

Comparison of Visualization Algorithms for Graphs and Implementation of Visualization Algorithm for Multi-Touch table using JavaFX

Daniel Sund

Advisor, Valentina Ivanova
Examiner, Patrick Lambrix

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

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

Ontologies are representations of specific domains. They help with understanding the concepts within these domains by providing definitions of their components and how they relate to each other. Ontologies are often huge and may contain thousands to hundreds of thousands of concepts. In order to easily understand them, they can be visualized as graphs. To effectively visualize these graphs, an application can use a type of graph drawing algorithm that follows pre-established criteria.

This thesis studies articles about the problem of graph visualization to establish specific criteria for algorithms to satisfy. Existing algorithms are then compared with these criteria, to determine which algorithm is most suitable for implementation. Based on the comparison evaluation, the FM³ Algorithm is then implemented in Prefux. Extensions to Prefux's input handler and touch functionality are also added to read files in the OWL format and enable interaction with the visualized graph on a state-of-the-art multi-touch table respectively.

Contents

- 1. Introduction..... 1
 - 1.1 Motivation..... 1
 - 1.2 Goals..... 1
 - 1.3 Limitations..... 2
 - 1.4 Report Structure..... 2
- 2. Background..... 3
 - 2.1 Ontology..... 3
 - 2.2 Small World Networks..... 4
 - 2.3 Force-Directed Algorithms..... 4
 - 2.4 The Prefuse Library..... 5
 - 2.5 Touch Functionality..... 6
- 3. Theory..... 7
 - 3.1 Criteria..... 7
 - 3.2 Algorithms..... 8
 - 3.2.1 Fruchterman-Reingold Algorithm..... 8
 - 3.2.2 Davidson-Harel Algorithm..... 8
 - 3.2.3 Kamada-Kawai Algorithm..... 8
 - 3.2.4 The GEM Algorithm..... 9
 - 3.2.5 The FMS Algorithm..... 9
 - 3.2.6 The GRIP Algorithm..... 9
 - 3.2.7 The FM³ Algorithm..... 10
 - 3.3 Related Works..... 10
 - 3.4 Comparison and Conclusion..... 10
- 4. Method..... 13
 - 4.1 The Workspace..... 13
 - 4.2 Parsing OWL Files..... 14
 - 4.3 Implementation of Algorithm..... 17
 - 4.4 Touch Functionality..... 23
 - 4.5 Algorithm Comparison..... 24
- 5. Result..... 27
 - 5.1 The Workspace..... 27
 - 5.2 Parsing OWL Files..... 29
 - 5.3 Implementation of Algorithm..... 29
 - 5.4 Touch Functionality..... 30
 - 5.5 Algorithm Comparison..... 31

6. Discussion	33
6.1 Result	33
6.2 Method	34
6.3 References	35
6.4 Other Perspectives	35
7. Conclusions.....	37
References.....	38
Appendix A. Test-Program (PrefuxTest.java)	41
Appendix B. GraphOWLReader	44
Appendix C. FM ³ Algorithm	47
Appendix D. Touch Controls.....	63
DragControl.java.....	63
FocusControl.java	66
ZoomControl.java	70
Appendix E. Time & Edge-Crossings Counter	72
Time Counter	72
Edge-Crossings Counter (for FM ³ Algorithm)	72
Edge-Crossings Counter (for Prefux's Algorithm).....	73
Appendix F. Tested Graphs.....	75

List of Figures

Figure 1. A social network (small world network) drawn with Prefuse's own force-directed algorithm.	5
Figure 2. Table showing the different runtime complexities for each algorithm.	11
Figure 3. How to use Gradle from command prompt.	13
Figure 4. How to add a new library to a NetBeans project.	14
Figure 5. Location of Jena related libraries.	15
Figure 6. The added dependencies to "build.gradle".....	16
Figure 7. How to use the GraphOWLReader.	17
Figure 8. The "Divide and Conquer" strategy of the implemented FM ³ algorithm.....	18
Figure 9. The "Multilevel Step" of the implemented FM ³ algorithm.	19
Figure 10. The "Grid-Embedder" of the implemented FM ³ algorithm.....	20
Figure 11. Force model of the implemented FM ³ algorithm.....	20
Figure 12. Force-Approximation of repulsion forces for the implemented FM ³ algorithm.....	21
Figure 13. The "Multipole-Framework" of the implemented FM ³ algorithm.	22
Figure 14. How to add an algorithm layout to a visualization.	23
Figure 15. How to add control listeners to a display.....	23
Figure 16. Table showing the ontologies and their corresponding amounts of nodes (V) and edges (E).....	24
Figure 17. How to instantiate a new FM ³ layout and individually set all parameters.	25
Figure 18. How the test-program reads an ontology.....	27
Figure 19. How the test-program creates a new visualization.....	27
Figure 20. Test-Program adding a new force-directed layout to the visualization.....	27
Figure 21. Test-Program creates an FxDisplay and adds control listeners to it.	27
Figure 22. How the Test-Program tests the FocusControl.	28
Figure 23. Test-Program adding features to all nodes of the graph.	28
Figure 24. Test-Program running the actions added to the visualization.	28
Figure 25. Visualization of the ontology "crs_dr.owl" with added labels.	29
Figure 26. Visualizations of (a) "edas.owl" and (b) "NCI_small_overlapping_fma.owl" using the FM ³ Algorithm.	29
Figure 27. (a) Original visualization of graph, (b) the same graph after one of its nodes has been dragged.....	30
Figure 28. Focused nodes are colored red. (a) One node of a graph is focused, (b) five nodes being focused.	30
Figure 29. Illustration of Zoom Control. (a) Zoomed out view of graph, (b) zoomed in view of the same graph.	30
Figure 30. Average runtimes in seconds, obtained for both algorithms.....	31
Figure 31. Average amount of edge-crossings, obtained for both algorithms.	32
Figure 32. Graph "crs_dr.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	75
Figure 33. Graph "PCS.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	75
Figure 34. Graph "cmt.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	75
Figure 35. Graph "MICRO.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	76
Figure 36. Graph "linklings.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	76
Figure 37. Graph "confOf.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	76
Figure 38. Graph "MyReview.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	77
Figure 39. Graph "paperdyne.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	77
Figure 40. Graph "sigkdd.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	77

Figure 41. Graph "Cocus.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	78
Figure 42. Graph "confious.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	78
Figure 43. Graph "Conference.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	78
Figure 44. Graph "OpenConf.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	79
Figure 45. Graph "ekaw.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	79
Figure 46. Graph "edas.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	79
Figure 47. Graph "iasted.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	80
Figure 48. Graph "mouse.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.	80
Figure 49. Graph "human.owl", (a) FM ³ Algorithm, (b) Prefux's Algorithm.....	80
Figure 50. Graph "FMA_small_overlapping_nci.owl", drawn with FM ³ Algorithm.	81
Figure 51. Graph "NCI_small_overlapping_fma.owl", drawn with FM ³ Algorithm.....	81
Figure 52. Graph "FMA_small_overlapping_snomed.owl", drawn with FM ³ Algorithm.	82
Figure 53. Graph "SNOMED_small_overlapping_fma.owl", drawn with FM ³ Algorithm.	82
Figure 54. Graph "NCI_small_overlapping_snomed.owl", drawn with FM ³ Algorithm.....	83
Figure 55. Graph "SNOMED_small_overlapping_nci.owl", drawn with FM ³ Algorithm.	83
Figure 56. Graph "NCI_whole_ontology.owl", drawn with FM ³ Algorithm.	84
Figure 57. Partial graph "FMA_whole_ontology.owl", drawn with FM ³ Algorithm.....	84

1. Introduction

This is a Bachelor thesis in computer engineering (16 credits), performed at the Department of Computer and Information Science at Linköping University. For this degree project a visualization algorithm for graph drawing is to be implemented for a state-of-the-art multi-touch table.

1.1 Motivation

In computer and information science, ontologies can be seen as representations of specific domains. Ontologies provide definitions of the domain's concepts and explain how the concepts relate to each other. The field of ontology engineering concerns the design, management and study of such ontologies.

To understand ontologies better, ontology engineering often visualizes them as graphs. Graphs usually represent some domain by associating its components with a set of nodes (entities) and a set of edges (relationships between entities). The readability and understandability of these graphs for human users depend on the algorithm that visualizes these graphs. Therefore, it is important that these algorithms satisfy certain criteria. These are collectively called "aesthetic criteria".

A commonly used method for drawing graphs is by indentation. This method places all nodes vertically in a list and allows to expand each node to show the node's immediate children, which are represented using indentation. The main problem with this method is that it fails to provide an overview of the system and that it requires a lot of navigation over large areas. This layout also does not show relationships between nodes that are not immediate children or parents.

The indentation layout works well for small and simple graphs with a tree-like structure. However, as the graphs become larger and more complex, it would become more difficult to understand the graph using the indentation method and thus it would not satisfy the aesthetic criteria well enough. Instead another algorithm for visualizing graphs would have to be used.

1.2 Goals

The purpose of this thesis is to provide additional information to the research of ontologies and find a suitable algorithm for graph visualization with regards to huge and complex ontologies. The Ontology Alignment Evaluation Initiative [2] is an international initiative that works with ontology engineering. From their website, a set of ontologies can be obtained. The expected result is to be able to visualize these ontologies on the multi-touch table in a way that makes them easy to read and understand (aesthetic criteria) and to visualize them within a reasonable amount of time.

This degree project will be used for previously developed Ontology Engineering systems, SAMBO [28, 31] and RepOSE [23, 29, 30]. This relates to one of the challenges for Ontology Engineering, to support user involvement and allow the user to experiment more with Ontology Engineering techniques [36]. Visualization of ontologies is among the major requirements for this challenge [13, 24, 32].

The main problem for this degree project is thus to compare the different existing algorithms, determine which is most suitable for visualizing the ontologies and implement that algorithm.

The goal of the degree project can be divided into the following questions:

- What graph visualization algorithms exist and which one is most suitable for visualization of large ontologies on the multi-touch table with regards to aesthetic criteria and time-complexity?
- How can the decided upon algorithm be implemented using the Java programming language?

1.3 Limitations

This thesis is not about creating a new algorithm, but instead find and research already existing algorithms in order to determine which is the most suitable for the specified purpose. Specifically, so called force-directed algorithms in the context of drawing small world networks (graphs) will be the main focus of this thesis.

The implementation will use the Prefuse library [6] for the Java programming language. In the context of this degree project, Prefuse is a set of software tools that provides interaction and visualization features for graphs.

The development platform to be used for the implementation is NetBeans IDE 8.0.2 for JavaFX applications. The reason for using JavaFX is because of the touch functionality it provides.

1.4 Report Structure

This report consists of seven chapters. This is the first chapter called Introduction, where the motivation and goals for the project are defined. Concepts that are related to the project and some additional requirements are then further explained in the second chapter, Background.

The third chapter, Theory, discusses other works that are relevant to the project and based on them, defines criteria for the project's comparison part. This chapter also concludes which of the studied algorithms seems most suitable for the project's implementation.

In the fourth chapter, the method used during the project's implementation and comparison parts are described. The results received from both these parts are then presented in the fifth chapter.

Discussions are then made about the used method, received results and the report itself in the sixth chapter. Finally, a conclusion to project is made in the seventh chapter.

2. Background

In this chapter, the project will be further explained. The terms and characteristics of “Ontology”, “Small World Networks” and “Force-Directed Algorithms” will be explained, as well as how they relate to the project. Finally, the Prefuse library that is to be used in this project will be described, as well as those components of it that are relevant to this project.

2.1 Ontology

The term “Ontology” is borrowed from philosophy; in which it can be explained as the study of the nature of existence. The word “ontology” comes from Greek and means “the study of being”. In information science, an ontology is a representation of a specific domain in which the concepts that “exist” within the domain are defined.

Gruber [17] defines ontology as “an explicit specification of a conceptualization”. According to Studer et al. [37] “A ‘conceptualization’ refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. ‘Explicit’ means that the type of concepts used, and the constraints on their use are explicitly defined.”

Ontologies are used in many different scientific fields, such as: Artificial Intelligence, Semantic Web and Medical Informatics. In all these fields, ontologies help to organize information and reduce complexities in databases. In semantic search, ontologies are also used to improve accuracy of search results.

When performing data integration (i.e. combining data from multiple sources), compatibility problems may occur. For instance, the same entity in two separate databases could have different names, alternatively the same name in both databases could have different meanings. Ontologies can help with this issue by resolving differences in concept definitions.

Ontologies are often structured as hierarchies, with several different components. Most commonly an ontology contains the components: concepts, individuals, axioms and relations. Concepts are different types of entities, individuals are specific instances of a concept, axioms describe features or properties that a concept has and relations describe how the concepts or individuals relate to each other.

