11  
         2.1.3.2 Review................................................................................................. 11  
   2.2 Software Platforms............................................................................................... 13  
      2.2.1 JavaFX, Touches and Gestures................................................................... 13  
      2.2.2 Graph Visualization Libraries................................................................. 14  
         2.2.2.1 Gephi................................................................................................. 14  
         2.2.2.2 Prefux................................................................................................. 14  
      2.2.3 Analysis of Graph Visualization Libraries.............................................. 14  
   2.3 Ontology Engineering and Alignment................................................................ 16  

3 Design and Implementation  
   3.1 Design................................................................................................................ 17  
   3.2 Implementation................................................................................................... 18  
      3.2.1 Setup......................................................................................................... 18  
      3.2.2 GemMain.................................................................................................... 19  
      3.2.3 GraphEmbedderLayout............................................................................ 21  
      3.2.4 GemControl............................................................................................... 24  
      3.2.5 ArrowRenderer.......................................................................................... 25  
   3.3 GEM-2............................................................................................................... 25  

4 Evaluation  
   4.1 Setup.................................................................................................................. 27  
   4.2 Result.................................................................................................................. 29  

5 Discussion  
   5.1 Algorithm............................................................................................................ 30  
   5.2 Application........................................................................................................... 32  
   5.3 Method................................................................................................................ 32  
   5.4 Possible Improvements and Changes................................................................. 33
List of Figures

Figure 1: An example of a waterfall model. (Source: W. Royce, 1970 [13]) 2

Figure 2: Gestures for moving, zooming and rotating the graph. (Source: Microsoft, 2016 [14]) 3

Figure 3: A symmetric drawing (a) and a planar drawing (b). (Source: Kamada et al., 1989 [11]) 6

Figure 4: A (very) basic overview of how Prefux works. 17

Figure 5: An overview of where the algorithm and the new functionalities will fit in. 19

Figure 6: An overview of Ont-1 (3,696 nodes, 3,693 edges) with a desired temperature of 3. 41

Figure 7: An overview of Ont-2 (6,488 nodes, 4,917 edges) with a desired temperature of 16. 42

Figure 8: An overview of Ont-3 (10,157 nodes, 10,154 edges) with a desired temperature of 32. 43

Figure 9: An overview of Ont-4 (13,412 nodes, 16,281 edges) with a desired temperature of 128. 44

Figure 10: The difference between Setup 2.1–2.4 on Ont-1. 45

Figure 11: The difference between Euclidean / Manhattan distance on Ont-1. 46

Figure 12: The difference between different temperature thresholds on Ont-1. 47

Figure 13: Gem-2, collapsed and expanded. 48
List of Tables

Table 1: Benchmark for different values for temp and maxTemp.  49
Table 2: Benchmark for different values for desiredEdgeLength and gravitationalConstant.  50
Table 3: Benchmark for different values for the opening angles.  51
Table 4: Benchmark for different values for euclideanDistance.  52
Table 5: Benchmark for different values for desiredTemp.  53
Table 6: Benchmark for the different ontologies.  54
1 Introduction

This report is the result of a Bachelor final year project in computer engineering performed at the Division for Database and Information Techniques (ADIT) at Linköping University. The purpose of this thesis is to do a comparison of different graph visualization algorithms in the context of ontology engineering, and then implement one of them using JavaFX on a suitable platform. The implemented algorithm and application will be usable on a visualization table with a touch screen.

1.1 Background

The term “ontology” comes from philosophy where it is the study of the kinds of things that exists [1]. In the context of computer and information science, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse [2]. There are different languages that can be used to describe ontologies, such as OWL and RDF. They can be represented in a number of different ways, for example as an indented list or a directed graph, and they can have complex relationships. Ontologies within the same domain can be very different, especially if they have been developed by different people and organizations. Ontology alignment is the process of finding similarities between different ontologies, and an alignment is a set of relations between two different ontologies [3]. This is an important area of research because it helps unifying different ontologies within the same domain. One of the challenges in building ontology alignment systems is finding ways to support the users in the alignment process [4], and few systems have tackled this challenge [5]. Visualization of the ontologies and the mappings between the ontologies is one of the major requirements for systems [6,7].

Another important area of research in computer science is graph drawing. It can be used in a lot of different application areas, for example databases, VLSI technology, production planning, chemistry and biology [8]. Historically, graphs have been drawn manually when the number of items in the graphs have been small [9]. As graphs become larger, the manual approach becomes more and more difficult, and therefore several different graph visualization algorithms have been introduced to make it easier to visualize graphs of varying sizes. One problem is that these graphs can get very large and it is sometimes not feasible to look at the entire graph even when using a big display with a high resolution. Another problem is that the graph visualization algorithms usually have high time complexities, which means that the time it takes to finish running the algorithm increases when the graphs get bigger [9,10,11,12]. These algorithms all have different characteristics and features, and the resulting graphs can look very different from each other.

1.2 Purpose

The main vision behind this work is to successfully create an application that can visualize huge graphs in a satisfactory manner, and also identify and solve any eventual pitfalls that might show up along the way. After all, implementing an algorithm is not a trivial task since the context in which it is implemented has to be taken into consideration. The platform must be chosen carefully to make sure that the chosen algorithm can be implemented without too much hassle. The ultimate goal is to make the graphs look as aesthetically pleasing as possible, while making sure that it does not take too much time producing them.
1.3 Problem

- Which graph drawing algorithm is the best suited for visualizing huge graphs for the purpose of ontology engineering?
- How can the chosen algorithm be implemented in an efficient way?

1.4 Method

The waterfall model was chosen as the software engineering methodology to be used in this report. W. Royce's [13] paper from 1970 is generally considered to be the first formal description of the waterfall model, although the term “waterfall” is not used. The model consists of a number of different steps which should be followed, and a step has to be completed before the next can be started. Many different variations of this model are used today. The figure below shows one model that Royce described in his paper:

![Waterfall Model Diagram](source)

*(Figure 1: An example of a waterfall model. (Source: W. Royce, 1970 [13])*)

Using the waterfall model, a lot of time is usually spent at the requirement and analysis steps. It is often better to find problems early, before the implementation has started. A drawback of the model is that it takes a long time to reach the testing step where the first results of the implemented application can be measured. If the results are not satisfactory, it could be necessary to go back all the way to the requirement steps.
1.5 Additional Application Requirements

A list of criteria for the application is also needed. One criterion is that the user should be able to move the graph, zoom in and out and also rotate the graph using a multi-touch display.

- Movement: The user should be able to move the graph by placing a finger anywhere on the graph and then drag it.
- Zooming: The user should be able to zoom in and out on the graph by pinching anywhere on the graph with two fingers.
- Rotation: The user should be able to rotate the graph in both directions by placing two fingers anywhere on the graph and then do a rotating gesture.

The user should also be able to collapse and expand nodes. Collapsing a node will make all its children invisible and expanding a node will make them visible again. This functionality can greatly help the user interact with very large graphs, since it enables the user to only display the part of the graph that is interesting and ignore everything else. The nodes must have a reasonable size. They cannot be too big because nodes can then overlap and the whole graph will not be visible at once, and they cannot be too small because then they will not be visible when the user zooms out. The same goes for the node-labels, they should be placed near the center of each node and they should be clearly visible, even when zoomed in or out. This could be done by resizing the labels automatically when the user zooms in or out, this way the labels will always have the same size.

In addition to these criteria, the procedure must be reasonably fast as well. The run time could be very different depending on the hardware used for the computations, and some of the algorithms could for example be optimized for multithreading. The amount of memory used by the algorithms will not be considered in this report, since all the algorithms are predicted to consume similar amounts of memory. This is because the algorithms are fundamentally similar to each other in many ways and they do not, at least in their original implementations, use any special caching mechanisms that would increase the memory usage by any significant amount.

1.6 Limitations

The main focus will be on graph visualization algorithms rather than data representation. Only the
“is-a” hierarchy is visualized. An “is-a” relationship is the subclass relationship between a class and its parent. All the graphs in this report are two-dimensional straight-line drawings without cycles and with directed edges. A straight-line drawing means that every edge is drawn as a straight line; there are no curved edges. Algorithms with the purpose of making a graph aesthetically pleasing will be considered for this report. There are other algorithms, for example some VLSI layout techniques, that do not focus on making the graph aesthetically pleasing and they are not relevant for this study. Interacting with the finished application will primarily be done using touch and not mouse and keyboard. The multi-touch display that the application will be tested on is a MT550S MultiTaction Cell 55”. There are different definitions of what a huge graph really is. For example, Frick et al. [10] defined a huge graph as a graph that contains more than 128 nodes. In this report, the goal is to visualize huge graphs with several hundreds or even thousands of nodes.

1.7 Outline

The first step is to do a theoretical comparison of some of the most well-known graph visualization algorithms and select the one that seems to be the best according to a list of criteria. The second step is to choose the platform to be used for implementing the algorithm. The third step is to implement the algorithm in an application that can be used on a visualization table.

The report is structured as follows: Chapter 2 contains graph drawing theory, information about different software platforms used for visualizing graphs and background information about ontology engineering and alignment. It contains the requirement and analysis steps from the waterfall model displayed in figure 1. Chapter 3 describes the design phase and the implementation of the algorithm and the rest of the functionality needed, it contains the design and coding steps from figure 1. Chapter 4 contains the tests that were made to evaluate the algorithm and the application, it has the testing step from figure 1. Chapter 5 is the discussion about the results, along with possible changes and improvements that can be made. Chapter 6 contains concluding remarks.
2 Background and Requirement Analysis

This chapter contains relevant background and an analysis of the algorithms and the software platforms. Section 2.1 provides relevant background about graphs in general, the criteria for drawing graphs, different methods that can be used to draw graphs, four different algorithms, previous work related to the comparison of these algorithms along with a review. Section 2.2 gives an introduction to JavaFX and how touches and gestures work. Two software platforms, Gephi and Prefux, are also introduced and compared to each other. Section 2.3 contains relevant information about ontology engineering and alignment.

2.1 Graph Drawing Theory

2.1.1 Basics, Measures and Distances

A graph is a set of nodes (or vertices) that are connected by a set of edges (or links), its purpose is to describe relationships between entities. The nodes can have attributes attached to them with any application-related information [15]. Graphs are usually classified into directed and undirected graphs. If the edges have a direction associated with them, it is a directed graph. If the edges are bidirectional or have no direction, it is undirected. A cyclic graph is a graph that contains at least one cycle, and an acyclic graph contains no cycles. This is a property that both directed and undirected graphs can have. A connected graph is a graph where there is a path between every pair of nodes. If there is at least one pair of nodes with no path between them, the graph is disconnected. A graph can be called a small-world graph if any node can be reached from a randomly chosen node with a small number of steps, all the nodes do not have to be neighbors.

Many different algorithms can be used to draw graphs, but the time complexities are often high which can lead to bad performance when large graphs are being drawn [9,10,11,12]. Another problem related to the evaluation of graphs is that there is no standard-set of graphs by which to judge on the quality of the drawings [10]. This could in some cases make the evaluation somewhat subjective. The goal is to find an algorithm that is “good enough” for drawing huge general graphs in reasonable time. In order to evaluate if a graph is good or not, a list of criteria is needed. Kamada et al. [11] said that it is difficult to find common criteria for general graphs since they can have all kinds of different structures. They describe two general criteria for creating an aesthetically pleasing graph:

- Nodes and edges should be distributed evenly.

If some nodes are very far away from others, it would make the graph unnecessarily large. If all the nodes could be grouped together it would make the graph easier to understand. At the same time, the nodes should not be too close together because that would make the graph look cluttered. Another way to put it is that all the edges should have a similar length, but they do not have to be perfectly uniform.

- The number of edge crossings should be minimal
If a graph is drawn so that there are no edge crossings, it is a “planar” graph. But minimizing edge crossings is not always optimal, since some graphs are more easy to understand with crossings than without. As an example, the figure below shows one symmetric and one planar drawing of a graph:

![Figure 3: A symmetric drawing (a) and a planar drawing (b). (Source: Kamada et al., 1989 [11])](image)

We can see that an even distribution of nodes with some edge crossings can be more aesthetically pleasing and easier to understand than a planar graph. Also, the bigger a graph is, the harder it is to prevent edge crossings. The goal should be to minimize the number of edge crossings, but not to the extent that possible symmetries disappear. This would imply another criterion:

- Symmetry should be displayed if possible.

According to Di Battista et al. [16], the display of symmetries is desirable in all graphic standards, and Kamada et al. [11] has said that symmetry is a valuable characteristic for structures. There is however a conflict between the desire to display symmetries and minimizing the number of edge crossings [9]. H. Purchase [17] has done research about which aesthetic has the greatest effect on human understanding. According to her research, the most important criterion is to minimize the number of edge crossings. Displaying symmetries is not nearly as important, as it did not have as big of an effect.