The ontologies relevant for this thesis (to be used in the final comparison) are in the OWL format [3]. OWL is an abbreviation of “Web Ontology Language” and is a Semantic Web language for representing ontologies. OWL is built upon an XML [7] standard called RDF (Resource Description Framework) [5], to which it adds features that increase interpretability of the ontologies. For example, OWL adds a larger vocabulary to allow more information to be given about the ontology’s concepts and properties.

Currently, the Prefuse library (which is to be used in this project) is not able to read ontologies in the OWL format to be used in the visualization. Therefore, there is an additional requirement to implement a way of parsing an OWL ontology into Prefuse.

2.2 Small World Networks

According to Watts and Strogatz [39], there are two different extremes in the connection topology for a network of nodes (graphs). One is called “regular”, characterized in graphs by having an equal amount of edges connected to each node. The other is called “random”, where the graph has a random distribution of edges connected to each node. However, many network types that are used in the real world are neither of these extremes, but instead belong to a “middle ground”. These are called small world networks.

Small world networks are graphs that connect multiple smaller sub-graphs (clusters) together. According to Auber et al. [8] the defining properties of small world networks depend on two parameters: the average path length and the clustering of the nodes. By this, they mean that small world networks are highly clustered and have a small average path length between its nodes.

Examples of small world networks include: social networks, neural networks and internet databases. Real world examples of this, according to Watts and Strogatz [39], include: the power grid of the western United States and the collaboration graph of actors in feature films. The advantages of systems using small world networks, according to the authors, are enhanced computational power, signal-propagation speed and better ability to synchronize.

In the context of this project, the ontologies obtained from the Ontology Alignment Evaluation Initiative to be used for visualization are small world networks.

2.3 Force-Directed Algorithms

The type of algorithm called Force-Directed Algorithm simulate mechanical forces between the nodes displayed in a graph, such as repulsion forces between nodes and spring forces between link endpoints (edges).

Pohl et al. [34] have compared the readability of force-directed, orthogonal and hierarchical layouts for human users. They did this through an experiment, where test-subjects were asked to perform five different tasks using graphs drawn with each of the three different layouts. The first task was to answer with “yes” or “no” whether the displayed graph contained a node with a specific label. The second task was to identify whether there was a path between two nodes and if so, name the labels of each node along the path. The third task was to identify whether the graph contained a given graph as a subgraph and if so, name the labels of the nodes belonging to that subgraph. The fourth task was similar to the third task, but lacked a visual description of the requested subgraph. The fifth task was to find the node with highest degree (i.e. the node connected with the most number of edges).

Pohl et al. analyzed the tasks based on the subjects’ answers and by tracking their eye-movement. The eye-tracking system used corneal reflection of infrared light to track the movement of the subjects’ eyes, by using cameras mounted on the screen in front of the subjects. They conclude that the force-directed layout outperform the others on most of the analyzed tasks. The only exceptions to this were for the first and fifth tasks, where the results were equal across all three layouts.

In the experiments of Auber et al. [8] they use a force-directed method for drawing graphs. They claim that force-directed algorithms naturally place neighboring nodes close to each other and therefore illustrate small world networks well. However, they also note that force-directed algorithms will not distinguish between edges and will induce the same attractive force between these. This problem is easily solved by assigning weights to edges.

2.4 The Prefuse Library

Prefuse [6] is a software framework for creating dynamic visualizations in the Java programming language. It contains many features, including data modeling, visualization and interaction.

Its creators, Heer et al. [22] explain the library as containing several components. These components simplify visualization of data by providing advanced functions that are commonly used in visualizations. In the context of this thesis, the most relevant components are called Layout, IO and Controls. Layout is relevant since it is the component that handles the placement (layout) of the graph's nodes, IO handles reading and writing from/to formatted files and Controls handles interactions with the Prefuse display.

The Layout component includes several algorithms that use different techniques for visualizing data. One of these algorithms is a force-directed algorithm, which will be used in the final comparison of this thesis. An example of a graph drawn with Prefuse's own force-directed algorithm can be seen in Figure 1. When implementing the algorithm for this project, the algorithm would need to be added to this Layout component.

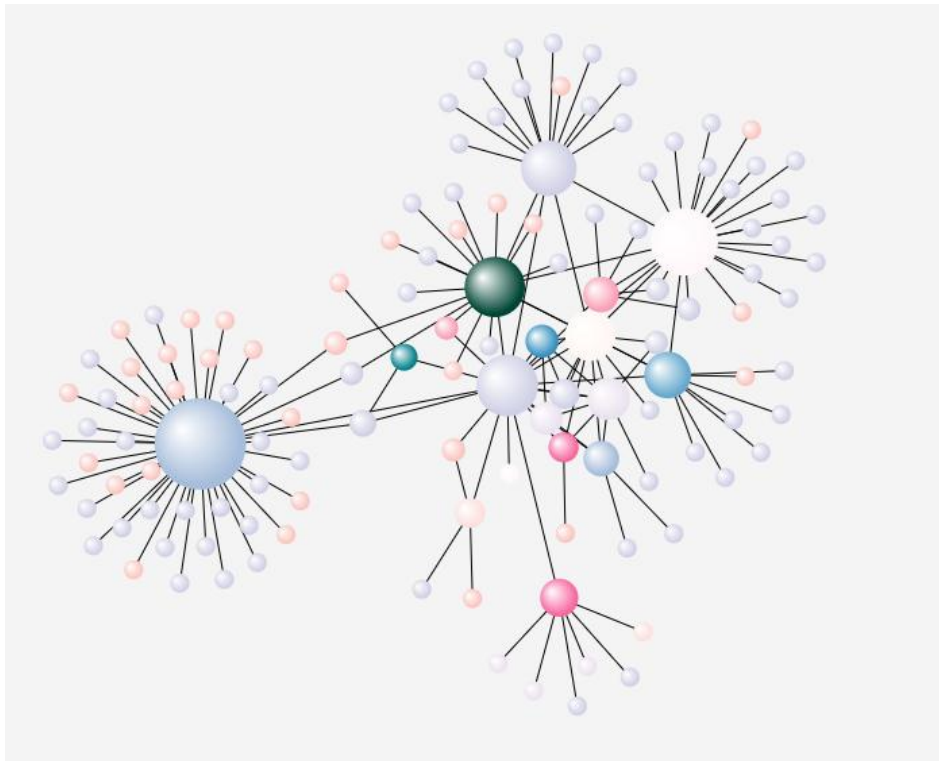


Figure 1. A social network (small world network) drawn with Prefuse's own force-directed algorithm.

The IO component contains interfaces for reading data from formatted files. For example, there is an interface for reading files in the GraphML format in order to create graph objects in Prefuse. As mentioned earlier (Section 2.1) there is a requirement to read ontologies in the OWL format. This functionality would therefore need to be added to the IO component.

Previously, the Prefuse library has been used in combination with Swing, an API that provides graphical user interfaces (GUI) for Java programs. Currently, Swing is being replaced by JavaFX. Therefore, there is an additional requirement that the Prefuse library can be integrated with JavaFX. Some of this work has already been done in an implementation of the Prefuse library in JavaFX, called "Prefux". However, there might be required features that are still missing. For example, since the multi-touch table makes use of touch functionality, there is a requirement for the Prefuse library to handle touch gestures when interacting with the visualized graphs. Therefore, the Controls component of the Prefuse library is relevant, since it handles interactions with the Prefuse display.

2.5 Touch Functionality

To read a visualized graph better and understand its represented ontology, it might not always be enough to just display the graph. To be able to interact with the graph in different ways could significantly increase the understandability of the ontology.

A simple Drag control already exists in the Prefux library, allowing for dragging a visualized graph's nodes to a desired position. The Drag control can be useful in various ways when working with a graph. For example, it can move specific nodes of interest together or position a node somewhere else on the screen to more easily see how that node is connected to others. The current version of the Drag control only works for mouse clicks on computers and doesn't work for touch gestures. Therefore, the Drag control will have to be modified to enable it to work for the multi-touch table.

Sometimes it can be useful to perform certain actions on only some of the graph's nodes. For example, if the graph is large it could be very overwhelming to display information about every single node. Instead one could focus on a small set of nodes and only display information about those nodes. This would require a Focus control to be implemented, to enable the user to select the set of nodes to perform the specific action on.

If the visualized graph is very large, it might not fit the entire graph within the display's boundaries. In this case, a Zoom control would be very useful to enable the user to scale the display. Therefore, an additional requirement will be to implement this Zoom control as well.

When an application that uses JavaFX runs on a platform that recognizes touch gestures, the application will also recognize these gestures and generate touch events. JavaFX is capable of distinguishing what type of gesture was performed and will generate the corresponding type of touch event for the application. For example, if a user puts two fingers on a touch screen and then brings them closer or further away from each other, JavaFX will recognize this as a zoom gesture. The events that are generated are: first a "zoom started" event that is generated as soon as the gesture is recognized, then a "zoom" event that is generated while the user is performing the gesture and finally the "zoom finished" event that is generated whenever the user stops the gesture.

In order to implement the touch functionalities, JavaFX would need to be used to handle the recognition of the touch gestures on the multi-touch table. Once a specific type of touch event is generated by the application, the implementation will need to perform specific operations on Prefux's components depending on what type of touch event was generated. For example, the zoom gesture should change the parameters of Prefux's display to perform the zoom operations.

3. Theory

In this chapter, the criteria used in the comparison will be better specified. The studied force-directed algorithms will be summarized and compared, with regards to the criteria, to decide which algorithm is to be implemented for this project.

3.1 Criteria

Before the comparison can be done, the criteria must be better specified since there are many “aesthetic criteria” and when one is improved another might worsen. This issue is discussed by Beck et al. [10]. In their article, they formulate several aesthetic criteria for graph visualization. Some of these include: reduction of visual clutter, reduction of spatial aliases and maximization of compactness. They conclude that, in practice, some of these criteria are in conflict with each other. They give the example of scalability and that the readability of the visualization can improve if one trades scalability of number of graphs for scalability of number of nodes or edges. By scalability, they mean how well the visualization scales with increasing number of graphs, nodes or edges.

An experiment on aesthetic criteria for graph visualization has been conducted by Purchase [35]. The experiment was conducted by presenting graphs to subjects who then answered questions by performing tasks on the graphs. In the experiment, the tasks were to: find a path from one node to another, determine the minimum amount of edges to remove in order to disconnect one node from another and determine the minimum amount of nodes to remove in order to disconnect one node from another. The author concludes that the experiment strongly suggests that minimization of edge-crossings is the most important aesthetic criterion for improving readability of graphs.

In a follow-up article to that of Purchase’s [35], Ware et al. [38] perform an additional experiment of the same kind as the one performed by Purchase. However, this experiment only focused on path finding tasks, where the subjects were asked to find the shortest path between two nodes of the given graphs. For these tasks, the authors conclude that while edge-crossings is an important aesthetic criterion, it can be worth allowing some edge-crossings if it improves path-continuity. By continuity they mean how much the entire path bends. The authors note that they focused on fewer task types than what Purchase’s experiment did and that minimization of edge-crossings could have a greater importance as a whole for other types of tasks.

With regards to the information above about aesthetic criteria, the comparison in this section of the report will therefore focus on algorithms that reduce edge-crossings.

Since the chosen algorithm should visualize relatively large ontologies, it could become very time consuming. This is because the runtimes of most force-directed algorithms scale according to the amount of nodes and edges in the graph. Therefore, another criterion that will be focused on in this comparison is the time-complexity of the algorithm.

3.2 Algorithms

As summarized by Kobourov [27] and Landesberger et al. [40], there exists several force-directed algorithms that draw graphs using different methods. This large amount of algorithms would take a long time to compare. Therefore, in order to reduce the number of algorithms to investigate, only those algorithms referenced by both articles will be compared.

3.2.1 Fruchterman-Reingold Algorithm

One of the most commonly mentioned algorithms is the Fruchterman-Reingold algorithm [15]. The authors present two different variants of this algorithm. The regular variant has a total runtime of $O(|V|^2 + |E|)$, where $|V|$ is the number of vertices (nodes) in a graph and $|E|$ is the number of edges connecting the vertices. The algorithm uses the concepts of “temperature” and “cooling” to handle the displacement of vertices in a graph. The idea is that the displacement of each vertex is limited to a temperature value that decreases over time, thus the adjustments become smaller as the graph achieves a better layout.

The grid-variant of this algorithm is used to speed up its runtime. It divides the graph drawing area into a grid of squares and only applies repulsion forces between the nodes contained in adjacent squares, thereby excluding the iteration of nodes further away. This reduces the runtime to achieve the time-complexity of $O(|V| + |E|)$. However, this runtime is only achieved under specific assumptions and according to Hachul [18] this is a best-case scenario, the worst-case scenario remains $O(|V|^2 + |E|)$.