Davidson et al. [9] described a criterion that they call “node-edge distances”. The idea is to separate nodes that are really close and move them further away from each other, regardless if they are connected or not. This turned out to be computationally heavy so they decided to not include it in the default settings of their main algorithm. Nevertheless, distances between nodes is an important aspect of graph drawing and it is something that will have to be calculated often. There are multiple ways to calculate distances between two points in a coordinate system, the two most basic ways are the Euclidean distance and the Manhattan distance. The Euclidean distance is the straight line distance between two points in a Euclidean space. It is calculated by using the Pythagorean theorem:

\[ distance = \sqrt{(\Delta x)^2 + (\Delta y)^2} \]

In contrast, the Manhattan distance is a more simple way to calculate distances. In a two-dimensional space, only the distances up-down and left-right can be measured using the Manhattan distance. It is calculated by simply adding together the differences in x- and y-coordinates:
\[ \text{distance} = |\Delta x| + |\Delta y| \]

Which one is better depends entirely on the context in which they are used. If it is important to be able to measure diagonal distances accurately, the Euclidean distance is a better choice. It is however more computationally expensive than calculating the Manhattan distance.

### 2.1.2 Algorithms

There exist a couple of different heuristics that can be used for constructing straight-line drawings of directed and undirected graphs. This report will focus on two of them, the spring embedder and simulated annealing.

The spring embedder, also known as a force-directed algorithm [10], is a heuristic that is based on a physical model. The edges are seen as springs to indicate that there is an attraction between the two nodes that are linked together. If the nodes are far apart, the springs should attract them to bring them closer together and on the other hand, if the nodes are too close to each other, they should repel and spread out [16]. Graphs drawn with this technique tend to be aesthetically pleasing and, even though they do not explicitly support the detection and display of symmetries, the graphs often turn out to be symmetrical to a certain degree anyway [10]. Given a graph with nodes placed in arbitrary places, spring embedder-algorithms typically start off by iterating over the system while calculating the forces between the nodes and updating their positions accordingly. This iteration either continues for a fixed number of times or until a specific condition has been met, or both [10].

Simulated annealing is also based on a physical model. Annealing is the process of heating up a material past its melting point and then slowly cooling it down to change the materials properties. Simulated annealing uses a temperature parameter. When the temperature is high, the algorithm is more likely to accept solutions that are worse than the current solution, it allows “uphill” moves. The temperature then decreases to lower the chance of uphill moves and to focus on the area of the search space that is close to the global optimum [9].

The problem with a spring embedder is that it might find a local optimum instead of a global optimum. When the search space is considered to be too big for an exhaustive search, simulated annealing can instead be used to try to locate the best possible solution. It is however known that simulated annealing is a time-consuming procedure [9].

As already mentioned, the size of the graph is very important to consider when testing different algorithms. This report will use the following graph size convention: graphs with 5,000 nodes or less are small, 10,000 are medium, 15,000 are large and anything above are huge. Frick et al. [10] distinguished between sparse, normal and dense graphs depending on how many edges there are in relation to the number of nodes. A sparse graph has less edges than nodes, a normal graph has less than three times as many edges as nodes and a dense graph has at least three times as many edges as nodes.

#### 2.1.2.1 KK (Kamada / Kawai)

In 1989 Kamada et al. [11] introduced an algorithm for drawing general undirected graphs, it will be referred to as “KK” in this report. This algorithm uses the spring model to bring the nodes that
are too spread out closer together. The desirable Euclidean/geometric distance between two nodes is called the “graph theoretic” distance. This becomes the desirable length of the spring that connects these two nodes in the corresponding graph. The optimal layout is when the total energy of all the springs is minimal.

Experiments have shown that the positions of the nodes in the initialization is not important, unless in a special case e.g. all the nodes lie on a straight line. In a simple initialization the nodes are placed on the nodes of a regular n-polygon. After the nodes have been positioned, the total energy is decreased by moving one node at a time to a more stable position. In order to reduce the total energy, the algorithm solves a system of partial differential equations by using the Newton-Raphson method. A simple pseudo-code summary could look like this:

• Find the shortest distance between every pair of nodes.
• Find the optimal length of the spring between every pair of nodes.
• Calculate the strength of the spring between every pair of nodes.
• Initialize the graph and place the nodes in their initial positions.
• While there exists an edge with an energy > ϵ (where ϵ is a predefined value):
  ○ Select the node that is the furthest away from its optimal position.
  ○ While the current node is too far away from its optimal position:
    ▪ Solve the linear equations to get the direction, $\delta x$ and $\delta y$.
    ▪ Move the node towards its new position.

The time complexity for finding the shortest distance between every pair of nodes is preliminary $O(n^3)$, although that could be reduced by using a more efficient algorithm for large graphs. In the inner while-loop, $O(n)$ time is required to check if the node is too far away from its optimal position and to calculate $\delta x$ and $\delta y$ respectively. In the outer while-loop, $O(n)$ time is required to find the node that is the furthest away from its optimal position. As a result, the iterative stage takes $O(T \cdot n)$ to finish, where $T$ is the number of times the inner loop is executed. It is difficult to estimate $T$ because it depends on the number of nodes and edges and also the initial positions of the nodes.

2.1.2.2 FR (Fruchterman / Reingold)

Fruchterman et al. [12] introduced an algorithm for drawing undirected graphs with straight edges in 1991, it will be referred to as “FR” in this report. It is a modified spring embedder and the goal of the implementation is speed and simplicity. It uses a global temperature parameter, similar to what can be found in simulated annealing. The distance moved is bounded by this temperature parameter and is decreased every iteration to make the displacements smaller and the adjustment finer.

The algorithm uses five criteria: distribute the nodes evenly, minimize edge crossings, make edge lengths uniform, reflect inherent symmetry and conform to the frame. The authors have two principles for graph drawing: nodes that are connected should be drawn close to each other, but nodes should not be drawn too close to each other. Basically, nodes that are connected are attracted to each other while nodes that are not connected repel each other. A simple summary of how the algorithm works could look like this:
• Place the nodes in random positions within the frame.
• Repeat the following (a fixed number of times):
  ◦ For every node in the graph:
    ▪ Calculate the repulsive forces between the current node and every other node.
  ◦ For every edge in the graph:
    ▪ Calculate the attractive force.
  ◦ For every node in the graph:
    ▪ Limit the displacement to the temperature and prevent movements outside the frame.
  ◦ Reduce the temperature.

The cooling function has a big effect on how long the algorithm takes to finish, so it should be chosen carefully. The authors conducted an experiment where they found out that it is better and faster to choose a low initial temperature than to choose a high temperature that decreases rapidly. The algorithm does not have an explicit termination condition like many other algorithms which means that after the algorithm has finished, the result could be “half-baked” and not fully satisfactory. The authors rejected planarity testing and planarization even though they are efficient, because they wanted to have a simple and coherent implementation and not a “bag-of-tricks”. Each iteration of the algorithm takes $O(|V|^2 + |E|)$, where $V$ is the number of nodes and $E$ is the number of edges. The attractive forces takes $O(|E|)$ to finish and the repulsive forces takes $O(|V|^3)$ to finish.

2.1.2.3 GEM (Frick / Ludwig / Mehldau)

GEM (short for graph embedder) is an algorithm that was introduced by Frick et al. [10] in 1994. The major design goal was to make it fast to compute even for medium-sized graphs, their definition of fast is under 2 seconds. The algorithm is a refined and randomized spring embedder that introduces several new heuristics including a global temperature, local temperatures for every node, the attraction of nodes towards their barycenter and the detection of rotations and oscillations. The barycenter is the point in the graph where all the nodes balance each other out, also known as the center of mass. The pseudo-code for the algorithm could look like this:

• For every node in the graph:
  ◦ Initialize the temperature, impulse and skew-gauge for the current node.
  ◦ Place the current node in a random position.
• While the global temperature is greater than the desired temperature and the maximum number of rounds have not been reached:
  ◦ Shuffle the list with all the nodes.
  ◦ For every node in the list:
    ▪ Calculate the impulse of the current node.
    ▪ Update the position and the temperature of the current node.

The algorithm consists of two stages, an initialization stage and an iteration stage. At the initialization stage, the initial impulse of each node is set to zero to indicate that they are stationary at the start. The initial positions of the nodes do not have a great effect on the final result, and therefore a random placement is sufficient. At the iteration stage, the nodes to be updated are chosen randomly. The iterations are grouped into rounds and at the beginning of each round a random permutation is chosen. This is because GEM is found to converge faster when it is using a random selection mechanism rather than a deterministic schedule.
The distance a node can travel at a given time is limited by its local temperature. The local temperature increases if the algorithm determines that the node is far from its goal and decreases if the node is close to its goal. The global temperature is the average of all the local temperatures in the graph. The lower it is, the more stable the graph is. The GEM-algorithm also memorizes the last movement for each node, this is called the impulse. If the current impulse for a node is roughly the same as the last one, GEM interprets this as a move in the right direction and raises the local temperature to accelerate its next move.

Every node has a direction skew-gauge that determines how likely it is that the node is oscillating or being part of a rotation. An oscillation is when a node moves back and forth without stabilizing. If the last and current impulses of a node are in opposite directions, there is an oscillation. GEM solves this by lowering the temperature of that node to make the movement smaller, at some point the oscillation will stop and the node will be in its optimal position. A rotation is when nodes continuously move around in a circle. It could happen for example if the final layout has been found but the temperature is still too high and therefore the nodes move around too much and cause a rotation. The solution is to lower the temperature to make the movement of each node smaller.

By experimenting, the authors found a heuristic that says that approximately $|V|$ rounds are needed for a good result. Since a round contains $|V|$ iterations and an iteration considers $|V|$ nodes, the estimated time complexity is $O(|V|^3)$.

### 2.1.2.4 DH (Davidson / Harel)

In 1996 Davidson et al. [9] introduced an algorithm that uses simulated annealing to produce visually pleasing graphs, it will be referred to as “DH” in this report. The authors themselves describe the algorithm as flexible, since it uses relative weights for the criteria that can be changed. It produces good results for graphs of modest size, competitive with those produced by spring embedder-algorithms. Their definition of a modest-sized graph is a graph that contains around 30 nodes.

Several entities must be defined when using this method. A configuration, which is an assignment of positions for every node, must be specified. This can be randomized in the initialization. Then, a generation rule for new configurations based on the neighborhood must be specified. A neighboring configuration is similar to the current configuration but with a slight change. We need a control parameter, or a temperature, where the initial value and the rules for how and when it changes are specified. The last thing we need is a target function to be minimized and a termination condition. The termination condition is often based on the target function and/or the control parameter. A simple summary of the algorithm could look like this:

- Choose an initial configuration and an initial temperature.
- Repeat the following (a fixed number of times):
  - Choose a new configuration from the neighborhood of the selected configuration.
  - Calculate the value of the cost function for both of the configurations.
  - If the new configuration has the better value OR if a randomized variable allows it:
    - Replace the selected configuration with the new one.
- Decrease the temperature.
- If the termination condition is not satisfied:
  - Go back to step 2.
Great care should be taken when defining the cost function, since it is the heaviest computational task of this algorithm. It sums over five criteria: the node distribution, the edge length, the edge crossings, the node-edge distances and the borderlines. To clarify, the borderline criterion is to make sure that the nodes do not spread out too much or get too close to the borderlines. The time complexity of the algorithm is at worst $O(|V|^2 \cdot |E|)$.

2.1.3 Analysis of Graph Visualization Algorithms

2.1.3.1 Related work

Some comparisons of these algorithms have already been made. Brandenburg et al. [8] experimented with five graph drawing algorithms for general undirected graphs. Their conclusion was that there is no universal winner, and for every algorithm they tested they found examples of where it could produce pleasant drawings. Worth noting is that their implementation of KK was modified from the original. In the original it was hard to predict the number of iterations but in this implementation the algorithm uses $10 \cdot n$ iterations. This has lead to a “significant speed-up”. The biggest graphs that they used to test the algorithms had roughly 160 nodes and edges. Since some of the graphs that will be tested in this report are going to be a lot bigger than that, these results may not be applicable in this case.

GEM and KK were reported to be the most competitive when it comes to speed while DH, on the other hand, was the most time-consuming algorithm. Davidson et al. [9] also mentions in their paper that the time performance in their implementation is “quite poor”. A graph with 37 nodes and 68 edges took two minutes to complete, but it is worth remembering that the paper was published in 1996 and the same graph would most likely be completed a lot faster on a modern computer.

FR is fast on small graphs but slow on graphs with over 60 nodes. DH is the most flexible, but it is difficult to find appropriate weights for the criteria. For example, if the penalties for edge crossings are high then the running time increases a lot and the uniformity of the edge lengths decreases. KK, FR, GEM and DH without crossing optimization often produced similar-looking graphs [8].

According to Frick et al. [10], GEM is consistently faster than both FR and KK. FR is about four times slower than GEM, and KK quickly deteriorates when the graph size increases. Davidson et al. [9] says that the quality of their graphs are competitive in comparison with other algorithms. The running time is a problem however, and the method “does not perform well” on graphs with over 60 nodes. They are confident that it could be improved with a more serious attempt at optimization.

2.1.3.2 Review

FR, GEM and DH use multiple parameters and that makes it difficult to compare their performances. For example, an algorithm might be really fast with a certain set of parameter values but the resulting graph might not be visually pleasing at all. When looking at the time complexities, it seems that KK and FR are the most competitive algorithms when it comes to speed. This is however a bit misleading. In the case of KK, the runtime depends heavily on what value $T$ has, which they did not specify in their report. In the case of FR, the authors claim that they can offer little justification for the number of iterations needed. They even call the algorithms termination
“guesswork”, since it is so difficult to say how many iterations are needed for a good result. Even the authors of GEM do not specify how many rounds are needed, although they suggested that the number of rounds can be the same as the number of nodes in the graph. The time complexities do not tell the whole story, they are however important to consider.

The run time depends on the implementation and the hardware used, and since most computers today have CPUs with multiple cores, some parts of the algorithms could be sped up using parallel programming. In KK, finding the shortest distance between every pair of nodes and solving the linear equations could be parallelized. In FR, the calculations of the repulsive and attractive forces could be sped up by parallelization since the nodes are stationary during that part of the algorithm. The iterative stage of GEM does not allow much parallelization however, and neither does DH.

All four algorithms seem to produce somewhat similar looking graphs. DH is the only algorithm here that uses simulated annealing, but it evidently works well in this context. But since there is a large amount of varieties of graphs, it is difficult to do an extensive evaluation. The creators of the algorithms did not include any benchmarks or examples on huge graphs, but according to the creators of GEM, their algorithm should scale better than the other algorithms.

Whether it is a good or a bad thing that an algorithm is very customizable is hard to say. It is a good thing because it allows the user to have more input on how the algorithm should work, however, the process of finding good values could be very time-consuming. For example, the values that work well for a graph with 500 nodes might not be as good for a graph with 7,000 nodes. A solution would be to try to find a set of default-values that work reasonably well independently of how big the graph is.

The differential equations in KK could prove to be difficult to solve, likewise, the features in GEM are relatively complicated. DH also seems to be complicated to both implement and use because of all the parameters it uses. FR seems to be the easiest algorithm to implement since it is built to be simple and coherent, but that is not necessarily a good thing from a performance standpoint. The authors rejected some functionalities to make the algorithm simple, even though they were said to be efficient.

Since the creators of DH said themselves that the method does not perform well on large graphs, it can be ruled out. It could still run reasonably fast using optimization and newer hardware, but the same could be said for every other algorithm. Since speed is an important criterion, this is a good enough reason to rule out this algorithm.

To sum it up:

- KK is said to be fast on medium-sized graphs. However, it does not scale well, it is not very customizable, the speed is very dependent on the number of iterations which means that large graphs will take a lot of time, and implementing the function for solving differential equations could prove to be difficult.

- FR should be the easiest algorithm to implement and it can be parallelized to some extent, but it is slow on big graphs. The resulting graph could also be bad since there is no explicit termination condition.

- GEM is fast, it is said to scale well and it is also customizable. It has built-in detection for rotations and oscillations, however, implementing this could potentially be difficult. It could
also be time-consuming to find good values for the parameters.

- DH is very customizable, but it also seems difficult and time-consuming to find good values for all the parameters. The algorithm is also slow, even according to the creators.

GEM is faster than KK and FR, it scales better than the other algorithms and it is customizable because of all of the parameters used and the detection for rotations and oscillations. For these reasons, it clearly seems like GEM is the algorithm that is best suited for visualizing big graphs.

2.2 Software Platforms

2.2.1 JavaFX, Touches and Gestures

JavaFX is a set of graphics and media packages that can be used for designing, creating, testing, debuggin and deploying rich internet applications. It is a library written as a Java API and it is intended to replace Swing as the standard GUI library for the standard edition of the Java platform. JavaFX is currently fully integrated within the latest Java SE Runtime Environment (JRE) and the Java Development Kit (JDK) and is available on Windows, Mac OS X and Linux. Starting with JavaFX 2.2, it is possible to create applications that users can interact with using touches and gestures on touch-enabled devices [18]. As of now, the latest version of JavaFX is version 8.

In JavaFX, an event is a notification that something has happened. Every touch and gesture in the application is registered as an event, and these events need to be processed correctly in order for the touch-functionality to work properly. There are a number of different events that can occur based on which gesture is used. For example, when a key on the keyboard is pressed, a KeyEvent is generated. To implement the functionality that is needed for the final application we will be using TouchEvent to be able to move the graph, ZoomEvent to zoom in and out and RotateEvent to rotate the graph.

A touch event is triggered when a user touches the touch screen with one or more fingers. There are four different types of touch events that can occur: when a touch point is pressed for the first time, when it is pressed and stationary, when it moves and when it is released. When a touch action has a single point of contact, a single touch event is created. When a touch action has multiple points of contact, a set of touch events are created where each event represents one touch point. Touch events can only be generated from touches on a touch screen, not from touchpads.

A touch point is an instance of the class “TouchPoint”, and it contains information about the location, the state and the target of the point of contact. A touch event is an instance of the class “TouchEvent”, and it contains a list of every touch point associated with the event that is currently active and information about them. Every touch point and touch event have a unique ID that is assigned when they are created, this makes it easier to handle multi-touch actions from one place while filtering each touch point separately.

A zoom event is triggered by a two-finger pinching motion where the fingers are moved away from each other to zoom in and brought together to zoom out. Three different events are generated: when the zooming gesture starts, when a user is zooming and when the zooming gesture ends. A rotate event is created by a two two-finger turning movement where one of the fingers moves clockwise or counterclockwise around the other finger. Similar to a zoom event it has three different events that
are generated: when the rotation gesture starts, when a user is rotating and when the rotation gesture ends. Both zoom events and rotate events can be generated from both touch screens and touchpads.

A small test-application was created in order to test these gestures on the multi-touch screen, a link to the Git repository can be found in Appendix A. It creates a simple graph containing three nodes that are connected to each other, and the user can move the whole graph, zoom in and out and rotate the whole graph. It is not possible to move the individual nodes using gestures in this small application, there is however a button that moves a random node to a random location inside the window.

2.2.2 Graph Visualization Libraries

There exist a number of different platforms that can be used to visualize graphs. Two of them, Gephi and Prefux, were chosen for this report. The chosen algorithm will be implemented in one of them, together with the necessary touch-functionality.

2.2.2.1 Gephi

Gephi is an open-source software for visualizing and understanding graphs. It helps users to reveal patterns and trends and highlight outliers within networks. It uses an ad hoc OpenGL 3D render engine to display graphs in real-time and the user can manipulate the structures, shapes and colors to reveal different properties. The community has said that the tool is “like Photoshop but for graphs”. Gephi already provides a couple of layout algorithms to help visualize networks [19]. Gephi is developed using NetBeans, which is a software development platform written in Java, and it follows a loosely coupled, modular architecture philosophy. This means that the application is split up into modules which depend on each other through APIs, and plugins are allowed to reuse those APIs to create new services or even replace the default ones. As of now, the latest version of Gephi is 0.8.2beta and it was released at the start of 2013.

2.2.2.2 Prefux

Prefux is a data visualization library that aims to migrate the Prefuse library to JavaFX [20]. Prefuse is a toolkit that uses the Java 2D graphics library together with Swing to visualize graphs that the user can interact with in different ways [21]. While there are a lot of similarities between the two libraries, they are not compatible with each other since Swing and JavaFX are different. The main concepts are similar which means that it should be possible to convert code from one library to the other without too much inconvenience. The main goal of Prefux is to implement the force based layout visualizations in JavaFX. It is a work in progress, which means that all of the functionality from Prefuse is not (yet) implemented. Prefux is developed using Eclipse, which is an integrated development environment mainly used for developing Java applications [22]. It is built using Gradle, which is an open source build automation system that is already included in Eclipse [23].

2.2.3 Analysis of Graph Visualization Libraries

Gephi and Prefux have many things in common, but the big difference is that Gephi is a more extensive platform with lots of functionality while Prefux is more of a bareboned library. Gephi has
a graphical interface where the user can for example change the color of the nodes. Prefux has no
graphical interface for doing that, the colors would have to be changed programmatically instead.
Both have APIs for handling graph objects and creating layout algorithms, therefore implementing
an algorithm should not be that difficult for either.

A lot of information on how to start developing plugins for Gephi can be found on their website,
together with a comprehensive Javadoc containing the documentation for all the APIs. In order to
help developers get started with developing plugins there is also a suite called “gephi-plugins-
bootcamp” available on github [24]. It is an out of the box plugin development suite and it contains
examples for a number of different plugins. In contrast, Prefux does not have any documentation
available. The documentation for Prefuse can be used to a large extent, but since they do not have
exactly the same methods, the developer needs to double check to make sure that the methods in the
documentation exist and are implemented in Prefux. There is a file called “JavaFxSample.java” in
the project that shows how to set up the application and run it, this is a good starting place for those
who want to start developing for Prefux. A few of the methods in Prefux are not implemented yet,
but everything that is essential for basic functionality is. Even if there is no fully accurate
documentation available, most of the methods in the code are commented.

Even though Gephi has documentation and Prefux does not, it can still be difficult to start
developing for Gephi since it is much more complex. In Gephi, the developer needs to integrate the
algorithm into the application so that a user can pick the algorithm and run it at any time. In Prefux,
the user would have to modify the code to choose which algorithm to run since there is no graphical
interface for doing that at run-time. The graphical interface could be extended to allow picking an
algorithm at run-time, but that would of course require additional work.

Gephi and Prefux use different techniques for rendering the graphs. Gephi uses a combination of
both the Abstract Window Toolkit (AWT) and Swing. AWT is a platform-dependent graphics toolkit
and Swing is a platform-independent GUI toolkit. AWT is considered heavyweight since its
components are implemented by platform-specific code, therefore components might look different
on different operating systems. Swing is however lightweight and its components are entirely
written in Java. Prefux on the other hand is already using JavaFX exclusively. JavaFX is very
important for this thesis because it enables support for touch devices, and this makes Prefux’s
approach more preferable. Gephi does not work with the latest version of Java which, at the time
this was written, is version 8. In order to run Gephi, Java 7 has to be installed and then a
configuration-file called “gephi.conf” has to be changed in order for Gephi to find the path to it.
Plugins have to use the older version of Java and JavaFX as well.

It is possible to create plugins for Gephi that use JavaFX, but those plugins will not be able to
utilize Gephi's renderers. In order to implement the algorithm in Gephi using JavaFX, the
visualization-part would have to be made from scratch, either by creating a new JavaFX-renderer
specifically for Gephi or by creating a stand alone graph visualizer and embed it inside Gephi. It
would be very time-consuming to create a new JavaFX-renderer, and creating a stand alone graph
visualizer would pretty much defeat the purpose of using Gephi in the first place since it would be
disconnected from all the other functionalities.

To sum it up:

- Gephi is a more extensive platform with a lot of documentation, but it does not use JavaFX.
  Creating a JavaFX-renderer for Gephi would take a lot of time.
• Prefux is more simple than Gephi and it uses JavaFX. The documentation is poor and some of the methods are not implemented yet, but the basic functionality is there.

Since implementing a JavaFX-renderer for Gephi would take too much time, the choice for this thesis ended up being Prefux since it already uses JavaFX.

2.3 Ontology Engineering and Alignment

Ontology engineering deals with the process of creating and developing ontologies, together with the tools, methodologies and languages used [25]. There exist a number of different tools for handling ontologies where the user can (for example) edit, browse, import and export ontologies of different formats. One example of such a tool is Protégé, which is an open-source ontology editor which is being developed at Stanford [26].

There are many different classifications of the different types of ontologies, since the numerous authors of these classifications have had different viewpoints [27]. Ruiz et al. [27] has combined two different classifications into one where the different types have been sorted by a generality level. The most general types are called domain independent, as opposed to domain dependent. Domain independent ontologies describe general concepts without restrictions to specific domains, while domain dependent ontologies are modeled after a particular domain and can be more specific. They can also be classified into lightweight versus heavyweight ontologies, where a heavyweight ontology models a certain knowledge in a “deeper way” with more restrictions and constraints.

Ontologies are considered to be one of the pillars of the so called Semantic Web. The semantic web is an extension of the World Wide Web, its purpose is to make it easier to process the information from websites using automated tools [25]. The way the web currently works is that information is usually presented in a way that makes it easy for humans to understand, but hard for computers. Today’s markup languages are usually used to specify what the data is going to look like in the browsers, rather than what the data is about. The semantic web is the idea of having data defined and linked so that it can be used for other purposes than just displaying it in browsers.