The authors acknowledge some generally accepted aesthetic criteria, such as: minimization of edge-crossings, even distribution of nodes in the frame and reflecting symmetry. However, they mention that they do not explicitly strive for these criteria and that the goal of their algorithm is speed and simplicity. They do note that their algorithm is good at: distributing vertices evenly, making edges’ lengths uniform and reflecting symmetry. They also mention that their grid-variant (the faster version) often produces more edge-crossings than the regular version.

3.2.2 Davidson-Harel Algorithm

Davidson and Harel [12] present an algorithm with the runtime of $O(|V|^2|E|)$.

This algorithm uses a method called simulated annealing, for finding a local minimum of the energy function. Simulated annealing is a technique developed for solving large combinatorial optimization problems. It originates in statistical mechanics from a method by Metropolis et al. [33] and was later formulated in general terms by Kirkpatrick et al. [26].

When designing this algorithm, the authors focused on some explicitly selected aesthetic criteria. They focused on: distributing nodes evenly, making edges’ lengths uniform, minimizing edge-crossings and keeping nodes from coming too close to edges. Fruchterman and Reingold [15] claim that this algorithm has slightly better aesthetics and flexibility for complex graphs compared to their own algorithm.

3.2.3 Kamada-Kawai Algorithm

According to Kobourov [27], the Kamada-Kawai algorithm [25] requires an “All-Pair-Shortest-Path” computation that makes it achieve different runtimes depending on which algorithm, for finding the shortest path between vertices in a given graph, it is used in combination with. If used with the so called Floyd-Warshall algorithm it has a runtime of $O(|V|^3)$ and if used with the so called Johnson algorithm it has a runtime of $O(|V|^2 \log|V| + |E||V|)$.

The method of this algorithm is to use spring forces that are proportional to the graph theoretic distance between the nodes. This algorithm does not have separate forces for attraction and repulsion, but instead makes the vertices either attract or repel each other depending on their graph theoretic distance.

The Kamada-Kawai algorithm focuses on producing graphs with good symmetry rather than distributing vertices and edges uniformly or minimizing edge-crossings. However, the authors note that the number of edge-crossings in the produced graphs is relatively small.

3.2.4 The GEM Algorithm

Frick et al. [14] present the GEM algorithm. The authors were not able to specify the exact time-complexity of this algorithm's runtime only that it has at least the runtime of $O(|V|^3)$. This algorithm was designed with the expectation to outperform both the Kamada-Kawai algorithm and the Fruchterman-Reingold algorithm in terms of runtime. According to Hachul [18] the total runtime of the GEM algorithm is $O(|V|(|V|^2 + |E|))$.

This algorithm builds upon the Fruchterman-Reingold algorithm, by adding new heuristics to it. For example, they use local temperatures instead of global temperatures in order to better detect and deal with oscillations and rotations.

For this algorithm, the authors chose not to explicitly minimize edge-crossings. However, when they compared their algorithm to the Fruchterman-Reingold and Kamada-Kawai algorithms they note that the GEM algorithm can resolve cycles and folds in the graph easily, resulting in the graph having a low number of edge-crossings.

3.2.5 The FMS Algorithm

The FMS algorithm [21] has a runtime of $O(|V||E|)$ according to Hachul [18].

The algorithm is built around the Kamada-Kawai algorithm and greatly improves its speed, by using a multi-scale method. This is a method that divides the given graph into simpler nested sub-graphs and then draws the graph by iterating one level at the time. This algorithm "coarsens" the graph by shrinking nodes that are close to each other into one single node and thereby eliminating many local details, but preserves the graph's original structure.

Since this algorithm is built around the Kamada-Kawai algorithm, it focuses on the same aesthetic criterion of reflecting symmetry in the visualization. In the article, the FMS algorithm is also compared against the Kamada-Kawai algorithm and the authors show that the Kamada-Kawai algorithm produces more edge-crossings than the FMS algorithm does.

3.2.6 The GRIP Algorithm

For the GRIP algorithm [16] the time-complexity for the runtime is not specified, but according to Hachul [18] it has a runtime of $O(|V| \log(\text{diam}(G)^2))$, where $\text{diam}(G)$ denotes the diameter of the graph, which refers to the largest distance between any two nodes of the graph.

The GRIP algorithm uses a multi-scale method for drawing its graphs. This algorithm first filters the given graph's nodes into independent sets. Then the algorithm creates an initial placement for each set and finally refines the nodes positions.

In the article for the GRIP algorithm [16], the authors do not say anything about the aesthetics other than that the resulting graphs are aesthetically pleasing. According to Hachul [18] the drawings generated by the GRIP algorithm contain relatively many edge-crossings though.

3.2.7 The FM³ Algorithm

The FM³ algorithm [19] is able to produce large graphs with a runtime of $O(|V| \log|V| + |E|)$.

This algorithm also uses a multi-scale method. It first partitions all nodes into a “galaxy representation”. Each node is marked as either a sun, planet or moon. Then each “solar system” is collapsed into a single node. This process is repeated until a stopping condition is met. Each node in the smallest sub-graph is given a random initial placement. The other nodes are placed depending on the earlier placed nodes. Finally, the positions are refined by applying forces to the nodes and edges.

The FM³ algorithm [19] has relatively few edge-crossings and overlapping edges, as shown in the comparison by Hachul and Jünger [20].

3.3 Related Works

A comparison between algorithms was made by Brandenburg et al. [11]. They concluded that the Kamada Kawai algorithm [25] and the GEM algorithm [14] were the ones that performed the best in terms of runtime. The GEM algorithm was originally designed to outperform the Kamada Kawai algorithm, but Brandenburg et al. [11] use a modified version of the Kamada Kawai algorithm to speed up its computation time. These algorithms are only useful for graphs of a few hundred nodes though.

For large graphs of thousands of nodes, another comparison was made by Hachul and Jünger [20], in which they compare their own FM³ algorithm [19] to other algorithms for drawing large graphs. They conclude that of the tested force directed algorithms only the GRIP algorithm [16] is faster than their own FM³ algorithm. They also claim that in some cases the FM³ algorithm is faster than the GRIP algorithm, but that this could not be tested due to an executable error. They also compare the amount of edge-crossings that are in each algorithm’s resulting graphs. They conclude that the FM³ algorithm produces drawings with the fewest amount of edge-crossings, among the tested force-directed methods.

3.4 Comparison and Conclusion

When comparing the runtimes of the algorithms, the fastest ones are FM³, GRIP and FMS. According to the comparison by Hachul and Jünger [20] the FM³ algorithm should in some cases be faster than the GRIP algorithm. The following table (Figure 2) shows the different runtime complexities for each of the compared algorithms, as noted by the studied articles. The table also shows short summaries about each algorithm’s focused aesthetic criteria.

Algorithm	Time Complexity	Aesthetic Criteria
Fruchterman-Reingold (Regular)	$O(V ^2 + E)$	Distributes nodes evenly, makes edge-lengths uniform, reflects symmetry.
Fruchterman-Reingold (Grid-Variant)	$O(V + E)$	Same as regular, but slightly worse aesthetics.
Davidson-Harel	$O(V ^2 E)$	Distributes nodes evenly, makes edge-lengths uniform, minimizes edge-crossings, keeps a distance between nodes and edges.
Kamada-Kawai (Floyd-Warshall)	$O(V ^3)$	Reflects symmetry.
Kamada-Kawai (Johnson)	$O(V ^2 \log V + E V)$	Same as with Floyd-Warshall.
GEM	$O(V (V ^2 + E))$	Resolves cycles and folds easily.

FMS	$O(V E)$	Reflects symmetry.
GRIP	$O(V \log(\text{diam}(G)^2))$	Relatively many edge-crossings in practice.
FM ³	$O(V \log V + E)$	Minimizes edge-crossings, minimizes overlapping edges.

Figure 2. Table showing the different runtime complexities for each algorithm.

The comparison made by Hachul and Jünger [20] contains a test graph of a social network (small world network). They used the FM³, FMS and GRIP algorithms to draw this graph. The result showed that both FM³ and GRIP were significantly faster than the FMS algorithm. It also showed that both the FM³ algorithm and the GRIP algorithm display the characteristics of a small world network much better than the FMS algorithm does. This means that they display clusters of nodes, where highly connected nodes are placed closer to each other compared to the other nodes. Based on this, both the FM³ and GRIP algorithms seem most suitable for drawing small world networks.

When comparing how well each algorithm follows the aesthetic criteria it was difficult to say which one was the best since it often depends on the input graph, as seen in both comparisons by Hachul and Jünger [20] and Brandenburg et al. [11]. However, it can be noted that the FM³, FMS and GEM algorithms often produce the least amount of edge-crossings and according to the comparison made by Hachul and Jünger, the FMS algorithm produces relatively many overlapping edges and edge-crossings compared to the FM³ algorithm.

Therefore, the algorithm that seems most suitable for this degree project and will be implemented is the FM³ algorithm.

4. Method

In this chapter, the method that was used will be described. First the workspace that was used for the implementation was set up. Then the actual implementation was performed: parsing OWL files, implementing the algorithm and the touch functionality. Finally, the comparison between the algorithms was conducted.

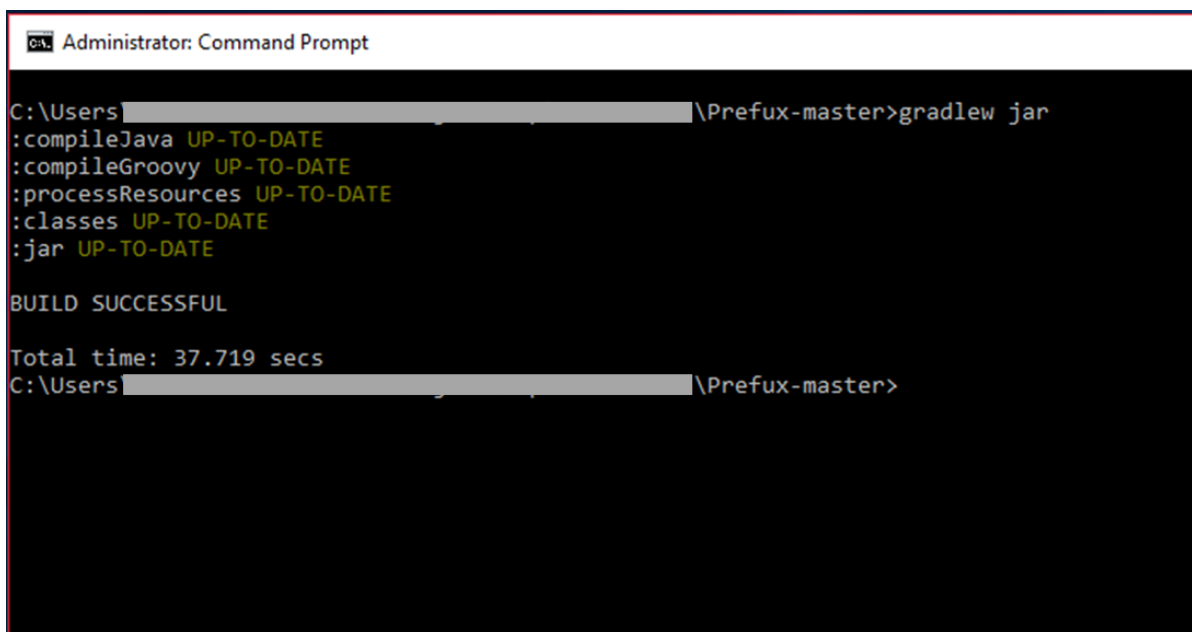
4.1 The Workspace

The implementation parts of this project were done using NetBeans IDE 8.0.2, which is a platform for developing applications using the Java programming language. The computer that was used during the project was running Windows 10.

First, the right version of Java Development Kit (JDK) was installed, Java 8 update 40 (64 bits). Then NetBeans IDE was installed with the path to the used JDK specified during the installation.

The JavaFX implementation of the Prefuse library, Prefux, was downloaded from its GitHub repository [4]. The path where the main Prefux components can be found is ".../src/main/java/prefux". Those are the earlier mentioned (Section 2.4) components that were modified and added to during the implementation.

After modifying any of the Prefux components, a new version of the Prefux library has to be created. In order to do this, Prefux uses the build tool Gradle. From the command prompt, change directory to the one containing the "gradlew.bat" file and then type "gradlew build", alternatively "gradlew jar" (Figure 3). This will build the library, which can then be found under "build/libs" as an executable jar file.



```
Administrator: Command Prompt
C:\Users\... \Prefux-master>gradlew jar
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar UP-TO-DATE

BUILD SUCCESSFUL

Total time: 37.719 secs
C:\Users\... \Prefux-master>
```

Figure 3. How to use Gradle from command prompt.

Next, a new JavaFX application was started in NetBeans IDE. This application is also the test-program that was used to test the implementation parts. Once the project template was created, it needed access to the Prefux library in order to use its components.

To add a new library in a NetBeans project, select “File” and then “Project Properties” for the specific project that is being worked on. Select “Libraries” from the “Categories” list in the newly opened window and “Add JAR/Folder” (Figure 4). Finally, navigate to and select the library (jar file) created earlier.

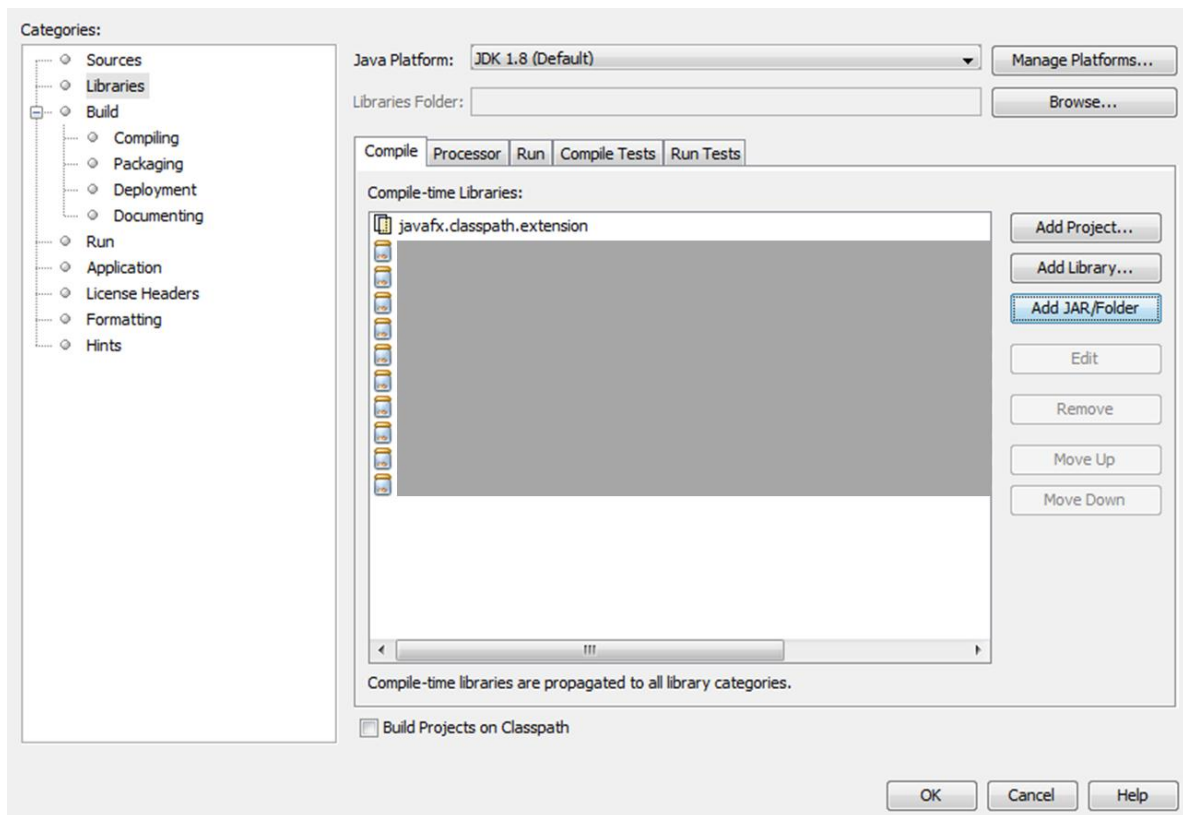


Figure 4. How to add a new library to a NetBeans project.

Once the workspace had been set up in this way, the design and implementation could begin. The code for the test-program can be found in Appendix A.

4.2 Parsing OWL Files

To make it possible for Prefux to read OWL files, a new java file was created for the IO component, located under “.../prefux/data/io”. This file was named “GraphOWLReader.java”.

A lab [1] has already been performed in an Eclipse project, to read and print the contents of OWL files by using a Java framework called Apache Jena. This framework provides an API for reading from and writing to RDF graphs. It also supports OWL. “GraphOWLReader.java” was implemented by using the Jena related libraries from this lab.

The libraries needed were the following:

- jena-2.6.2.jar
- slf4j-api-1.5.6.jar
- log4j-1.2.13.jar
- slf4j-log4j12-1.5.6.jar
- xercesImpl-2.7.1.jar
- iri-0.7.jar
- icu4j-3.4.4.jar

Before these libraries could be used in Prefux, the Gradle Wrapper needed to be modified to gain access to the libraries. First, the required libraries were added to Gradle’s “lib” folder (Figure 5).

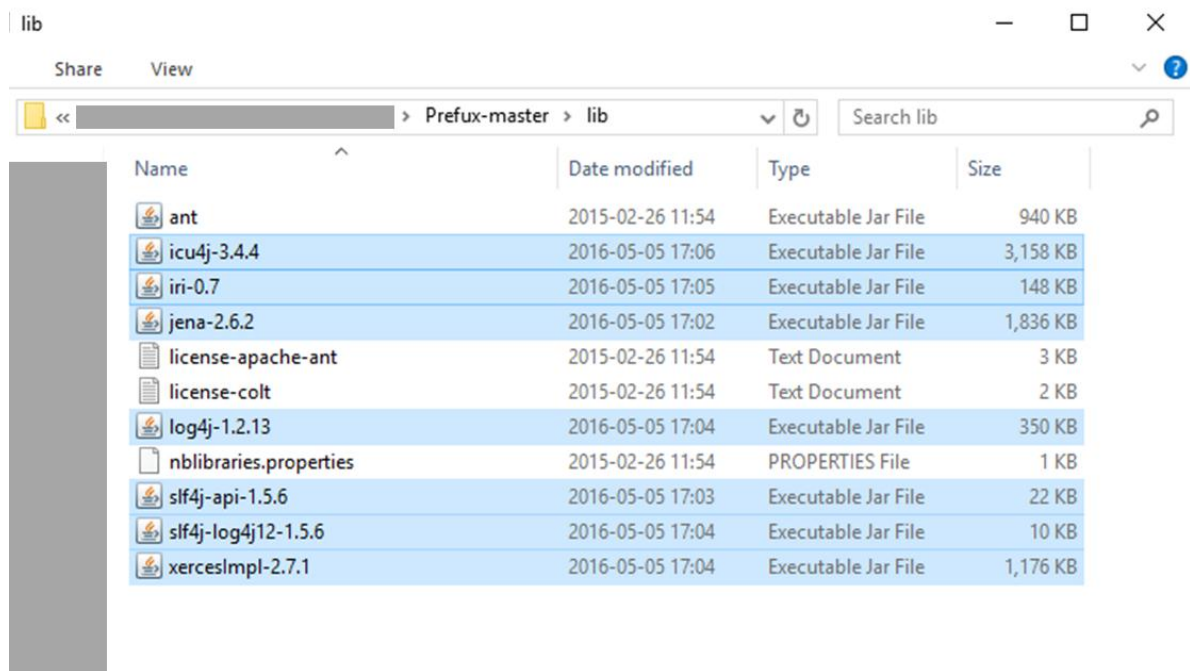


Figure 5. Location of Jena related libraries.

The Gradle file “build.gradle” was then edited, by adding the following lines under “dependencies” (Figure 6). This can be done using any suitable text editor, such as: GNU Emacs or WordPad.

- compile fileTree(dir: 'lib', include: 'jena-2.6.2.jar')
- compile fileTree(dir: 'lib', include: 'slf4j-api-1.5.6.jar')
- compile fileTree(dir: 'lib', include: 'slf4j-log4j12-1.5.6.jar')
- compile fileTree(dir: 'lib', include: 'icu4j-3.4.4.jar')
- compile fileTree(dir: 'lib', include: 'iri-0.7.jar')
- compile fileTree(dir: 'lib', include: 'log4j-1.2.13.jar')
- compile fileTree(dir: 'lib', include: 'xercesImpl-2.7.1.jar')

```

dependencies {
    compile "org.apache.lucene:lucene-core:4.10.1"
    compile 'org.apache.lucene:lucene-queryparser:4.10.1'
    compile 'org.apache.lucene:lucene-analyzers-
common:4.10.1'

    compile group: 'org.apache.logging.log4j', name: 'log4j-
api', version: '2.1'
    compile group: 'org.apache.logging.log4j', name: 'log4j-
core', version: '2.1'

    compile "org.codehaus.groovy:groovy-all:2.3.6"

    compile fileTree(dir: 'lib', include: 'jena-2.6.2.jar')
    compile fileTree(dir: 'lib', include: 'slf4j-
log4j12-1.5.6.jar')
    compile fileTree(dir: 'lib', include: 'slf4j-
api-1.5.6.jar')
    compile fileTree(dir: 'lib', include: 'log4j-1.2.13.jar')
    compile fileTree(dir: 'lib', include:
'xercesImpl-2.7.1.jar')
    compile fileTree(dir: 'lib', include: 'iri-0.7.jar')
    compile fileTree(dir: 'lib', include: 'icu4j-3.4.4.jar')

    /// compile "org.neo4j:neo4j:2.1.3"
    testCompile "org.spockframework:spock-core:0.7-
groovy-2.0"
}

```

Figure 6. The added dependencies to "build.gradle"

Then, the Prefix library needed to be rebuilt and the Jena related libraries needed to be added to the NetBeans project, according to the earlier method (Section 4.1).

The purpose of the GraphOWLReader is to read any given OWL ontology and create a Graph object that is compatible with the Prefix library. The first step in this process is to read the ontology into an OntModel, available from the earlier mentioned Jena libraries. This will create an ontology model that represents the given ontology.

Next, a new empty Prefix Graph is created and initialized. For each ontology class in the model, a new node is created for the Prefix Graph to represent that class. Information about the node (such as ID or name) is also added to the Prefix Graph.

All properties of each newly created node are iterated and for each property that is desired to be represented in the graph (such as the "subClassOf" property), information is gathered about that property for later use.

Object-Properties in the model that are not related to any specific class are also iterated and information is gathered from them as well.

When all information has been gathered about the properties, the edges of the Prefix Graph are created to represent each of the properties.

In order for the test-program to use this GraphOWLReader, it has to read the desired OWL file into a Prefix Graph according to Figure 7.

```

// A graph object set to null
Graph graph = null;
try
{
    graph = new GraphOWLReader().readOWL("/Anatomy/human.owl", true);
}

```

Figure 7. How to use the GraphOWLReader.

Once this process has been completed, the GraphOWLReader will return the Prefix Graph representing the ontology to be used by any of the other Prefix components.

The code for the GraphOWLReader can be found in Appendix B.

4.3 Implementation of Algorithm

When implementing the FM³ Algorithm, the file named “FM3Layout.java” was added to Prefux’s Layout component together with Prefux’s own force-directed algorithm, located under “...prefux/action/layout/graph”.

The file (FM3Layout.java) has the same structure and type of methods as Prefux’s own algorithm, called “ForceDirectedLayout.java”. This template contains the class “FM3Layout” that extends the already existing abstract base class “Layout” of Prefux. The class has a constructor for creating a new instance of the layout and the main “run” method, which starts whenever the visualization object it belongs to calls for it.

What happens when the run method is called, is that the algorithm uses its techniques to determine a placement for the nodes of the given graph. The FM3Layout’s run method was implemented by following the instructions in the authors’ article about the algorithm [19].

The main strategy for the FM³ algorithm is a so called “Divide and Conquer” strategy. It consists of three steps: Divide-Step, Multilevel-Step and Impera-Step.

First, the Divide-Step starts by taking the given graph and if the graph is disconnected (i.e. no path exists that connects all the nodes of the graph), it will divide the graph into multiple smaller connected subgraphs. Each subgraph is created by starting from one node, traversing all neighboring nodes and adding these nodes to the subgraph. This process is repeated until all nodes of the original graph have been traversed.

In the Multilevel-Step, the actual force calculation is performed on the nodes of each subgraph to create drawings of them. This step is further split-up into multiple sub-steps that will be explained later.

Once the force calculation is done, the Impera-Step will take the drawings of each subgraph and place them together in such a way that they won’t overlap each other. When performing this placement, the Impera-Step will: rotate the subgraphs, sort them into an order of decreasing size (amount of nodes) and pack them together to use as small total drawing area as possible.

The pseudo-code for the “Divide and Conquer” strategy is shown in Figure 8. The Multilevel-Step refers to the algorithm that draws each of the subgraphs (written as “A_{draw}” in the pseudo-code).