Ontologies can be broken and have defects, such as inconsistencies and unsatisfiable concepts (or classes), which obviously makes it difficult for semantically enabled applications to process them. Several approaches to debugging ontologies have been introduced, including common reasoning techniques, machine learning and statistical methods [3,25].

Ontology alignment is an important subfield of ontology engineering. It is the process of finding concepts and relations in different ontologies and matching them together. An alignment is represented by one or more mappings between concepts from different ontologies, these concepts can then be referred to as mapped concepts [3]. Ontology alignment can be done either manually by the user, or automatically using different algorithms and applications. It is not an easy task, and it is difficult to ensure that everything is correct as the ontologies grow bigger [3]. The work in this thesis is related to the work on the SAMBO ontology alignment system developed at Linköping University [5,28].
3 Design and Implementation

This chapter presents the design of the application together with details about the implementation of the algorithm and all the necessary functionality.

3.1 Design

As already mentioned, Prefux is a data visualization library and it is relatively simple to use and modify. The project already contains a file called “JavaFxSample.java”, this is the main application file that initializes the application. This is the most important part of Prefux for our purposes, and a good starting place for adding new functionality. A simple overview can be seen in the figure below:

![Diagram](Image)

Figure 4: A (very) basic overview of how Prefux works.

The algorithm, the controller and the renderer are chosen in the main application file. These parts are loosely coupled and they do not communicate directly with each other. The purpose of the algorithm is to change the layout of the graph by moving the nodes, the controller can add functionality for handling user input, for example touches on a screen, and the renderer makes sure that the items on the screen are rendered correctly. This is a short summary of what happens when JavaFxSample is executed:

- A window is created for the visualization.
- An XML-file is read and a graph is created from its content.
- A set of renderers are selected for the visualization.
- An algorithm is selected.
- An “FxDisplay” is added to the window.
- A controller is added to the display to enable the user to drag the nodes.
- The algorithm executes.

The visualization is located in the FxDisplay, which in turn is placed in the application window. Other things are done as well, such as adding a color palette to help the user distinguish between different types of nodes, among other things. Multiple renderers can be used at the same time, which will be needed to display both the nodes and the labels. However, when a combination of renderers are used, the methods for changing their looks do no longer work. This can be circumvented by modifying the JavaFX-object directly. A simple explanation of the process
• For every node in the graph:
  ◦ Cast the node to a Group-object to gain access to its children.
  ◦ For every child:
    ▪ If the child is a node:
      • Change the node’s appearance.
    ▪ Else, if the child is a label:
      • Change the label’s appearance.

A Group is a type of node that has a list that can contain other nodes, this will be used when multiple renderers are combined into one. The process of expanding and collapsing nodes will be very similar to this, but the nodes and the edges will instead be set to invisible.

All the variables and constants that the algorithm will need will be placed within the algorithm file, there is no need for any other part of the application to be able to access them. Since there will be two different ways to measure distances, Euclidean and Manhattan, the algorithm must only use one of them at a time. The default renderer that exist in Prefux does not show the relationship between connected nodes. A new renderer will have to be implemented that renders an arrowhead at the target-node with the correct rotation.

3.2 Implementation

3.2.1 Setup

Building Prefux using Gradle is done by first opening Eclipse, choosing File and then Import. Then select Gradle Project, set the Project root directory to the folder where Prefux is located and click on Finish. The included JavaFxSample sets up an application that displays a graph using a layout called “ForceDirectedLayout”, which is an interactive spring embedder where a user can move the nodes using a mouse. A copy of this file was made and named “GemMain”. The GEM-algorithm was implemented by creating a new layout class called “GraphEmbedderLayout”, and then “GemMain” was modified so that it used the new layout-algorithm instead, among other things. Two additional files were also created: “GemControl” handles the touch-functionality for expanding, collapsing, selecting and moving nodes. “ArrowRenderer” is used to render the edges as arrows instead of just simple lines, this is to help the user understand the relation between the connected nodes.
The ontologies that have been tested are written in the Web Ontology Language (OWL), which is a semantic web language designed for use by applications that need to process information rather than just presenting it [29]. The file JavaFxSample already uses a method that generates a graph from an XML-file, but this method cannot read OWL-files. Instead, a framework called Apache Jena is used, it provides an API to help extract data from ontology files. Jena 3.1.0 is used in this project, it is currently the latest version. It is available for download from Apache's website, a link can be found in Appendix A.

The libraries needed are contained in jar-files and they are located in the folder “apache-jena-3.1.0/lib”. Adding them to the Prefux project is done by first right clicking on Prefux-master in the Package Explorer in Eclipse and then clicking on Properties. The libraries are listed under Java Build Path and then Libraries. They are added by clicking on Add External JARs, and then selecting the jar-files from where they are located.

3.2.2 GemMain

This is the file that contains the main application. Its purpose is to set up the application before starting the algorithm. In short, it displays the application window, creates a new graph based on the content of a specified ontology file and then starts the algorithm. It also contains the touch-functionalities for moving, zooming and rotating the graph.

A javafx.scene.layout.Pane is used as the application window. A javafx.scene.layout.BorderPane is used in JavaFxSample and it uses a slider for zooming in and out, but this is not a requirement for an application with the ability to zoom using touch gestures. Using a Pane also allows for future modification to some extent since it is the base class for some other panes.

The reading of the ontology-file is done via code that was sent to me from my mentor. An OntModel, which is an interface of a Jena model that contains ontology data, is created and it is used to read the ontology-file by calling a read-method. The location of the file is passed as a string to the method. An iterator is created that iterates over all the ontology-classes that are not anonymous. An anonymous resource would in a graph be represented as a blank node, which is a node with no data, and there is no point in including these in the graph. All the relevant ontology-classes are then added to an ArrayList called “ontList”. Two tables are created, one for the nodes and one for the edges. A loop iterates through ontList and adds the nodes and edges to these tables. A graph-object is created using the two tables, and this graph-object is then added to the visualization.
A set of renderers are needed in order to draw the shapes of the nodes and the edges on the screen, and it is possible to use multiple renderers at the same time by combining them using the class CombinedRenderer. Two separate renderers should be used for the nodes and the edges, by default a ShapeRenderer is used for the nodes and an EdgeRenderer is used for the edges. In this application, a combination of a ShapeRenderer and a LabelRenderer are used to render the nodes. The ShapeRenderer draws a circle and the LabelRenderer adds a label on top of the circle with the name of the node. A custom class called ArrowRenderer was created specifically for this application to render the edges. The EdgeRenderer only draws a simple line between nodes, but the ArrowRenderer draws an arrow instead. This is preferable because then the user can easily see the relation between the connected nodes. The implementation will be explained in more detail later on.

The algorithm is added to an action list, which is a list containing actions that process VisualItems, and this list is then added to the visualization. To run the algorithm, the action list needs to be scheduled to run by the visualization. It only needs to run once in this implementation, there is no need to run it again after it has finished.

Changing the looks of the nodes is bit of a hassle in Prefux, since all the functionalities from Prefuse have not been implemented. In order to change what a node looks like, the JavaFX-object needs to be accessed. This is done by calling the method “getNode” on a VisualItem which returns a javafx.scene.Node, this is then casted to a javafx.scene.Group. When a CombinedRenderer is used, the groups list of children needs to be accessed to get to the underlying objects such as the circle and the label. A method called “initializeNodes” was created to take care of this before the algorithm starts executing. It changes the color of the nodes, increases the stroke width, sets the font to Verdana and also adjusts the size of the label based on the zoom level.

The touch-functionality for moving, rotating and zooming in and out on the graph is implemented in this file, and the rest is implemented in GemControl. The reason why these functionalities are separated is because they have different responsibilities. The touch-functionality in GemMain does not modify the graph-object, it only changes the scale, zoom-factor and rotation for the Pane in which the graph is contained. Meanwhile, the touch-functionality contained in GemControl is able to modify the graph-object. It does not need access to the Pane. Implementing the touch-functionality in GemMain is done by implementing the event handlers for the gestures on the Pane. These event handlers are independent of the algorithm, which means that they could be used together with any of the other algorithms that already exist in Prefux. There are a total of nine event handlers in this file, three for each gesture:

- setOnTouchPressed
- setOnTouchMoved
- setOnTouchReleased
- setOnZoomStarted
- setOnZoom
- setOnZoomFinished
- setOnRotationStarted
- setOnRotate
- setOnRotationFinished

Moving the graph is done by holding a finger on the screen and then dragging it, the display that contains the graph will then follow the touch point to its new location. Only one touch point is allowed at a time, a new touch point can only be created after the active touch point has been
The zooming is not affected by the one touch point limit. Zooming is done by changing the scale of the Pane based on the scale before the zoom event started. When the zoom event finishes, the degree of visibility is changed based on the new scale. This degree of visibility decides how many children a node needs to have in order for its label to be shown. The more zoomed in it is, the less children the nodes need to have, and all the labels will be shown when the graph is zoomed in far enough. If all the labels would be visible all the time the screen would be very cluttered, especially when fully zoomed out on a huge graph.

Rotating the graph is done by simply rotating the Pane. When the rotation event is finished, all the labels need to be rotated separately to stay leveled. The rotation functionality is commented out because it turned out to be computationally heavy when the graph contains thousands of nodes.

### 3.2.3 GraphEmbedderLayout

The whole algorithm is contained in this file. It has access to all the nodes in the graph and is able to utilize all the methods for modifying and fetching data from them. In this algorithm, a node is represented as a custom class called Vertex. The reason is that every node needs to have some additional properties in order for the algorithm to work, and this is an easy and straightforward way to define and access them.

```java
private class Vertex {
    private final VisualItem item;
    private double[] impulse = new double[2];
    private double temp = 1024;
    private double skew = 0;
    private double[] coordinates = new double[2];
    private List<Vertex> neighbors = new ArrayList<>();
    ...
}
```

A VisualItem is an interface for representing a visual item. A VisualItem is in this case either a node or an edge. There are several methods for manipulating this visual representation, in this file we will mostly focus on just changing its position. When the coordinates for a VisualItem change, the item will move to its new location immediately. The variable “Impulse” is a vector that specifies the last direction and distance that a node moved towards. The reason why we want to save this is to check whether the next impulse triggers a rotation or an oscillation. The variable “temp” is the node’s local temperature, it indicates how far the node will move and it will decrease as the node moves toward its optimal position. The variable “skew” is used to adjust the local temperature in the case of a rotation. The higher it is, the lower the temperature will be after the scaling has been completed. The last two variables, “coordinates” and “neighbors”, will help speed up some calculations as we can see later on.

A couple of additional variables are also needed for the algorithm to work. The creators of GEM did not specify exactly which values to use for all the variables, so some of the values below are just estimations. More optimal values will be explored and discussed later.
Some type of list should be used in order to have fast access to all the nodes, an ArrayList is sufficient in this implementation. The boolean value of “initialized” is used to indicate whether the algorithm has been initialized or not. The number of rounds that have been completed is stored in “nrRounds.” Meanwhile, “maxRounds” is the maximum number of rounds allowed before the algorithm terminates, it will be set to nodeList.size() * 4 during the initialization. The integer “updateFrequency” determines how often the visualization is updated, this will be explained later.

The boolean “euclideanDistance” determines whether the distances between nodes are calculated using the Euclidean distance or the Manhattan distance. The global temperature is stored in “globalTemp” and a new value will be assigned after every round. The sum of all the coordinates is stored in “sumPos”, this is used later to quickly calculate the coordinates for the barycenter, or the center of mass.

The maximal temperature that a node can have is stored in “maxTemp”, and “desiredTemp” is the desired global temperature. The desired edge length is the preferred distance between two nodes and the gravitational constant determines how strongly a node is driven towards the barycenter. The oscillation- and rotation opening angles are used to see if a new impulse will trigger either an oscillation or a rotation, and the respective sensitivities determines how much the temperature is changed if they occur.

The initialization-stage is the first part of the algorithm. The following things happen in the initialization-stage:

- The nodes are placed in random coordinates inside a square, the square's dimension is 2048 by 2048 units.
- The visible nodes are added to “nodeList” to allow fast access and the additional properties that are needed for the algorithm.
- The neighbor-lists for all the nodes are initialized to allow faster access to neighboring nodes. With this solution, we do not have to loop through all the nodes and check whether they are neighbors every time we need access to a node’s neighbors.
- The variables “maxRounds” and “rotationSensitivity” are set based on the number of nodes in the graph.
After this has been completed, the main loop will start executing. It will loop until the desired temperature has been reached, or until the number of rounds completed have reached the maximum number of rounds allowed. When the algorithm finishes, it sets a boolean property called “fixed” to true on every VisualItem in the graph, this is to indicate that all the nodes are in position and the algorithm will not move them around anymore.