Function: Divide & Conquer Strategy

Input: a graph G and an aspect ratio r for the drawing area

Output: a drawing $D(G)$ of G with non-overlapping components and edges drawn as straight lines

Start

C denotes the set of components of G

Start (Divide-Step)

Divide G into an array of connected subgraphs $G_1, \dots, G_{|C|}$

End

For $i = 1: |C|$

$D(G_i) \leftarrow A_{draw}(G_i)$

End

Start (Impera-Step)

For $i = 1: |C|$

$D(G_i) \leftarrow Rotate(D(G_i))$

End

$\{D(G_1), \dots, D(G_{|C|})\} \leftarrow Sort(\{D(G_1), \dots, D(G_{|C|})\})$

$D(G) \leftarrow Pack(\{D(G_1), \dots, D(G_{|C|})\})$

End

End

Figure 8. The "Divide and Conquer" strategy of the implemented FM³ algorithm.

The Multilevel-Step consists of two phases: Coarsening phase and Refinement phase. These two phases will be run once for each of the subgraphs created in the Divide-Step.

In the Coarsening phase, the subgraph will be partitioned into a "galaxy" representation and collapsed to create a smaller version of the subgraph, called multilevel graph. Each node of the subgraph is assigned a role: sun, planet or moon. To select which nodes become suns, the algorithm randomly chooses one of the nodes that have the smallest star mass (a unique integer value for each node). In the original version of the subgraph all nodes are assigned the same star mass of 1. Once a sun has been selected, all its neighbors become planets belonging to the chosen sun. Additionally, the planets' neighbors become moons belonging to their respective planets. This selection process is repeated until all nodes of the subgraph have been assigned a role. Once the selection process is completed, the algorithm collapses each solar system to create a new multilevel graph. The star masses of this multilevel graph are simply obtained by adding together the star masses of each node in each solar system. This Coarsening phase will then repeat until a stopping criterion is met (i.e. when the smallest multilevel graph contains less than a determined amount of nodes).

The Refinement phase will begin with the smallest multilevel graph and generate a random initial placement for its nodes. Then the force model will be applied on that multilevel graph. The algorithm will then go through each multilevel graph step-by-step towards the original subgraph. For each multilevel graph, a placement will be generated based on the current position of the nodes in the former multilevel graph. Then the force model will be applied to the multilevel graph. This process is repeated until a drawing of the original subgraph has been generated.

Function: Multilevel Step

Input: a connected graph G

Output: a straight-line drawing $D(G)$ of G .

Start

Start (Coarsening-Phase)

$i \leftarrow 0$

$G_i \leftarrow G$

While not *Stopping_Criterion*(G_i)

 Create a galaxy partitioning of G_i using

Select_By_Star_Mass

 Create G_{i+1} by collapsing the solar systems of G

$i \leftarrow i + 1$

End

End

Start (Refinement-Phase)

$\{G_0, \dots, G_k\}$ denotes the set of multilevel graphs

Generate a random initial placement of the nodes of G_k

$D(G_k) \leftarrow \text{Force_Model}(G_k)$

$i \leftarrow k - 1$

While $i \geq 0$

 Generate an initial placement of the nodes of G_i

$D(G_i) \leftarrow \text{Force_Model}(G_i)$

$i \leftarrow i - 1$

End

End

End

Figure 9. The "Multilevel Step" of the implemented FM³ algorithm.

The algorithm that applies the force model is called Grid-Embedder. For each node in the multilevel graph, the Grid-Embedder will calculate the spring forces and the repulsion forces that act on the node and then move the node's position based on these forces.

Initially, the Grid-Embedder will calculate a maximum amount of iterations to run the algorithm, based on which multilevel graph it is currently working on. As the multilevel graph becomes larger, the maximum amount of iterations will decrease. This method is used to preserve the total runtime of the FM³ algorithm.

For every iteration, the Grid-Embedder will restrict the multilevel graph's nodes to a grid and boundary. This prevents the graph from growing unnecessarily large during the force application.

Function: Grid Embedder

Input: a graph G

Output: a straight-line drawing $D(G)$ of G

Start

Calculate Max_Iter

$i \leftarrow 1$

Do

Restrict the positions of the nodes of G

V denotes the set of nodes of G

Calculate spring forces F_{Spring} acting on the nodes of G

Calculate repulsion forces F_{Rep} acting on the nodes of G

Move the nodes of G based on F_{Spring} and F_{Rep}

$i \leftarrow i + 1$

While ($i \leq Max_Iter$)

Restrict the positions of the nodes of G

End

Figure 10. The "Grid-Embedder" of the implemented FM^3 algorithm.

Figure 11 shows the force model of the Grid-Embedder. $F_{Rep}(v)$ denotes the repulsion force acting on the node v because of node u . $F_{Spring}(v)$ denotes the spring force acting on the node v due to an edge between the nodes v and u . pos_v and pos_u denote the positions of the nodes. $l_{desired}(e)$ denotes the desired length of the edge.

$$F_{Spring}(v) = \begin{cases} -1 \cdot \log\left(\frac{\|pos_v - pos_u\|}{l_{desired}(e)}\right) \cdot \|pos_v - pos_u\| \cdot (pos_v - pos_u), & pos_v \neq pos_u \\ 0, & pos_v = pos_u \end{cases}$$
$$F_{Rep}(v) = \begin{cases} \frac{pos_v - pos_u}{\|pos_v - pos_u\|^2}, & pos_v \neq pos_u \\ 0, & pos_v = pos_u \end{cases}$$

Figure 11. Force model of the implemented FM^3 algorithm.

This formula for calculating the repulsion force is only one of two methods that the FM^3 algorithm uses. The actual repulsion force is calculated using a force-approximation algorithm shown in Figure 12.

Function: Force-Approximation

Input: a set of nodes V of a graph

Output: repulsive forces F_{Rep} for the nodes in V

```
Start
  If  $|V| \leq crossover\_point$ 
    For  $i = 0: |V|$ 
       $F_{Rep}(V_i) \leftarrow Direct\_Force\_Calculation$ 
    End
  Else
    Create a quad-tree  $T$  based on the nodes in  $V$ 
     $F_{Rep}(V_i) \leftarrow Multipole\_Framework(V, T)$ 
  End
End
```

Figure 12. Force-Approximation of repulsion forces for the implemented FM^3 algorithm.

If the size of the multilevel graph is smaller than a predetermined size value, the algorithm will apply the repulsion forces using the naïve method (i.e. the formula shown in Figure 11). If the multilevel graph is larger than the predetermined size value, the algorithm will create a so called Quad-Tree representation of the multilevel graph and apply the repulsion forces using the Multipole-Framework shown in Figure 13. The Multipole-Framework consists of four parts: Initialization, Bottom-Up Traversal, Top-Down Traversal and Obtain the Forces.

Each node in the Quad-Tree relates to the other nodes in different ways. This information is stored in five different node sets: I, D1, D2, D3 and K. The I-set stores for each Quad-Tree node the nodes that are well separated from it. The D1-set stores the nodes that are ill separated, neighbors and larger in boundary size. The D2-set stores the nodes that are ill separated, non-neighbors and larger in boundary size. The D3-set stores the nodes that are ill separated, neighbors and smaller in boundary size. The K-set stores the nodes that are ill separated, non-neighbors and smaller in boundary size. All of these node sets are initialized in the initialization part of the Multipole-Framework.

In the second part, “Bottom-Up Traversal”, a coefficient is calculated for each Quad-Tree node. This coefficient is called “multipole coefficient” and is calculated based on the multilevel graph nodes contained in the region represented by the Quad-Tree node. The third part, “Top-Down Traversal”, calculates a similar coefficient called “local coefficient”. These coefficients are then used for obtaining an approximation of the repulsion forces affecting the nodes of the multilevel graph. The multipole coefficient is used to calculate an approximate repulsion force based on the Quad-Tree nodes in the K-set. The local coefficient is used to calculate an approximate repulsion force based on the Quad-Tree nodes in the I-set and D2-set.

Thus, the repulsion forces from all non-neighboring nodes are approximated using the coefficients. The forces from the remaining nodes are calculated using the direct force model shown earlier (Figure 11).

Function: Multipole_Framework

Input: a set of nodes C and a quad-tree T with a set of quad-nodes V

Output: repulsive forces F_{Rep} for the nodes in C

Start

Start (Part 1: Initializations)

 If T contains only one node

 For $i = 0: |C|$

$F_{Rep}(C_i) \leftarrow Direct_Force_Calculation$

 End

 Exit

 For $i = 0: |V|$

$I(V_i) \leftarrow D_1(V_i) \leftarrow D_2(V_i) \leftarrow \emptyset$

 If V_i is a leaf node

$D_3(V_i) \leftarrow K(V_i) \leftarrow \emptyset$

 End

 End

End

Start (Part 2: Bottom-Up Traversal)

 For $i = 0: |V|$

 Calculate the coefficients of $M^p(V_i)$

 End

End

Start (Part 3: Top-Down Traversal)

$Calculate_Local_Expansions_and_Node_Sets(T, C)$

End

Start (Part 4: Obtain the Forces)

 Foreach leaf $v \in V$

 Foreach $c \in Sm(v)$

 Calculate $F_{local}(c)$ using $L^p(v)$

 Calculate $F_{direct}(c)$ using $D_1(v) \cup D_3(v)$

 Calculate $F_{multipole}(c)$ using $K(v)$

$F_{Rep} \leftarrow F_{local} + F_{direct} + F_{multipole}$

 End

 End

End

End

Figure 13. The "Multipole-Framework" of the implemented FM^3 algorithm.

In order for the test-program to use this algorithm, an instance of its layout has to be added to the visualization according to Figure 14.

```
ArrayList layout = new ArrayList(Activity.INFINITY, 30);
layout.add(new FM3Layout("graph"));
vis.putAction("layout", layout);
```

Figure 14. How to add an algorithm layout to a visualization.

The code for the FM³ algorithm can be found in appendix C.

4.4 Touch Functionality

The multi-touch table was connected to the computer as an alternative screen and sending data via an Ethernet connection.

When implementing the touch functionality, the desired controls were added to Prefux's "Controls" component, located under ".../prefux/controls".

In its original state, Prefux only had a "Drag Control", used for dragging the nodes of a visualized graph. This control did not react to touch events (gestures), so it was modified to perform the same actions for gestures as it would for regular mouse events on a computer.

To enable the controls to react to gestures, JavaFX provided a functionality for inputs that could be used to detect gestures performed on the display. Any events performed could then be compared to specific types of touch events to determine what actions to perform.

The Prefuse library contains several more controls than just a Drag Control. Two additional controls were added to Prefux: Focus Control and Zoom Control. Both of these are based on their corresponding controls from Prefuse.

Focus Control allows for selecting certain nodes of a visualized graph to be added to a focus group. Then any desired action can be performed on only the nodes in this focus group.

Zoom Control detects the gesture of moving two touch-points closer or further away, resulting in the display zooming out or in respectively.

To use these controls for a display object in the test-program, one has to add listeners to the display, according to Figure 15.

```
FxDisplay display = new FxDisplay(vis);

// Control Listeners
display.addControlListener(new DragControl());
display.addControlListener(new ZoomControl(display));
```

Figure 15. How to add control listeners to a display.

The code for the touch controls can be found in Appendix D.

4.5 Algorithm Comparison

For the final comparison, the FM³ Algorithm was compared to Prefux’s own force-directed algorithm, with regard to the previously defined criteria.

The following ontologies (Figure 16), obtained from the “Ontology Alignment Evaluation Initiative” [2], were tested to compare the algorithms:

Ontology	V	E
crs_dr.owl	14	25
PCS.owl	23	32
cmt.owl	29	72
MICRO.owl	31	33
linklings.owl	37	21
confOf.owl	38	42
MyReview.owl	38	68
paperdyne.owl	45	60
sigkdd.owl	49	54
Cocus.owl	54	67
confious.owl	56	68
Conference.owl	59	86
OpenConf.owl	62	65
ekaw.owl	73	84
edas.owl	103	105
iasted.owl	140	155
mouse.owl	2744	2856
human.owl	3304	3761
oaei2014_FMA_small_overlapping_nci.owl	3696	3693
oaei2014_NCI_small_overlapping_fma.owl	6488	4971
oaei2014_FMA_small_overlapping_snomed.owl	10157	10154
oaei2014_SNOMED_small_overlapping_fma.owl	13412	16287
oaei2014_NCI_small_overlapping_snomed.owl	23958	19013
oaei2014_SNOMED_small_overlapping_nci.owl	51128	31299
oaei2014_NCI_whole_ontology.owl	66724	59891
oaei2014_FMA_whole_ontology.owl	78988	78985

Figure 16. Table showing the ontologies and their corresponding amounts of nodes (|V|) and edges (|E|).

To compare the runtime of the algorithms, a simple “Timer” object was created. This timer was set to start counting the elapsed time at the beginning of each algorithm’s main “run” method and was set to stop at the end of the method (when the given graph had been completely finished).

When comparing how well each algorithm follows the aesthetic criterion of edge-crossings amount, a similar object was created. This “Edge-Crossings Counter” iterated through each edge of the completed graph and counted how many times two edges crossed each other.

In Appendix E, the codes for the Timer and Edge-Crossings Counter can be found.

The comparison experiments were performed on a computer running Windows 10, 64-bit system with 8 GB RAM. The CPU was an “Intel Core i5 4690K” (3.50 GHz). The version of Java used was Java 8 update 40 (64-bits).

When running the FM³ algorithm, there are five different parameters of interest: Desired Edge Length, Spring Stiffness Factor, Repulsion Factor, Displacement Factor and Force Threshold. Each of these parameters can be set upon instantiation of a new FM³ layout according to Figure 17.

```
layout.add(new FM3Layout ( "graph", 50.0, 0.01, 100.0, 0.5, 0.2 ));
```

Figure 17. How to instantiate a new FM³ layout and individually set all parameters.

The Desired Edge Length parameter controls the uniform edge length that the FM³ algorithm will try to maintain. When this value is relatively small, it results in a graph with short edges. The graph will have fewer edge-crossings and take up a smaller total drawing area, but the nodes will be put closer to their neighbor nodes. This could affect the graph's readability negatively, as it could be difficult to differentiate the nodes. If this value is instead relatively large, it results in a graph with long edges. In that case, the nodes become more separated and easier to differentiate. However, this would result in more edge-crossings and a larger total drawing area. For this comparison, the Desired Edge Length was set to 50, as this gives a decent balance between the two extremes.

Spring Stiffness Factor is the parameter that controls how "stiff" the springs are in the force model. A relatively small value will result in very elastic edges that can easily change lengths. This generally causes the graph's nodes to spread out more and edges to become longer. The amount of edge-crossings also increases. If this value is instead set relatively large, the edges become very stiff and difficult for the algorithm to move efficiently. This also results in more edge-crossings. Therefore, to receive the best possible amount of edge-crossings it is best to set this value on a middle ground to avoid the negative aesthetic effects that come with the two extremes. Thus, the value was set to 0.01 for this comparison.

Repulsion Factor is the parameter that controls how strong the repulsive forces are. If this value is set too small, it results in there being almost no repulsion force at all between the nodes. This can cause many nodes to overlap each other, which would negatively affect the readability of the graph. A large value will ensure that the nodes are well separated from each other, but it will instead increase the average edge length and the total drawing area. As mentioned about the earlier parameters, if the edge lengths increase, the amount of edge-crossings increases. In this comparison, this value was set to 100, as it gives a balanced result.

The Displacement Factor parameter controls how strongly both the spring forces and repulsion forces affect the placement of the nodes. If this value is set relatively small, the forces affecting the nodes will be very weak. This usually results in the forces having no effect and nodes having the same placement as the one that was initially given to it. This results in a large amount of edge-crossings. A large value will instead strengthen both spring forces and repulsion forces. This will also have a negative effect on the visualization as it results in a lot of oscillation. The algorithm will then spend many of its iterations to deal with the oscillation instead of finding a good placement for the nodes. The results from having it set to a large value is many edge-crossings and a large total drawing area. For this comparison, a good balance was found by setting this value to 0.5.

When the algorithm applies the forces to the nodes, it does this once for all nodes of the graph one iteration at the time. The amount of iterations depends on the Force Threshold parameter. This value can be seen as a minimum average force value that signalizes to the FM³ algorithm when the visualization has reached an equilibrium state (i.e. a state when the forces affecting the nodes are so small that the visualization can be considered completed). If this value is relatively small, it should result in better aesthetics, as the algorithm allows smaller forces to act on the graph before it finishes. However, this would result in longer runtimes. The opposite would be to set this value relatively large, which speed up the algorithm. However, at the cost of aesthetics. For this comparison, a balanced value was found to be 0.2.

The ontologies were tested several times to obtain average values for each graph. The results obtained from these evaluation methods were then put into tables and used to compare the algorithms.

All the tested graphs can be found in Appendix F.

5. Result

In this chapter the results obtained from each part of the project will be described.

5.1 The Workspace

From the set-up workspace in NetBeans, the test-program called "PrefixTest.java" was created.

Initially, the test-program will attempt to read an ontology into a Prefix graph object using the GraphOWLReader and then print out the size of that graph according to Figure 18.

```
Graph graph = null;
try
{
    graph = new GraphOWLReader().readOWL("../mouse.owl", true);

    System.out.println("Size: N "+graph.getNodeCount()+"", E "+graph.getEdgeCount());
```

Figure 18. How the test-program reads an ontology.

Next, the Prefix visualization object is created and the graph added to it according to Figure 19.

```
// Create a new visualization object
Visualization vis = new Visualization();

// Adding a group-name and a graph to the visualization object
vis.add(GROUP, graph);
```

Figure 19. How the test-program creates a new visualization.

The desired force-directed algorithm is then added to the visualization as a new Prefix action list according to Figure 20.

```
// Create an actionlist called layout
ActionList layout = new ActionList(1,30);

// Add the forcedirected layout to it and place it on the visualization
//layout.add(new ForceDirectedLayout("graph", false, true));
layout.add(new FM3Layout("graph"));
layout.add(new RepaintAction());
vis.putAction("layout", layout);
```

Figure 20. Test-Program adding a new force-directed layout to the visualization.

An instance of Prefix's FxDisplay is then created based on the visualization. To the display, the listeners for the two touch controls DragControl and ZoomControl are added as shown in Figure 21.

```
FxDisplay display = new FxDisplay(vis);

// Control Listeners
display.addControlListener(new DragControl());
display.addControlListener(new ZoomControl(display));
```

Figure 21. Test-Program creates an FxDisplay and adds control listeners to it.

For testing the focus control, a new Prefix coloring action is created. This action will color all nodes in the focus group of the visualization red. Then a listener for the FocusControl is added to the display. This is shown in Figure 22.

```
// -----FocusControl-----
int[] testPalette = new int[] {ColorLib.rgb(255,0,0)};
InGroupPredicate inFocusGroup
    = new InGroupPredicate( Visualization.FOCUS_ITEMS );
DataColorAction FocusColor
    = new DataColorAction(NODES, inFocusGroup, "id",
        Constants.NOMINAL, VisualItem.FILLCOLOR,
        testPalette);
nodeActions.add(FocusColor);
vis.putAction("nodes", nodeActions);
display.addControlListener(new FocusControl(1,"nodes"));
//-----End of FocusControl-----
```

Figure 22. How the Test-Program tests the FocusControl.

To add universal features to all nodes of the graph, the test-program does as shown in Figure 23. This will add a universal color to all nodes of the graph and determine the size of the nodes (set to 2.0).

```
// Node Features
ActionList nodeActions = new ActionList();
SizeAction size = new SizeAction(NODES, 2.0);
int[] class_color = new int[] {ColorLib.rgb(240,200,20)};
DataColorAction node_color
    = new DataColorAction( NODES, "id", Constants.NOMINAL,
        VisualItem.FILLCOLOR, class_color );
nodeActions.add(node_color);
nodeActions.add(size);
```

Figure 23. Test-Program adding features to all nodes of the graph.

Finally, the test-program runs all actions that were added to the visualization according to Figure 24.

```
vis.run("nodes");
vis.run("layout");
```

Figure 24. Test-Program running the actions added to the visualization.

The complete code for the test-program can be found in Appendix A.

5.2 Parsing OWL Files

To parse the OWL files, the GraphOWLReader was created and added to Prefix.

Figure 25 shows an example of a graph parsed from an OWL file, using the GraphOWLReader. The yellow nodes represent classes of the ontology, such as the classes: Article, Document, Author and Person. The edges between the nodes represent the nodes' properties. For example, the "subClassOf" property show that Article is a sub-class of Document and Author is a sub-class of Person. There is also the object-property of "has author" represented as an edge between Article and Author.

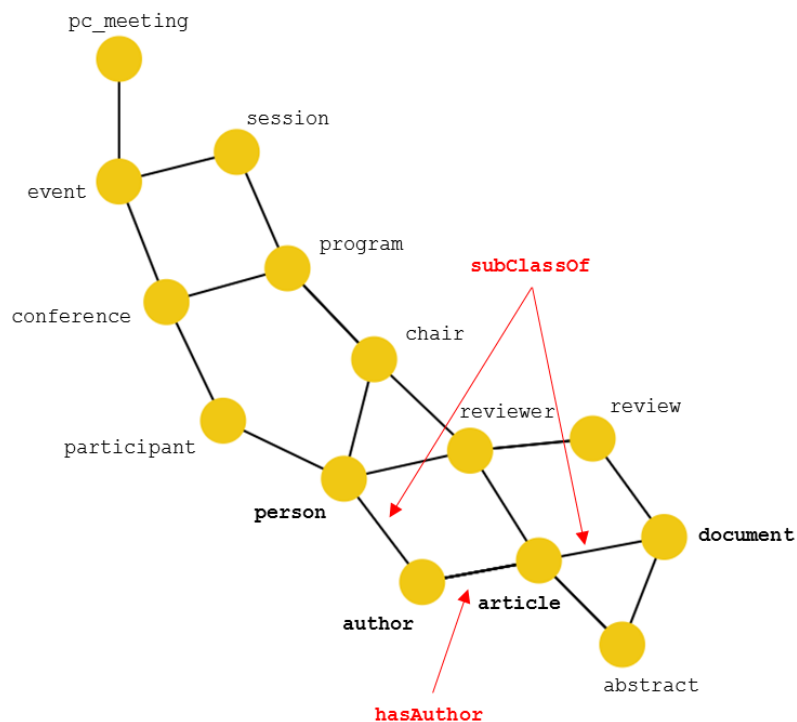


Figure 25. Visualization of the ontology "crs_dr.owl" with added labels.

5.3 Implementation of Algorithm

The FM³ Algorithm was implemented as "FM3Layout.java" and added to Prefix.

Figure 26 shows two examples of graphs drawn with the FM³ Algorithm.

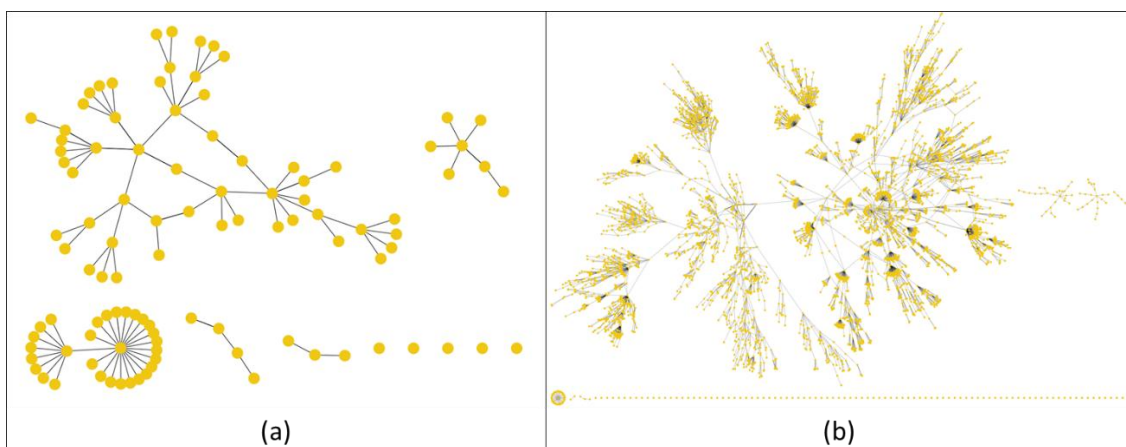


Figure 26. Visualizations of (a) "edas.owl" and (b) "NCI_small_overlapping_fma.owl" using the FM³ Algorithm.

5.4 Touch Functionality

This implementation resulted in three touch-enabled controls: DragControl, FocusControl and ZoomControl.

Figure 27 illustrates the DragControl, where one node of the visualized graph is dragged. This drag event is performed by touching the node on the display, moving it and then releasing the node.