Changing a VisualItem's coordinates is a rather time-consuming operation, so in order to speed up the algorithm we want to minimize the number of times that this is done. One could argue that it is not necessary for the user to see every move that is made, the user is mostly interested in the final result. It is not necessary to move a node to its new location immediately after its impulse has been calculated. Therefore, we store every node's current location in a local array called “coordinates”. We can then decide how often the visualization should be updated to the most recent state. The code below is executed at the end of every round:

```java
if(nrRounds % updateFrequency == 0 || globalTemp <= desiredTemp) {
    System.out.println("Updating visualization...");
    for(Vertex v : nodeList) {
        v.item.setX(v.coordinates[0]);
        v.item.setY(v.coordinates[1]);
    }
}
```

All the movements will be grouped together and executed together, and the number of times it will happen depends on the variable “updateFrequency”. For example, if “updateFrequency” has the value 5, the visualization will be updated every five rounds. The higher it is, the faster the algorithm will be overall. Worth noting is that it can not be set to 0, and setting the variable to 1 is still faster than moving the nodes continually after every impulse-calculation.

The most time-critical part of the algorithm is where the repulsive forces are calculated. Every time a node's repulsive forces are calculated, the algorithm will iterate through every other node in the graph. The distance from the current node to every other node is calculated and then scaled before it is added to the impulse vector. Every round this part will be executed n^2 – n times, where n is the number of nodes in the graph. For example, if we have a graph with 3,000 nodes, this part will execute 3,000 * 2,999 times, or 8,997,000 times.

The original paper doesn't mention which method should be used to calculate distances [10]. In this implementation, the distance between two nodes can be calculated using either the Euclidean distance or the Manhattan distance. Which one of these methods are used is decided by the value of the variable “euclideanDistance”. The Euclidean distance is calculated using the two nodes coordinates and the Pythagorean theorem:

```java
distance = Math.sqrt(delta[0] * delta[0] + delta[1] * delta[1]);
```

Here, “delta[]” is an array that contains the differences in x- and y-coordinates between the two nodes. There is already a method in the Math-library called “Math.hypot” that can take of this calculation, but it is very slow and using it increases the time it takes to finish the algorithm by almost tenfold [30]. The Manhattan distance is however even more simple to calculate, and it is faster. The method called “Math.abs” needs to be called twice to get the absolute values of the two values in “delta[]”, and then the sum of them is assigned to “distance”:
distance = Math.abs(delta[0]) + Math.abs(delta[1]);

The method for updating a node's temperature is slightly different from the method used by the authors of the original paper [10]. Looking at the pseudo-code from the original paper, there is no way that a node's last impulse will ever be updated. There, the last impulse will only be set if the last impulse is not equal to zero, but this will never happen since it is set to zero in the initialization. In this implementation, the last impulse will always be set to the new impulse at the end of this method, no matter what. This only changes the way the algorithm works in the first round, after the first round the last impulse will always be updated anyway since it will never be equal to zero again. Another thing that differs from the pseudo-code is the temperature adjustment when an oscillation is detected. The cosine of the angle is used in the original paper, but that can produce negative values and it does not make any sense to give the temperature a negative value. Instead, the temperature is multiplied with the sensitivity-constant when a move in the right direction is detected, and divided with the constant when an oscillation is detected.

3.2.4 GemControl

This file contains the touch-functionality for selecting nodes, moving them and expanding and collapsing subtrees. The class GemControl is an extension of ControlAdapter, which is a class for processing interface events. In order to include GemControl in the application, it needs to be added as a control listener to the display in GemMain.

The method “itemEvent” is implemented to take care of the different functionalities, it triggers when the user touches an item in the graph. There are two recursive methods implemented that hide all the children of the node that is passed to these methods, they are used when the user collapses or expands nodes. There are also three additional methods that return the circle-object, the label-object and a list with all the outgoing lines from the node that is passed to these methods. These are used to get access to the underlying JavaFX-objects and to be able to show and hide the objects from the user when a subtree is collapsed or expanded. The last three methods are also used in GemMain when the nodes are initialized.

When the user touches an item in the graph there is a check to make sure that the item that was touched is fixed, which it will be only if the algorithm has finished running. It is therefore not possible to interact with the graph while the algorithm is still running. Then there is a check to see what kind of state the active touch point has, and the appropriate action will execute based on that.

When the user first touches a VisualItem, the current value of the system timer in nanoseconds is fetched and stored in the variable “startTime”. If the user then holds the finger on the item without moving it too much, the time elapsed will be calculated by fetching the current value of the system timer again and subtracting “startTime”. This is then converted to milliseconds by dividing the result with 1,000,000. This is done over and over again until either the number of milliseconds exceed 500, or the user releases the item. If the item is held for over 500 milliseconds, it is selected. If the item already was selected, it is deselected. If the user releases the item before 500 milliseconds have passed, the item is expanded or collapsed based on its previous state.

It is possible to select multiple items. Selecting an item is done by first adding the item to a list containing all the selected items. The item has a boolean property called “highlighted” that is set to true, and finally the circle-object is fetched and the stroke width is changed so that the user can see
that the item has been selected. Deselecting an item does the opposite; the item is removed from the list, “highlighted” is set to false and the stroke width of the circle is reset to the default value.

Collapsing a node is done by first calling the recursive method “hideChildren” with the current node as its argument. The method loops through every child of the current node and makes them invisible. After this has been done, all the outgoing edges from the current node are made invisible. The color of the current node is changed to indicate that it has been collapsed, and a boolean property called “expanded” is set to false. Expanding a node then does the exact opposite. Another solution would be to remove the collapsed nodes from the visualization instead of just hiding them, but that would be more complicated since we would have to keep track of which nodes were removed and store them somewhere temporarily. Expanding would be done by adding the correct nodes to the visualization again.

The user can also move one or multiple items, only items that have been selected and are expanded can be moved to a new location. In order to move an item, the user must first touch and hold the item until it is selected, then the item can be moved by simply dragging the finger on the screen. Every item that is selected will be moved along with the current item, as long as they are expanded.

### 3.2.5 ArrowRenderer

The purpose of the ArrowRenderer is to render the lines between the nodes as arrows. The renderer extends the class AbstractShapeRenderer and implements the method “getRawShape”, this method returns a javafx.scene.Node and it is called when the nodes are rendered on the screen. In this method, the default EdgeRenderer creates a javafx.scene.shape.Line, binds its coordinates to the source node and the target node and then returns the line-object. This way, whenever a node's coordinates change, all the connected lines will move as well.

The method “getRawShape” in ArrowRenderer returns a Group that consists of a Line and a javafx.scene.shape.Polygon. A Polygon is a closed shape that is defined by an array of x- and y-coordinates. In this case it is the arrowhead that is supposed to point towards the target node. Every time a node moves to a new location, the arrowhead needs to be rotated so that it does not point in the wrong direction. A method called “adjustArrowRotation” was implemented to take care of this, it is called every time a node moves to a new location and it adjusts all the arrows that are connected to this node. The correct angle is calculated by using the method “Math.atan2” which converts the rectangular coordinates to polar coordinates, the new angle is then applied to the polygon. Worth mentioning is that the y-axis is inverted in Prefux which means that when a node's y-coordinate increases, the node moves downwards instead of upwards.

### 3.3 GEM-2

Since the algorithm is estimated to run in cubic time (O(n³)), the time it takes to finish the algorithm on a graph with twice as many nodes will increase by a factor of 8. To address this, the algorithm can be modified to only expand a specified number of levels at first. The idea is to introduce a variable which sets the number of levels that will be shown at first. After the algorithm has finished, it is possible to expand additional nodes and this will trigger the algorithm to run again, but only on the nodes that are being expanded. The nodes that have already been placed by the algorithm are fixed and cannot be moved again, unless they are moved using touch-gestures. Three new files were created for this new version of the algorithm: “GemMain2”, “GraphEmbedderLayout2” and
“GemControl2”. They are mostly similar to the original implementation, but a few changes were made.

An integer called “level” was introduced in GemMain2 to specify how many levels that are to be shown initially. To accomplish this, the root-nodes need to be found first. This is done by iterating over all the nodes and calling the method “getInDegree” that returns the number of parents the node have. If the returned value is zero, the node is a root-node and it is stored in an ArrayList. A recursive method called “hideNodes” is called on every root-node, this method hides the children that are a specific number of levels down. This method is somewhat similar to the method “hideChildren” from “GemControl”, the biggest difference is that it takes a second argument which is the current depth. This depth is increased by one every time there is a recursive method call and as long as it is more than the value of “level”, the current node will be hidden.

In GraphEmbedderLayout2, only the nodes that are visible and not fixed will be moved using this implementation. Since the original implementation of the algorithm is only meant to run once, the initialization-stage needs to be modified in this version. Like the original implementation, the init-method will only be called the first time the algorithm runs. All the nodes in the graph are added to “nodeList” instead of just the ones that are visible, and “maxRounds” and “rotationSensitivity” are set in the run-method instead since they will be different depending on how many nodes are being expanded.

A new ArrayList called “nodeListVisible” was introduced to keep track of all the nodes that are visible. When a new group of nodes is expanded, the positions of the nodes that have already been used before and are fixed in place needs to be accessed when calculating the repulsive forces. If not, the new nodes will not be repelled by the nodes that are already fixed in place and that would lead to a lot of overlapping nodes and edges. Every time the algorithm starts, the nodes from “nodeList” that are visible are added to “nodeListVisible”. If every visible node is already frozen, the algorithm will terminate because there is no need to move those nodes again. After that, the variables “maxRounds” and “rotationSensitivity” are set based on the number of visible nodes, and “sumPos” is assigned the sum of the coordinates from the nodes that are visible or fixed instead of just all the nodes. The main loop iterates over “nodeListVisible” and the nodes that are fixed will be skipped. The calculation of the repulsive forces iterates over the visible nodes only, and the attractive forces iterates over the current node’s neighbors that are visible. It does not make sense to fetch and use the coordinates from nodes that are neither visible nor fixed here, they are irrelevant.

The only difference in GemControl2 is that the algorithm is started every time a user expands a node. This is done by fetching the ActionList that contains the algorithm from the visualization and schedule it to run again.
4 Evaluation

This chapter presents the steps taken to finding a good setup for the algorithm. It also describes the testing environment and some details about the ontologies tested.

4.1 Setup

First of all, it is difficult and time-consuming to do an extensive evaluation for a graph visualization algorithm. GEM has many variables, it uses a lot of randomization and there are many ontologies who all have different sizes and properties. The two most important aspects for the evaluation are to look at how long it takes for the algorithm to finish, and what the resulting graph looks like. A number of different setups will be tested to see which one has the best balance of making the graphs look good and not taking too long. They will be tested several times so that the average and the standard deviation can be calculated.

The constants that have the biggest impact on what the graph looks like are the desired temperature, the desired edge length and the gravitational constant. A good temperature threshold is needed, the problem is that it depends on the size of the graph. If it is too low, the algorithm will continue running even if no major changes are happening to the graph. If it is too high, the algorithm will stop before the whole graph is stable. The desired edge length should be quite high when the graph is large to separate the nodes that are not related to each other. If it is too high though, the nodes will be too far away from each other. When the gravitational constant is increased, the nodes do not spread out as much from the barycenter and the global temperature decreases slightly faster. When the constant is decreased, the nodes spread out more and clusters of nodes are formed. If it is set to zero, any disconnected parts of the graph will move too far away. Generally, a low value is recommended when the graph is very large.

The constants that have the biggest impact on the execution time are the desired temperature, the initial temperature, the maximum temperature allowed, the opening angles for rotation- and oscillation detection, the sensitivities for rotations and oscillations, and the maximum number of rounds allowed. The initial temperature should be quite high, but if it is too high the algorithm will take longer to finish since the nodes will just “bounce around” aimlessly in the beginning. If it is too low, the temperature will increase the first couple of rounds as the nodes spread out. The maximum temperature allowed also plays an important role. If it is lower than the initial temperature, the nodes will have their initial temperatures in the first round, and after that they will be adjusted to the maximum temperature allowed. Since the initial placement of the nodes is randomized, this could result in nodes spreading out far away in opposite directions even if they are connected to each other. This could degrade the performance of the algorithm since they then have to move closer to each other again, sometimes over multiple rounds.