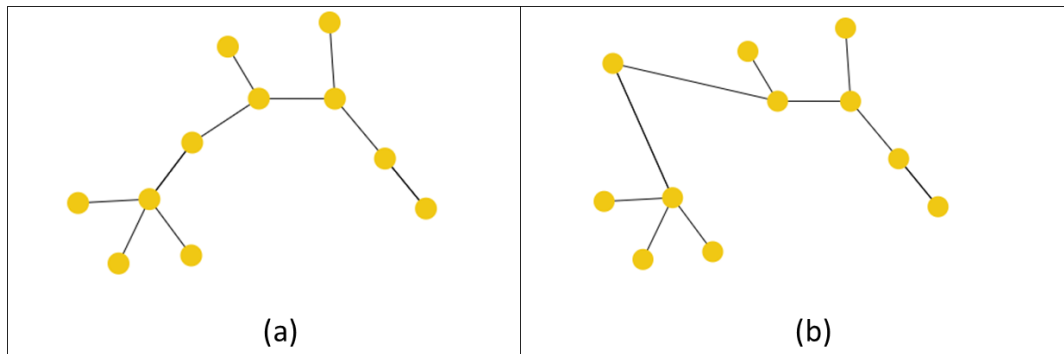


Figure 27. (a) Original visualization of graph, (b) the same graph after one of its nodes has been dragged.

The FocusControl is illustrated in Figure 28, changing the color to red of all the nodes added to the focus group. To focus a node, simply tap the node on the display. To focus more than one node, touch and hold the additional desired nodes.

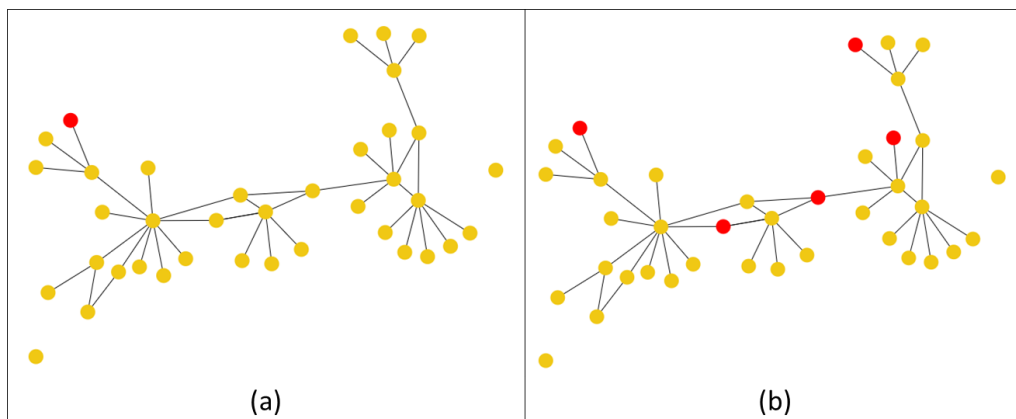


Figure 28. Focused nodes are colored red. (a) One node of a graph is focused, (b) five nodes being focused.

To use the ZoomControl, touch two points of the display and move the points further away or closer to each other. This will result in the display zooming in or out, illustrated in Figure 29.

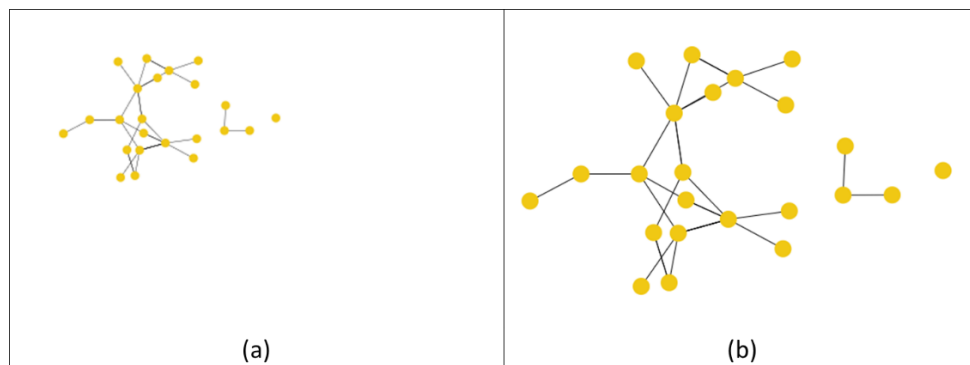


Figure 29. Illustration of Zoom Control. (a) Zoomed out view of graph, (b) zoomed in view of the same graph.

5.5 Algorithm Comparison

The results from comparing the algorithms' runtimes were obtained and placed into the table, shown in Figure 30. Each ontology was tested ten times for both algorithms, except for the eight largest ontologies (shown last in the table), which were tested two times for both algorithms. The table shows the average runtimes in seconds of each algorithm.

Ontology	FM ³ Algorithm	Prefux's Algorithm
crs_dr.owl	0.03	0.01
PCS.owl	0.04	0.02
cmt.owl	0.06	0.03
MICRO.owl	0.05	0.03
linklings.owl	0.02	0.04
confOf.owl	0.07	0.04
MyReview.owl	0.07	0.04
paperdyne.owl	0.07	0.04
sigkdd.owl	0.12	0.05
Cocus.owl	0.10	0.05
confious.owl	0.13	0.06
Conference.owl	0.12	0.06
OpenConf.owl	0.09	0.07
ekaw.owl	0.11	0.07
edas.owl	0.14	0.10
iasted.owl	0.67	0.14
mouse.owl	26.22	5.50
human.owl	41.78	7.60
oaei2014_FMA_small_overlapping_nci.owl	56.51	Stack-Overflow
oaei2014_NCI_small_overlapping_fma.owl	88.69	Stack-Overflow
oaei2014_FMA_small_overlapping_snomed.owl	409.98	Stack-Overflow
oaei2014_SNOMED_small_overlapping_fma.owl	435.50	Stack-Overflow
oaei2014_NCI_small_overlapping_snomed.owl	1912.00	Stack-Overflow
oaei2014_SNOMED_small_overlapping_nci.owl	13747.34	Stack-Overflow
oaei2014_NCI_whole_ontology.owl	42500.86	Stack-Overflow
oaei2014_FMA_whole_ontology.owl	63449.65	Stack-Overflow

Figure 30. Average runtimes in seconds, obtained for both algorithms.

When comparing the algorithms' amount of edge-crossings, the results were placed into the table in Figure 31. Each ontology was tested ten times for both algorithms, except for the eight largest ontologies (shown last in the table), which were tested two times for both algorithms. This table shows the average amount of edge-crossings obtained for each algorithm.

Ontology	FM³ Algorithm	Prefux's Algorithm
crs_dr.owl	1	7
PCS.owl	8	12
cmt.owl	69	104
MICRO.owl	0	0
linklings.owl	0	0
confOf.owl	2	2
MyReview.owl	47	37
paperdyne.owl	10	17
sigkdd.owl	4	5
Cocus.owl	21	9
confious.owl	20	13
Conference.owl	46	35
OpenConf.owl	4	5
ekaw.owl	6	9
edas.owl	7	5
iasted.owl	25	46
mouse.owl	1552	1931
human.owl	6627	7498
oaei2014_FMA_small_overlapping_nci.owl	1262	Stack-Overflow
oaei2014_NCI_small_overlapping_fma.owl	5005	Stack-Overflow
oaei2014_FMA_small_overlapping_snomed.owl	5349	Stack-Overflow
oaei2014_SNOMED_small_overlapping_fma.owl	311327	Stack-Overflow
oaei2014_NCI_small_overlapping_snomed.owl	55259	Stack-Overflow
oaei2014_SNOMED_small_overlapping_nci.owl	629427	Stack-Overflow
oaei2014_NCI_whole_ontology.owl	257180	Stack-Overflow
oaei2014_FMA_whole_ontology.owl	137568	Stack-Overflow

Figure 31. Average amount of edge-crossings, obtained for both algorithms.

Some of the graphs that were tested for Prefux's Algorithm, resulted in a Stack-Overflow error that prevented values from being obtained for that graph. This is shown in both tables as "Stack-Overflow".

6. Discussion

In this chapter, a discussion is performed about the project. This includes a section about the obtained results, where important comments are made about them. The next section discusses the method that was used and potential flaws that it might have. Then, a discussion is made about the references used throughout the theoretical parts of this thesis. Finally, a discussion is made about the project from other perspectives.

6.1 Result

The largest ontologies were unfortunately not possible to test with Prefux's Algorithm, due to a "Stack-Overflow" error being received.

From the results, the runtimes for the FM³ Algorithm are generally longer than those for Prefux's Algorithm. This is to be expected though, since the time-complexity for the FM³ Algorithm is $O(|V| \log|V| + |E|)$, whereas for Prefuse's Algorithm it is $O(|V| + |E|)$, according to the articles of each respective algorithm [19, 22]. The only exception to this is the ontology "linklings.owl".

The reason "linklings.owl" is drawn faster for the FM³ Algorithm is that the graph contains many small disconnected graphs. Because of the way the FM³ Algorithm is designed, this results in fewer iterations when calculating the forces acting between the nodes.

For most of the graphs, the FM³ Algorithm has about the same amount of edge-crossings as Prefux's Algorithm. However, sometimes the amount differs slightly in favor of either one of the algorithms, such as for: "cmt.owl", "Cocus.owl" or "Conference.owl".

It can be noted that as the ontologies grow larger, the FM³ Algorithm tends to draw graphs with significantly less edge-crossings than Prefux's Algorithm. This can be seen when comparing the ontologies "iasted.owl", "mouse.owl" and "human.owl" for both algorithms. However, there is not enough evidence to say that this is always true, it could just be that it is easier for the FM³ Algorithm to draw those types of graphs.

When determining which of these algorithms is most satisfactory, with regard to the earlier defined criteria (Section 3.1), both algorithms have their own strengths and weaknesses. The FM³ Algorithm is able to draw larger graphs and with fewer edge-crossings. However, Prefux's Algorithm has a significantly faster runtime.

6.2 Method

The implementation of the project was done in NetBeans IDE, although any suitable platform for Java applications could just as easily have been used, such as Eclipse.

Each part of this project was performed using the software engineering method of the waterfall model, meaning that each part of the project was done sequentially as phases. Initially, the requirements of the project were determined. Then, a study on the subject helped to determine how the design and implementation should be done. After each implementation, the code was tested using Java's "System Class" to print out debugging information to NetBeans's output window. If a bug was detected in the code, the project would: take one step back, work on the implementation part again and then test the code again. The reason for using the waterfall model was because it is easy to understand its step-by-step structure, from planning to design to implementation to testing. Because it was easy to understand, it resulted in much documentation, which was very useful when writing this report. However, a problem with the waterfall model is that if an issue is discovered about an earlier step later in the process, it could be costly in terms of time as the subsequent steps would then have to be done again.

When parsing the OWL files, only the minimal amount of information was taken into account to create the graphs. The reason for this was to save time, both for the implementation and for the comparisons. If more information had been represented in the graphs, the graphs would have contained more nodes and edges, hence it would have taken more time to run the algorithms during the comparison parts of the project.

The FM³ Algorithm was implemented by following the authors' article [19]. The problem with this method, is that misunderstandings about specific parts of the algorithm can arise. For instance, some parts of the algorithm are not well detailed in the article. An example of this is the "select by star mass" method, which is used to partition the graph's nodes to a galaxy representation (mentioned in Section 4.3). This method is not well explained in the article and only has about one paragraph describing it. Because of this, it can lead to confusion about exactly how the method should be implemented. This can have dire consequences for the results, as the implemented algorithm might have differences to that of the original. For example, it might take longer runtimes or show worse aesthetic characteristics than the original would.

Another problem with this method of implementing the FM³ Algorithm is that the concepts used by the authors are different from those used in the Prefuse library. For example, a "graph" object could be defined differently in the article compared to how it is defined in Prefuse. Therefore, the operations that are performed on the graphs by the authors might not be as easily performed in Prefuse.

If one would do the project again, using the same comparison method between the algorithms, it is most likely not going to produce the exact same result. When visualizing the graphs, there are certain factors that can produce faster or slower runtimes, as well as produce more or less edge-crossings. For example, many parts of the FM³ algorithm are based on random values. The initial placement of the graph's nodes is random and the "galaxy partitioning" process selects the initial nodes at random. Depending on these random values the nodes can be placed on coordinates that makes it easier or more difficult for the algorithm to "untangle" the graph, which can result in different runtimes and different edge-crossings amounts. Hence the reason why the graphs were tested several times for both algorithms, to produce average values for the comparison.

The results from the runtime comparison were obtained by testing how much time it took for the algorithms' run method to completely finish. The used timer started counting at the beginning of the method and stopped at the end of the method. These results can be considered valid, since it is the run method that performs the actual force-calculation and placement of the nodes. Therefore, the time this method took from start to finish is the exact time that it took for the algorithm to visualize the graph.

To compare the amount of edge-crossings in a visualized graph, the edge-crossings counter iterated through every edge of the graph and checked that its coordinates were not placed over the same coordinates as another edge. These results could unfortunately be wrong in some circumstances, due to the precisions of the coordinates. If two edges do not cross but are very close to each other, the edge-crossings counter could interpret it as an edge-crossing. The results received from this counter should therefore be considered as an approximation and not as an exact value.