If the maximum temperature allowed is higher than the initial temperature, the global temperature will sometimes increase for a couple of rounds while the nodes are spreading out until it slows down and it starts decreasing. If the initial temperature and the maximum temperature allowed are the same, the global temperature should simply decrease in the beginning. It is difficult to speculate on which setup would be the best, so these five different setups will be tested:

**Setup 1.1:** \( \text{temp} = 1024 \quad \text{maxTemp} = 256 \)
Setup 1.2: \( \text{temp} = 1024 \) \( \text{maxTemp} = 512 \)
Setup 1.3: \( \text{temp} = 256 \) \( \text{maxTemp} = 256 \)
Setup 1.4: \( \text{temp} = 512 \) \( \text{maxTemp} = 512 \)
Setup 1.5: \( \text{temp} = 1024 \) \( \text{maxTemp} = 1024 \)

Since the desired edge length and the gravitational constant are closely related, they will be changed and tested together. The best values for the initial and maximum temperature above will be used for the following new setups:

Setup 2.1: \( \text{desiredEdgeLength} = 128 \) \( \text{gravitationalConstant} = 1 / 16 \)
Setup 2.2: \( \text{desiredEdgeLength} = 128 \) \( \text{gravitationalConstant} = 1 / 32 \)
Setup 2.3: \( \text{desiredEdgeLength} = 256 \) \( \text{gravitationalConstant} = 1 / 16 \)
Setup 2.4: \( \text{desiredEdgeLength} = 256 \) \( \text{gravitationalConstant} = 1 / 32 \)

If the angles for detecting rotations and oscillations are too big, rotations and oscillations will be detected all the time. If they are too small, they will rarely be detected at all. A figure from the original paper suggests that the angle for detecting rotations should be wider than the angle for detecting moves in the right direction and oscillations [10]. Three new setups will be tested with the best values from the two previous tests:

Setup 3.1: \( \text{oscillationOpeningAngle} = \pi / 2 \) \( \text{rotationOpeningAngle} = \pi \)
Setup 3.2: \( \text{oscillationOpeningAngle} = \pi / 4 \) \( \text{rotationOpeningAngle} = \pi / 2 \)
Setup 3.3: \( \text{oscillationOpeningAngle} = \pi / 6 \) \( \text{rotationOpeningAngle} = \pi / 3 \)

Using the best result from this test, the difference between the Euclidean distance and Manhattan distance will be tested with the two following setups:

Setup 4.1: \( \text{euclideanDistance} = \text{true} \)
Setup 4.2: \( \text{euclideanDistance} = \text{false} \)

The last variable that will be tested is the desired temperature. As already mentioned, the value for this variable will greatly affect both the execution time and the look of the graph. The following four setups will be tested with the best values from all previous tests:

Setup 5.1: \( \text{desiredTemp} = 3 \)
Setup 5.2: \( \text{desiredTemp} = 8 \)
Setup 5.3: \( \text{desiredTemp} = 16 \)
Setup 5.4: \( \text{desiredTemp} = 32 \)

Setup 5.1 will surely look more stable than Setup 5.4, but the question is if the extra time it takes to reach 3 degrees is worth it. The maximum number of rounds allowed has the value that was recommended in the original paper. However, this constant is somewhat irrelevant in this case since the algorithm will never run for that long with the values chosen with these setups.

The smallest ontology will be tested first, and then the same setup will be tested on the bigger ontologies. The best setup for the smallest ontology is not necessarily the best setup overall, but this will unfortunately have to be ignored since the evaluation is very time-consuming, especially for large graphs. GEM-2 is predicted to have very similar performance to GEM when the same number of nodes are moving, therefore, the different setups will not be tested as extensively for GEM-2. One of the algorithms included in Prefux, ForceDirectedLayout, will also be tested for comparison.
4.2 Result

This chapter presents the result of the evaluation, all the figures and tables can be found in Appendix B. The following ontologies have been tested:

- **Ont1**: “oaei2014_FMA_small_overlapping_nci.owl”, 3,696 nodes, 3,693 edges
- **Ont2**: “oaei2014_NCI_small_overlapping_fma.owl”, 6,488 nodes, 4,917 edges
- **Ont3**: “oaei2014_FMA_small_overlapping_snomed.owl”, 10,157 nodes, 10,154 edges
- **Ont4**: “oaei2014_SNOMED_small_overlapping_fma.owl”, 13,412 nodes, 16,281 edges
- **Ont5**: “oaei2014_NCI_small_overlapping_snomed.owl”, 23,958 nodes, 18,930 edges

GEM-2 is predicted to be as fast as GEM when it is moving the same number of nodes, therefore there will not be many GEM-2 benchmarks. The time it takes to read the ontology-file and set up the application is not included in these benchmarks. Also worth noting is that the graphs produced by the algorithm can look very different from each other since there is a lot of randomization involved.

5 samples are taken with every setup, unless otherwise stated. The samples are taken using a 2012 MacBook Air. The computer has an Intel Core i5-3427U with a base speed of 1.8 GHz, and 4 GB of 1600MHz DDR3L memory. It is running OS X El Capitan, version 10.11.6. JavaFX version 8.0.60-b27 was used along with Eclipse version 4.5.2. Since the samples are taken on a laptop, there is a chance that the CPU will throttle and have slightly decreased performance.

The initial configuration had the following values:

```java
desiredTemp = 3;
desiredEdgeLength = 128;
gravitationalConstant = (double) 1/16;
maxTemp = 256;
oscillationOpeningAngle = Math.PI/2;
rotationOpeningAngle = Math.PI;
oscillationSensitivity = 1.1;
rotationSensitivity = (double) 1/(2*nodeList.size());
maxRounds = nodeList.size() * 4;
euclideanDistance = true;

Vertex {
    temp = 1024;
}
```
5 Discussion

The algorithm, the application and the method used will be discussed and evaluated in this chapter. Possible changes and improvements will also be presented.

5.1 Algorithm

As previously described, the criteria for the algorithm are that nodes and edges should be distributed evenly, the number of edge crossings should be minimal and symmetry should be displayed if possible. From figure 6 we can see that the algorithm works well for the smallest ontology. Here the nodes are distributed evenly, the number of edge-crossings is small and, since the nodes seem to spread out evenly from the barycenter, it has some degree of symmetry as well.

Figure 7 shows a bigger ontology with more disconnected parts than Ont-1 and a desired temperature of 16. The disconnected parts do not drift too far away, which means that the gravitational constant works as intended. The center of the graph seems more dense than the graph in figure 6, and it has more edge crossings as well. Sparse graphs seem to be easier to layout than normal or dense graphs, which makes perfect sense since denser graphs have more edges that can cause edge crossings.

Figure 8 is somewhat similar to figure 6 in that the graph has a tree-like structure. The desired temperature had to be increased yet again to 32, otherwise the algorithm would take too much time to complete. Figure 9 is more special, it is the only ontology that has been tested in this report that has more edges than nodes. This makes it difficult for the algorithm to minimize the number of edge crossings. Apparently the ontology has a lot of disconnected nodes which seem to form a large circle around the dense center, and this is how the algorithm should work when there are a lot of disconnected nodes. Unfortunately the largest ontology, Ont-5, took too long to run. It also caused the JVM to run out of memory which resulted in a crash, this could however be fixed by increasing the heap size in Eclipse.

ForceDirectedLayout was tested on Ont-1, but problems occurred here as well. The algorithm uses a force simulator to calculate the forces between the nodes, and one of the methods calls itself recursively. This caused a stack overflow error for Ont-1, this is most likely because the number of nodes is so large. The algorithm works on the file “socialnet.xml” which is included in Prefux, it has 129 nodes and 161 edges. It is somewhat laggy and it takes a couple of seconds before the layout finishes. Running GEM on this graph is much faster, it finishes pretty much instantly and the resulting graphs look similar.

Setup 1.1–1.5 all turned out to look similar, so the choice for the best setup was taken based on the average number of rounds required and the standard deviation. Figure 10 shows the differences between Setup 2.1–2.4. We can clearly see that the increased desired edge length on Setup 2.3 and 2.4 makes the graph seem more spread out. Whether this is a good thing or not is hard to say, and it depends on the structure of the graph and its size. If the graph is huge, it could benefit from the higher edge length to separate the nodes and the clusters more. If the graph is small however, the distance between the nodes could be unnecessarily long. The lower gravitational constant on Setup 2.2 and 2.4 seem to separate the node-clusters more. This should be a good thing, since it makes it easier for the user to distinguish between different parts of the graph.
Setup 3.1–3.3 produced similar looking graphs, but the time and number of rounds required were very different. The wider the detection angles for rotations and oscillations, the faster the algorithm seems to be. It also seems like the standard deviation is a lot higher when the angles are more narrow. When the angles are wide, rotations and oscillations will trigger more often than if the angles are narrow. Since the angles are very wide in setup 3.1, these events will trigger most of the time which makes the algorithm seem more predictable and consistent. If they do not trigger very often, they will appear more random which makes it more important to have a good initial graph when the algorithm starts. Based on these samples, setup 3.1 seems to be far better than the other setups.

Figure 11 shows the difference between Euclidean distance and Manhattan distance. The nodes are more evenly distributed using the Euclidean distance, and the graph looks more compact using the Manhattan distance. We can see that the distance between the nodes that are not directly connected to each other but have the same parent are uniform, especially when the Euclidean distance is used. The nodes are closer to each other when using the Manhattan distance. This makes sense because the Manhattan distance between two nodes is always greater than or equal to the Euclidean distance, therefore the algorithm “thinks” that the nodes are further away from each other than they actually are. Therefore, the resulting graph is more compact. From table 4 we can see that the average number of rounds and the average time required is both lower for Manhattan. The average number of rounds executed per second was 4.63 for Euclidean and 5.12 for Manhattan, but since the first criterion was that the nodes and edges should be distributed evenly, the Euclidean distance should be preferred even though it is slightly slower to calculate than the Manhattan distance.

Figure 12 shows an interesting property of the algorithm. It shows a part of the graph in four different states as the global temperature decreases. We can see that there are a lot of edge crossings when the temperature is higher, but as it decreases it seems like the graph is unfolding and the edge crossings start to disappear. This can clearly be seen when looking at the difference between 8 and 3 degrees. This is because the nodes are spreading out away from the barycenter in combination with the repulsive forces which makes sure that the nodes and clusters are not too close to each other. The fact that the clusters spread out in the same general direction also makes it easy to navigate the graph when zoomed in, because the user can always see in which direction the center of the graph is.

Changing the values of the variables can greatly affect both the look of the graph and the time it takes for the algorithm to finish. Whether the best setup for Ont-1 is also the best setup for the other ontologies is difficult to say. One of the most important variables to consider when switching to an ontology of a different size is the desired temperature. As we can see from the benchmarks, the larger ontologies need to have a higher temperature threshold or else they would take far too long to finish. It is important to find a good balance based on what is more important: the execution time or having a more stable graph.

None of the ontologies tested in this report have been small-world graphs. Frick et al. [10] did however test their implementation on small-world graphs, as can be seen from figure 17-19 in their paper. Those graphs do not have nearly as many nodes as the graphs in this report. As we already know, the more nodes there are, the more difficult it is to create a graph with a minimal amount of edge crossings. It is therefore difficult to say how GEM could handle really large small-world graphs, and whether it would be more efficient than the other algorithms.

Figure 13 shows what a graph can look like after running GEM-2. Only five levels were expanded
at first and after it finished running, the rest of the graph was expanded. The result usually contains a lot of edge crossings. This is because the nodes that were expanded at first are frozen in place, this can make it difficult for the newly visible nodes to move away from the barycenter properly. As we can see from table 5, GEM-2 has very similar performance to GEM when the same number of nodes are moved. Overall, GEM-2 behaves as expected. It performs better than GEM when the user only wants to expand a specific number of levels.

5.2 Application

One of the criteria for the application was that the user should be able to move the graph, zoom in and out and rotate the graph. This functionality has been implemented, but when the graph is big the interaction with it becomes slow. Moving the graph causes the application to lag, zooming and rotating even more so. For this reason, the functionality for rotating the graph is commented out since a rotation event easily triggers when touching the graph with more than one finger. The ArrowRenderer has a bad impact on the smoothness of the application, when it is used instead of the default EdgeRenderer the performance degrades a lot. This should not be a big surprise, since every arrow is a Group that consists of a Line and a Polygon instead of just one Line. Because of this there are a lot more objects on the screen when the graph is big.

Since the user can zoom in and out, the size of the nodes will appear different depending on the zoom level. It is difficult for the user to expand, collapse or select nodes when the graph is zoomed out, since the nodes will appear very small. When the graphs are huge, it is difficult to make all the nodes clearly visible inside the frame of the display. As it is now, the user has to zoom in to make the nodes more visible. All the nodes have the same size in this implementation.

In the original paper the authors said that they measured a speed of 120,000 iterations per second [6]. In this evaluation, a speed of approximately 15,000 iterations per second was measured. The big difference is that they did not test the algorithm on very large graphs. The larger the graph is, the longer it takes to finish one iteration, since most of the time is spent on calculating the repulsive forces which will take longer the more nodes there are in the graph.

5.3 Method

As already mentioned in the Introduction-chapter, the waterfall model was used for this work. It takes a lot of time to get to the testing phase, and it is very time-consuming to then go back if something needs to be changed. The application starts lagging when the graphs are really big, and this becomes a problem that is difficult to solve within the time frame of this work. It is difficult to say whether this is a problem with JavaFX or if Prefux is just badly optimized. In hindsight, it would have been a good idea to create a test-application outside of Prefux with a couple of thousand nodes and edges. If that would lag, it would most likely be a problem with JavaFX.

One thing that might cause confusion is the fact that some of the touch-functionality is implemented in GemMain, and some in GemControl. The reason why they are separated is because they have different responsibilities. The touch-functionality in GemMain is simple and it only needs access to the Pane. The graph-object is not modified here; it only changes position, scale and rotation within the Pane. Meanwhile, the touch-functionality in GemControl only needs access to the graph-object, because the functionalities there need to be able to modify the graph. This file does not need access
to the Pane because it does not matter which container the graph is in. Placing all of the touch- functionality in only one of the files would be possible, but it would increase the coupling between them. There is a method in ControlAdapter called “event” which could be implemented in GemControl, the method is supposed to be called whenever the user touches any part of the area where the graph is contained. However, this method does not work at all in the current version of Prefux. If it were to be fixed, all the touch-functionalities could likely be moved to GemControl, although they would have to be modified. ArrowRenderer is not GEM-specific, it can be used with other algorithms in Prefux.

Prefuse already has a renderer for drawing directed edges, unlike Prefux where it had to be implemented. The renderer from Prefuse could not easily be ported to work in Prefux, therefore a completely new renderer had to be created. As mentioned some of the methods in Prefux are not implemented, for example the two methods called “zoomAbs” and “panToAbs” in the file “FxDisplay”. These methods, if implemented, could be used to zoom and move the graph internally in the Prefux-display instead of having to use the event handlers for the main window.

5.4 Possible Improvements and Changes

Even though the algorithm and the application are satisfactory, there are always things that can be improved or changed. New setups can always be tried and more than five samples can also be taken to compare them to each other.

The way expanding, collapsing and selecting nodes work could be reversed so that nodes are expanded or collapsed when a long press is detected and nodes are selected by tapping them. If it is more important to be able to select nodes instead of expanding and collapsing them, this reversal should make more sense and make it easier for the user. A long press can be difficult to perform on a very sensitive screen, since very small movements can be detected even if the finger is stationary which would result in the long press being cancelled. Selecting nodes is also very basic right now, it can be extended with more functionality.

In the original paper, the authors made the algorithm rely on integer arithmetics, which should be faster than the double arithmetics that this implementation of the algorithm uses [10]. It would be difficult to switch to integer arithmetics here since the coordinates are stored as doubles in Prefux. However, replacing the doubles with floats could possibly improve the performance, since a float only takes up half the space of a double. This could result in more coordinates being stored in the CPU cache which would reduce the time it takes for the CPU to fetch the coordinates when doing the calculations.

The ArrowRenderer could possibly be modified to reduce the number of objects on the screen, even though it would be difficult to do so. The idea would be to represent an arrow with just a Polygon, instead of a Polygon and a Line contained in a Group. Since a Polygon can take basically any form, it could be formed as an arrow that is placed between its source- and target-node. Whenever one of these nodes move, the polygon would have to be adjusted so that the length and rotation of it is correct. This could potentially be very difficult and time-consuming to implement, but it would result in less objects on the screen.

Recursive methods are used when the user collapses or expands the graph. If the graph has a cycle, the base case would never be reached which would result in the application crashing due to running out of memory. An improvement would be to check every child whether the method has already
traversed it.

The algorithm could be modified to make it deterministic by removing all the randomization. In this implementation, randomization is used in three places: the initial placement of the nodes, the shuffling of the node-list before every round and the random disturbance vectors that are added to the impulses. These could all be removed, but it is difficult to say how it would affect the algorithm's performance and the resulting graph. The algorithm would most likely benefit from placing the nodes in a grid layout in the initialization rather than placing all the nodes in the exact same place.

Another possibility would be to optimize the algorithm for multithreading. For example, one modification would be to save the coordinates of every node in a list at the start of every round, and then fetch all the coordinates from that list when the forces between the nodes are calculated. The coordinates in that list would only be updated at the start of the following round. All the new impulses would be calculated based on what the graph looked like at the end of the last round. This would of course change how the algorithm works, but it would allow multiple threads to move nodes independently during every round.

The nodes and the clusters could be painted in different colors or have their sizes changed in order to make the graphs even more visually pleasing. This would make it even easier to distinguish which nodes are directly related to each other. This is not an area I have touched, but some sort of algorithm could possibly be used to achieve this.

The algorithm or the application could be modified to only run for a specific number of seconds, instead of having a temperature threshold. That way the user would not have to guess how long the algorithm takes to finish, which makes it a bit more predictable. The resulting graph would most likely look better on faster computers, since they can execute more rounds per seconds than slower computers.

GEM-2 can also be modified. Instead of specifying how many levels are expanded initially, the user could specify how many nodes should be expanded initially at most. The algorithm could then automatically expand the right amount of levels as long as the number of nodes do not exceed that number. Both versions of the algorithm could have a timer that stops the algorithm after a set number of seconds, to avoid the algorithm taking too much time to finish. GEM-2 could also be modified to move all the visible nodes when the user expands something, regardless if they are already fixed in place. This would take longer, but the resulting graph would most likely have less edge crossings.

Creating a custom class called Vertex is just one way to allow the use of additional properties needed for the algorithm. The parent class of VisualItem is called Tuple and it has methods for setting and getting custom properties from the table that contains all the nodes. Instead of using Vertex, these methods could be used to set and get the additional properties. The table in Prefux has support for storing arbitrary objects, which means that it should be able to store the lists containing the neighboring nodes. This solution would most likely result in slightly less memory being used, since the Vertex-objects would not have to be created. It is however hard to predict which solution would be the fastest and how big the difference would be.

Instead of fetching the visual objects every time they have to be modified, they could be fetched once and then stored in some type of list. When the user then collapses a node, for example, all the visual objects of its children could be fetched from that list to make the process faster. The nodes
could also have different sizes depending on how many children they have. That would make it easier to identify the parent nodes and it could make it easier for the user to get a better overview of the graph. It would be important to not make the nodes too big though, or else they could overlap their children or other nodes.

The gravitational constant makes sure that disconnected graphs do not float away too far from each other, but it has to be chosen carefully. One possible improvement could be to move any disconnected parts of the graph closer to the barycenter after the algorithm has terminated, without causing any overlapping or edge crossings. Using this technique, it would not matter if any disconnected parts float far away while the algorithm is running. It could be useful if there are a lot of disconnected parts, but it would also be time-consuming and difficult to implement.
6 Conclusions and Future Work

My task was to create an application for visualizing huge graphs using a fitting algorithm. It was very important that the resulting graph should look visually pleasing, and it should also be produced within a reasonable time frame. In order to do this, I had to dive into an area of research that is relatively new to me. I studied what ontology engineering is about, as well as ontology alignment to better motivate why this work is important. I had to do a background and requirement analysis and learn about graph drawing theory. I studied and compared different graph visualization algorithms according to a list of criteria. Two different software platforms, Gephi and Prefux, were also introduced and compared to each other. The conclusion was that GEM was deemed to be the best suited algorithm for this work, along with Prefux.

After that, the design phase started. I had to delve into Prefux to understand how it worked before trying to implement the algorithm. During the implementation, I encountered issues that I had to deal with along with new ideas on how to make the application more effective and usable. After GEM had been implemented I started working on a new version of the algorithm, GEM-2, based on suggestions from my mentor.

The evaluation shows that the choice of algorithm and platform turned out to be good. The algorithm chosen is customizable enough to give the user a lot of flexibility with graphs of varying sizes and characteristics. In order to get the most out of it, the user needs to spend some time studying the algorithm and how it works to know the differences between the different setups and variables. Gephi has been updated since the choice of platform was made. However, it does not change the outcome of the comparison with Prefux since there is still no built-in support for JavaFX.

The three most important findings in making sure the algorithm and the application is as fast as possible were the following:

- The visualization does not need to be updated continually or after every round. A lot of time can be saved by only updating the visualization once, after the algorithm has finished.
- The section of the code that is executed the most, in this case the calculation of the repulsive forces, needs to be as fast as possible.
- It is beneficial to have fast access to the nodes and their coordinates, since they are used very often in the algorithm.

GEM-2 is just a quick implementation of an idea that was conceived during the implementation of the original algorithm. There are many things that can be changed or improved further, as we saw in the previous chapter. Additional algorithms can also be implemented, and possibly combined with GEM. For example, some algorithms could be more efficient at the initial stage of the visualization. After the initial stage is over, the application can switch over to another algorithm. The hope is that the result of this thesis will be used in future work where further analysis is made in this area.
References


[14] Microsoft, "Use touchscreen gestures on Microsoft Surface"


[18] Oracle, "Handling JavaFX Events"


[30] Oracle, "Math (Java Platform SE 8)"
Appendix A – Links

The code for everything from the implementation-chapter can be found in the following Git repository: https://github.com/SoEgd/Prefux

The application created for testing the touch-functionalities can be found in the following Git repository: https://github.com/SoEgd/JavaFX-test-application

The ontologies used in this report, along with many others, can be downloaded from the following website: http://oaei.ontologymatching.org/2014/largebio/index.html

The Jena-library that was used in this report can be downloaded from the following website: https://jena.apache.org/download/index.cgi
Figure 6: An overview of Ont-1 (3,696 nodes, 3,693 edges) with a desired temperature of 3.
Figure 7: An overview of Ont-2 (6,488 nodes, 4,917 edges) with a desired temperature of 16.
Figure 8: An overview of Ont-3 (10,157 nodes, 10,154 edges) with a desired temperature of 32.
Figure 9: An overview of Ont-4 (13,412 nodes, 16,281 edges) with a desired temperature of 128.
Figure 10: The difference between Setup 2.1–2.4 on Ont-1.
Figure 11: The difference between Euclidean / Manhattan distance on Ont-1.
Figure 12: The difference between different temperature thresholds on Ont-1.
Figure 13: Gem-2, collapsed and expanded.
<table>
<thead>
<tr>
<th>Setup1.1</th>
<th>Setup1.2</th>
<th>Setup1.3</th>
<th>Setup1.4</th>
<th>Setup1.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>1024</td>
<td>temp</td>
<td>1024</td>
<td>temp</td>
</tr>
<tr>
<td>maxTemp</td>
<td>256</td>
<td>maxTemp</td>
<td>512</td>
<td>maxTemp</td>
</tr>
<tr>
<td>desiredTemp</td>
<td>3</td>
<td>desiredTemp</td>
<td>3</td>
<td>desiredTemp</td>
</tr>
<tr>
<td>desiredEdgeLength</td>
<td>128</td>
<td>desiredEdgeLength</td>
<td>128</td>
<td>desiredEdgeLength</td>
</tr>
<tr>
<td>gravitationalConstant</td>
<td>1 / 16</td>
<td>gravitationalConstant</td>
<td>1 / 16</td>
<td>gravitationalConstant</td>
</tr>
<tr>
<td>oscillationOpeningAngle</td>
<td>PI / 2</td>
<td>oscillationOpeningAngle</td>
<td>PI / 2</td>
<td>oscillationOpeningAngle</td>
</tr>
<tr>
<td>rotationOpeningAngle</td>
<td>PI</td>
<td>rotationOpeningAngle</td>
<td>PI</td>
<td>rotationOpeningAngle</td>
</tr>
<tr>
<td>oscillationSensitivity</td>
<td>1.1</td>
<td>oscillationSensitivity</td>
<td>1.1</td>
<td>oscillationSensitivity</td>
</tr>
<tr>
<td>euclideanDistance</td>
<td>TRUE</td>
<td>euclideanDistance</td>
<td>TRUE</td>
<td>euclideanDistance</td>
</tr>
<tr>
<td>rounds</td>
<td>time(s)</td>
<td>rounds</td>
<td>time(s)</td>
<td>rounds</td>
</tr>
<tr>
<td>532</td>
<td>125</td>
<td>467</td>
<td>104</td>
<td>569</td>
</tr>
<tr>
<td>624</td>
<td>133</td>
<td>653</td>
<td>151</td>
<td>519</td>
</tr>
<tr>
<td>526</td>
<td>124</td>
<td>625</td>
<td>147</td>
<td>609</td>
</tr>
<tr>
<td>482</td>
<td>106</td>
<td>649</td>
<td>137</td>
<td>530</td>
</tr>
<tr>
<td>708</td>
<td>159</td>
<td>441</td>
<td>102</td>
<td>554</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>average</td>
<td></td>
<td>average</td>
</tr>
<tr>
<td>574.4</td>
<td>129.4</td>
<td>567.0</td>
<td>128.2</td>
<td>556.2</td>
</tr>
<tr>
<td>stdev</td>
<td></td>
<td>stdev</td>
<td></td>
<td>stdev</td>
</tr>
<tr>
<td>90.8</td>
<td>19.3</td>
<td>104.1</td>
<td>23.6</td>
<td>35.4</td>
</tr>
</tbody>
</table>

*Table 1: Benchmark for different values for temp and maxTemp.*
<table>
<thead>
<tr>
<th>Setup2.1</th>
<th><strong>Setup2.2</strong></th>
<th>Setup2.3</th>
<th>Setup2.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>256</td>
<td>temp</td>
<td>256</td>
</tr>
<tr>
<td>maxTemp</td>
<td>256</td>
<td>maxTemp</td>
<td>256</td>
</tr>
<tr>
<td>desiredTemp</td>
<td>3</td>
<td>desiredTemp</td>
<td>3</td>
</tr>
<tr>
<td>desiredEdgeLength</td>
<td>128</td>
<td>desiredEdgeLength</td>
<td>128</td>
</tr>
<tr>
<td>gravitationalConstant</td>
<td>1 / 16</td>
<td>gravitationalConstant</td>
<td>1 / 32</td>
</tr>
<tr>
<td>oscillationOpeningAngle</td>
<td>PI / 2</td>
<td>oscillationOpeningAngle</td>
<td>PI / 2</td>
</tr>
<tr>
<td>rotationOpeningAngle</td>
<td>PI</td>
<td>rotationOpeningAngle</td>
<td>PI</td>
</tr>
<tr>
<td>oscillationSensitivity</td>
<td>1.1</td>
<td>oscillationSensitivity</td>
<td>1.1</td>
</tr>
<tr>
<td>euclideanDistance</td>
<td>TRUE</td>
<td>euclideanDistance</td>
<td>TRUE</td>
</tr>
<tr>
<td>rounds</td>
<td>569</td>
<td>rounds</td>
<td>608</td>
</tr>
<tr>
<td></td>
<td>519</td>
<td></td>
<td>580</td>
</tr>
<tr>
<td></td>
<td>609</td>
<td></td>
<td>631</td>
</tr>
<tr>
<td></td>
<td>530</td>
<td></td>
<td>639</td>
</tr>
<tr>
<td></td>
<td>554</td>
<td></td>
<td>660</td>
</tr>
<tr>
<td>average</td>
<td>556.2</td>
<td>average</td>
<td>623.6</td>
</tr>
<tr>
<td></td>
<td>124.8</td>
<td></td>
<td>134.8</td>
</tr>
<tr>
<td>stdev</td>
<td>35.4</td>
<td>stdev</td>
<td>30.7</td>
</tr>
<tr>
<td></td>
<td>9.2</td>
<td></td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Benchmark for different values for desiredEdgeLength and gravitationalConstant.
<table>
<thead>
<tr>
<th><strong>Setup3.1</strong></th>
<th><strong>Setup3.2</strong></th>
<th><strong>Setup3.3</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>temp 256</td>
<td>temp 256</td>
<td>temp 256</td>
</tr>
<tr>
<td>maxTemp 256</td>
<td>maxTemp 256</td>
<td>maxTemp 256</td>
</tr>
<tr>
<td>desiredTemp 3</td>
<td>desiredTemp 3</td>
<td>desiredTemp 3</td>
</tr>
<tr>
<td>desiredEdgeLength 128</td>
<td>desiredEdgeLength 128</td>
<td>desiredEdgeLength 128</td>
</tr>
<tr>
<td>gravitationalConstant 1 / 32</td>
<td>gravitationalConstant 1 / 32</td>
<td>gravitationalConstant 1 / 32</td>
</tr>
<tr>
<td>oscillationOpeningAngle PI / 2</td>
<td>oscillationOpeningAngle PI / 4</td>
<td>oscillationOpeningAngle PI / 6</td>
</tr>
<tr>
<td>rotationOpeningAngle PI</td>
<td>rotationOpeningAngle PI / 2</td>
<td>rotationOpeningAngle PI / 3</td>
</tr>
<tr>
<td>oscillationSensitivity 1.1</td>
<td>oscillationSensitivity 1.1</td>
<td>oscillationSensitivity 1.1</td>
</tr>
<tr>
<td>euclideanDistance TRUE</td>
<td>euclideanDistance TRUE</td>
<td>euclideanDistance TRUE</td>
</tr>
<tr>
<td>rounds 608</td>
<td>rounds 1528</td>
<td>rounds 745</td>
</tr>
<tr>
<td>time(s) 138</td>
<td>time(s) 346</td>
<td>time(s) 153</td>
</tr>
<tr>
<td>rounds 580</td>
<td>rounds 1027</td>
<td>rounds 600</td>
</tr>
<tr>
<td>time(s) 133</td>
<td>time(s) 237</td>
<td>time(s) 138</td>
</tr>
<tr>
<td>rounds 631</td>
<td>rounds 1110</td>
<td>rounds 1184</td>
</tr>
<tr>
<td>time(s) 136</td>
<td>time(s) 256</td>
<td>time(s) 244</td>
</tr>
<tr>
<td>rounds 639</td>
<td>rounds 897</td>
<td>rounds 1738</td>
</tr>
<tr>
<td>time(s) 133</td>
<td>time(s) 182</td>
<td>time(s) 356</td>
</tr>
<tr>
<td>rounds 660</td>
<td>rounds 971</td>
<td>rounds 774</td>
</tr>
<tr>
<td>time(s) 134</td>
<td>time(s) 224</td>
<td>time(s) 158</td>
</tr>
<tr>
<td>average 623.6</td>
<td>average 1106.6</td>
<td>average 1008.2</td>
</tr>
<tr>
<td>stdev 30.7</td>
<td>stdev 248.1</td>
<td>stdev 462.1</td>
</tr>
<tr>
<td>stdev 2.2</td>
<td>stdev 60.7</td>
<td>stdev 91.7</td>
</tr>
</tbody>
</table>

Table 3: Benchmark for different values for the opening angles.
<table>
<thead>
<tr>
<th></th>
<th>Setup4.1</th>
<th></th>
<th>Setup4.2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>temp</strong></td>
<td>256</td>
<td><strong>temp</strong></td>
<td>256</td>
</tr>
<tr>
<td><strong>maxTemp</strong></td>
<td>256</td>
<td><strong>maxTemp</strong></td>
<td>256</td>
</tr>
<tr>
<td><strong>desiredTemp</strong></td>
<td>3</td>
<td><strong>desiredTemp</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>desiredEdgeLength</strong></td>
<td>128</td>
<td><strong>desiredEdgeLength</strong></td>
<td>128</td>
</tr>
<tr>
<td><strong>gravitationalConstant</strong></td>
<td>1 / 32</td>
<td><strong>gravitationalConstant</strong></td>
<td>1 / 32</td>
</tr>
<tr>
<td><strong>oscillationOpeningAngle</strong></td>
<td>PI / 2</td>
<td><strong>oscillationOpeningAngle</strong></td>
<td>PI / 2</td>
</tr>
<tr>
<td><strong>rotationOpeningAngle</strong></td>
<td>PI</td>
<td><strong>rotationOpeningAngle</strong></td>
<td>PI</td>
</tr>
<tr>
<td><strong>oscillationSensitivity</strong></td>
<td>1.1</td>
<td><strong>oscillationSensitivity</strong></td>
<td>1.1</td>
</tr>
<tr>
<td><strong>euclideanDistance</strong></td>
<td>TRUE</td>
<td><strong>euclideanDistance</strong></td>
<td>FALSE</td>
</tr>
<tr>
<td><strong>rounds</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>608</td>
<td>138</td>
<td>644</td>
<td>110</td>
</tr>
<tr>
<td>580</td>
<td>133</td>
<td>597</td>
<td>119</td>
</tr>
<tr>
<td>631</td>
<td>136</td>
<td>436</td>
<td>88</td>
</tr>
<tr>
<td>639</td>
<td>133</td>
<td>513</td>
<td>107</td>
</tr>
<tr>
<td>660</td>
<td>134</td>
<td>536</td>
<td>108</td>
</tr>
<tr>
<td><strong>average</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>623.6</td>
<td>134.8</td>
<td>545.2</td>
<td>106.4</td>
</tr>
<tr>
<td><strong>stdev</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30.7</td>
<td>2.2</td>
<td>79.8</td>
<td>11.3</td>
</tr>
</tbody>
</table>

Table 4: Benchmark for different values for euclideanDistance.
<table>
<thead>
<tr>
<th>Setup5.1</th>
<th>Setup5.2</th>
<th>Setup5.3</th>
<th>Setup5.4</th>
<th>(GEM-2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp</td>
<td>temp</td>
<td>temp</td>
<td>temp</td>
<td>temp</td>
</tr>
<tr>
<td>maxTemp</td>
<td>maxTemp</td>
<td>maxTemp</td>
<td>maxTemp</td>
<td>maxTemp</td>
</tr>
<tr>
<td>desiredTemp</td>
<td>3</td>
<td>desiredTemp</td>
<td>8</td>
<td>desiredTemp</td>
</tr>
<tr>
<td>desiredEdgeLength</td>
<td>128</td>
<td>desiredEdgeLength</td>
<td>128</td>
<td>desiredEdgeLength</td>
</tr>
<tr>
<td>gravitationalConstant</td>
<td>1 / 32</td>
<td>gravitationalConstant</td>
<td>1 / 32</td>
<td>gravitationalConstant</td>
</tr>
<tr>
<td>oscillationOpeningAngle</td>
<td>PI / 2</td>
<td>oscillationOpeningAngle</td>
<td>PI / 2</td>
<td>oscillationOpeningAngle</td>
</tr>
<tr>
<td>rotationOpeningAngle</td>
<td>PI</td>
<td>rotationOpeningAngle</td>
<td>PI</td>
<td>rotationOpeningAngle</td>
</tr>
<tr>
<td>oscillationSensitivity</td>
<td>1.1</td>
<td>oscillationSensitivity</td>
<td>1.1</td>
<td>oscillationSensitivity</td>
</tr>
<tr>
<td>euclideanDistance</td>
<td>TRUE</td>
<td>euclideanDistance</td>
<td>TRUE</td>
<td>euclideanDistance</td>
</tr>
<tr>
<td>rounds</td>
<td>time(s)</td>
<td>rounds</td>
<td>time(s)</td>
<td>rounds</td>
</tr>
<tr>
<td>569</td>
<td>133</td>
<td>369</td>
<td>90</td>
<td>269</td>
</tr>
<tr>
<td>725</td>
<td>167</td>
<td>590</td>
<td>139</td>
<td>256</td>
</tr>
<tr>
<td>648</td>
<td>141</td>
<td>437</td>
<td>98</td>
<td>291</td>
</tr>
<tr>
<td>642</td>
<td>139</td>
<td>468</td>
<td>104</td>
<td>293</td>
</tr>
<tr>
<td>622</td>
<td>147</td>
<td>471</td>
<td>112</td>
<td>261</td>
</tr>
<tr>
<td>average</td>
<td>641.2</td>
<td>average</td>
<td>467.0</td>
<td>108.6</td>
</tr>
<tr>
<td>stdev</td>
<td>56.2</td>
<td>stdev</td>
<td>80.1</td>
<td>18.8</td>
</tr>
</tbody>
</table>

Table 5: Benchmark for different values for desiredTemp.
<table>
<thead>
<tr>
<th>Ont-1</th>
<th>Ont-2</th>
<th>Ont-2</th>
<th>Ont-3</th>
<th>Ont-4</th>
<th>Ont-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>nodes</td>
<td>3696</td>
<td>nodes</td>
<td>6488</td>
<td>nodes</td>
<td>10157</td>
</tr>
<tr>
<td>edges</td>
<td>3693</td>
<td>edges</td>
<td>4917</td>
<td>edges</td>
<td>10154</td>
</tr>
<tr>
<td>temp</td>
<td>3</td>
<td>temp</td>
<td>3</td>
<td>temp</td>
<td>16</td>
</tr>
<tr>
<td>rounds</td>
<td>608</td>
<td>time(s)</td>
<td>138</td>
<td>rounds</td>
<td>764</td>
</tr>
<tr>
<td></td>
<td>580</td>
<td></td>
<td>133</td>
<td></td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>631</td>
<td></td>
<td>136</td>
<td></td>
<td>312</td>
</tr>
<tr>
<td></td>
<td>639</td>
<td></td>
<td>133</td>
<td></td>
<td>336</td>
</tr>
<tr>
<td></td>
<td>660</td>
<td></td>
<td>134</td>
<td></td>
<td>249</td>
</tr>
<tr>
<td>average</td>
<td>623.6</td>
<td>average</td>
<td>778.4</td>
<td>average</td>
<td>423.8</td>
</tr>
<tr>
<td>stdev</td>
<td>30.7</td>
<td>stdev</td>
<td>112.4</td>
<td>stdev</td>
<td>80.8</td>
</tr>
</tbody>
</table>

*after 127 rounds and 50 minutes: temp = 236

Table 6: Benchmark for the different ontologies.