6.3 References

In the beginning of this project several relevant sources were given by the advisor. In this thesis, these are the website references and the article by Landesberger et al. [40]. These were highly relevant to the implementation parts of the project and offered an initial insight into the subject.

To make sure the references used in this thesis were as reliable as possible, they were obtained using the university's referenced databases for article searches. These were primarily Google Scholar and Springer Link. References from Springer Link can be considered reliable, since their database is used specifically for publishing scientific articles.

While Google Scholar provides articles from academic journals, books and conference papers etc. it can sometimes provide many articles that are irrelevant to the subject or doesn't come from well-known sources. This can affect the reliability of articles obtained from Google Scholar as they might have very little relevance to the project. However, most of the references obtained in this manner have been cited by at least one of the more reliable sources. Another method for determining the reliability of an article was to check how many times the article had been cited by others.

Many of these articles have also been performed at other universities. Such as the algorithm articles by Hachul and Jünger [19, 20], which were performed at the university of Köln in Germany.

6.4 Other Perspectives

As a whole, this project provides the possibility of visualizing graphs based on ontologies of the OWL format. It provides an additional algorithm capable of visualizing large graphs containing thousands of nodes. Also, it provides touch functionality so that the graphs can be interacted with on a multi-touch table. The comparison parts of this project provides some information about what can be expected from both the implemented FM³ Algorithm and Prefux's own force-directed algorithm.

Visualization of graphs can further help other scientific fields, for example: by showing a visual representation of a database and thus make its content easier to understand.

7. Conclusions

The motivation for this project was to be able to visualize large ontologies on the multi-touch table with regards to time and aesthetic criteria. Ontologies provide information about specific domains and help with understanding the concepts within those domains. The reason why visualization of ontologies is important is because they are often huge and can be difficult to read. Therefore, it can greatly improve the readability and understandability of the ontology if they are visualized.

This project has studied articles about existing graph drawing algorithms and the problems of graph visualization. Using this information, the aesthetic criterion of edge-crossings amount was determined to be the most important. The existing algorithms have been compared with regards to runtime complexity and amount of edge-crossings. The comparison concluded that the FM³ algorithm satisfied the comparison criteria best of the studied algorithms. Therefore, the first of the two defined goals can be considered completed.

For the second goal, this project has implemented the FM³ algorithm into the Prefux library, using the Java programming language, in a similar way as Prefux's own algorithm has been implemented. Therefore, this goal can also be considered completed.

Additional requirements of this project were to be able to read ontologies of the OWL format and to be able to interact with the visualized graph on the multi-touch table. This project has also fulfilled these requirements as it provides methods for reading OWL files into a Prefux graph and interacting with the graph using JavaFX's touch functionality.

A comparison between the implemented FM³ algorithm and Prefux's own force-directed algorithm has also been provided. This comparison concluded that the FM³ algorithm has fewer edge-crossings than Prefux's algorithm, but that Prefux's algorithm is significantly faster in terms of runtimes.

Overall, this report has provided information about ontologies, the Prefuse library and graph drawing algorithms etc. that can further help research into scientific fields.

For future work, additional parts can be added to Prefux. Compared to the original Prefuse library, Prefux lacks some parts that have not yet been implemented. Such as: additional touch controls or ability to read other types of data (other than GraphML or OWL).

An example of a work is to use this report and its referenced articles to try to create a completely new graph drawing algorithm with the intent of outperforming both Prefux's algorithm and the FM³ algorithm. The referenced articles provide many different methods for graph visualization and by combining several of these methods one could probably create an algorithm that can visualize graphs very fast and with good enough aesthetics. This new algorithm could then be implemented in Prefux by using the same method as the one provided in this report.

References

- [1] "Lab exercise: Ontology Tools," [Online]. Available: https://www.ida.liu.se/~TDDD43/themes/ontology_tools_lab_140916.pdf. [Accessed 13 September 2016].
- [2] "Ontology Alignment Evaluation Initiative," [Online]. Available: <http://oaei.ontologymatching.org/2015/>. [Accessed 13 September 2016].
- [3] "OWL," [Online]. Available: <https://www.w3.org/OWL/>. [Accessed 01 November 2016].
- [4] "Prefix," [Online]. Available: <https://github.com/effrafax/Prefix>. [Accessed 13 September 2016].
- [5] "RDF," [Online]. Available: <https://www.w3.org/RDF/>. [Accessed 01 November 2016].
- [6] "The prefuse visualization toolkit," [Online]. Available: <http://prefuse.org/>. [Accessed 13 September 2016].
- [7] "XML," [Online]. Available: <https://www.w3.org/XML/>. [Accessed 01 November 2016].
- [8] D. Auber, Y. Chiricota, F. Jourdan and G. Melançon, "Multiscale visualization of small world networks," in *Information Visualization, IEEE Symposium on*, IEEE Computer Society, 2003, October, pp. 10-10.
- [9] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force-calculation algorithm," in *Nature*, 1986, pp. 446-449.
- [10] F. Beck, M. Burch and S. Diehl, "Towards an Aesthetic Dimensions Framework for Dynamic Graph Visualisations," in *Information Visualisation, 2009 13th International Conference*, IEEE, 2009, pp. 592-597.
- [11] F. J. Brandenburg, M. Himsolt and C. Rohrer, "An experimental comparison of force-directed and randomized graph drawing algorithms," in *Graph Drawing*, Springer Berlin Heidelberg, 1996, January, pp. 76-87.
- [12] R. Davidson and D. Harel, "Drawing graphs nicely using simulated annealing," *ACM Transactions on Graphics (TOG)*, vol. 15, no. 4, pp. 301-331, 1996.
- [13] Z. Dragisic, V. Ivanova, P. Lambrix, D. Faria, E. Jimenez-Ruiz and C. Pesquita, "User validation in ontology alignment," in *International Semantic Web Conference*, 2016, pp. 200-217.
- [14] A. Frick, A. Ludwig and H. Mehldau, "A fast adaptive layout algorithm for undirected graphs (extended abstract and system demonstration)," in *Graph Drawing*, Springer Berlin Heidelberg, 1995, January, pp. 388-403.
- [15] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and experience*, vol. 21, no. 11, pp. 1129-1164, 1991.

- [16] P. Gajer and S. G. Kobourov, "GRIP: Graph dRrawing with intelligent placement," in *Graph Drawing*, Springer Berlin Heidelberg, 2001, January, pp. 222-228.
- [17] T. R. Gruber, "A translation approach to portable ontology specifications," *Knowledge acquisition*, vol. 5, no. 2, pp. 199-220, 1993.
- [18] S. Hachul, "A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs," Ph.D. dissertation, Universität zu Köln, 2005.
- [19] S. Hachul and M. Jünger, "Drawing large graphs with a potential-field-based multilevel algorithm," in *Graph Drawing*, Springer Berlin Heidelberg, 2005, January, pp. 285-295.
- [20] S. Hachul and M. Jünger, "Large-Graph Layout Algorithms at Work: An Experimental Study," *J. Graph Algorithms Appl.*, vol. 11, no. 2, pp. 345-369, 2007.
- [21] D. Harel and Y. Koren, "A fast multi-scale method for drawing large graphs," in *Graph drawing*, Springer Berlin Heidelberg, 2001, pp. 183-196.
- [22] J. Heer, S. K. Card and J. A. Landay, "Prefuse: a toolkit for interactive information visualization," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 2005, April, pp. 421-430.
- [23] V. Ivanova and P. Lambrix, "A unified approach for aligning taxonomies and debugging taxonomies and their alignments," in *Extended Semantic Web Conference*, 2013, pp. 1-15.
- [24] V. Ivanova, P. Lambrix and J. Åberg, "Requirements for and Evaluation of User Support for Large-Scale Ontology Alignment," in *Extended Semantic Web Conference*, 2015, pp. 3-20.
- [25] T. Kamada and S. Kawai, "An algorithm for drawing general undirected graphs," *Information processing letters*, vol. 31, no. 1, pp. 7-15, 1989.
- [26] S. Kirkpatrick, C. Gelatt, Jr. and M. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, pp. 671-680, 1983.
- [27] S. G. Kobourov, "Force-directed drawing algorithms," in *Handbook of graph drawing and visualization*, CRC press, 2013.
- [28] P. Lambrix and R. Kaliyaperumal, "A Session-based Ontology Alignment Approach enabling User Involvement," *Semantic Web Journal*, 2016.
- [29] P. Lambrix and V. Ivanova, "A unified approach for debugging is-a structure and mappings in networked taxonomies," *Journal of Biomedical Semantics*, vol. 4, 2013.
- [30] P. Lambrix, F. Wei-Kleiner and Z. Dragisic, "Completing the is-a structure in light-weight ontologies," *Journal of Biomedical Semantics*, vol. 6, 2015.
- [31] P. Lambrix and H. Tan, "SAMBO - A System for Aligning and Merging Biomedical Ontologies," *Journal of Web Semantics*, vol. 4, no. 3, pp. 196-206, 2006.

- [32] P. Lambrix, Z. Dragisic, V. Ivanova and C. Anslow, "Visualization for Ontology Evolution," *International Workshop on Visualization and Interaction for Ontologies and Linked Data*, pp. 54-67, 2016.
- [33] N. Metropolis, A. Rosenbluth, R. M.N., A. Teller and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087-1092, 1953.
- [34] M. Pohl, M. Schmitt and S. Diehl, "Comparing the Readability of Graph Layouts using Eyetracking and Task-oriented Analysis," in *Computational Aesthetics*, 2009, May, pp. 49-56.
- [35] H. C. Purchase, "Effective information visualisation: a study of graph drawing aesthetics and algorithms," *Interacting with computers*, vol. 13, no. 2, pp. 147-162, 2000.
- [36] P. Shvaiko and J. Euzenat, "Ontology matching: State of the art and future challenges," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 1, pp. 158-176, 2013.
- [37] R. Studer, V. R. Benjamins and D. Fensel, "Knowledge engineering: principles and methods," *Data & knowledge engineering*, vol. 25, no. 1, pp. 161-197, 1998.
- [38] C. Ware, H. Purchase, L. Colpoys and M. McGill, "Cognitive measurements of graph aesthetics," *Information Visualization*, vol. 1, no. 2, pp. 103-110, 2002.
- [39] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440-442, 1998.
- [40] T. Von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J. D. Fekete and D. W. Fellner, "Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges," in *Computer graphics forum*, vol. 30, Blackwell Publishing Ltd, 2011, September, pp. 1719-1749.

Appendix A. Test-Program (PrefuxTest.java)

```
package prefuxtestproject;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.event.Event;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;
import prefux.Constants;
import prefux.FxDisplay;
import prefux.Visualization;
import prefux.action.ActionList;
import prefux.action.RepaintAction;
import prefux.action.assignment.DataColorAction;
import prefux.action.assignment.ShapeAction;
import prefux.action.assignment.SizeAction;
import prefux.action.layout.graph.FM3Layout;
import prefux.action.layout.graph.ForceDirectedLayout;
import prefux.activity.Activity;
import prefux.controls.DragControl;
import prefux.controls.FocusControl;
import prefux.controls.ZoomControl;
import prefux.data.Graph;
import prefux.data.io.DataIOException;
import prefux.data.io.GraphMLReader;
import prefux.data.io.GraphOWLReader;
import prefux.render.DefaultRendererFactory;
import prefux.render.LabelRenderer;
import prefux.util.ColorLib;
import prefux.util.PrefuseLib;
import prefux.visual.VisualItem;
import prefux.visual.expression.InGroupPredicate;

/**
 *
 * @author Daniel Sund
 */
public class PrefuxTest extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }

    private static final double WIDTH = 800; //300;
    private static final double HEIGHT = 600; //250;
    private static final String GROUP = "graph";

    @Override
    public void start(Stage primaryStage)
    {
        // The Primary stage (program)
        primaryStage.setTitle("Prefux Test");

        // Window borders (create a new window)
        BorderPane root = new BorderPane();

        // Set primary stage (program) to have a new scene in the window
        primaryStage.setScene(new Scene(root, WIDTH, HEIGHT));

        // the window is made part of a class called "display"
        root.getStyleClass().add("display");

        // Show the window on the user interface
        primaryStage.show();
    }
}
```

```

// A graph object set to null
Graph graph = null;
try
{
    graph = new GraphOWLReader().readOWL("../linklings.owl", true);

    System.out.println("Size: N "+graph.getNodeCount()+" , E
"+graph.getEdgeCount());

    // Create a new visualization object
    Visualization vis = new Visualization();

    // Adding a group-name and a graph to the visualization object
    vis.add(GROUP, graph);

    // create a new default renderer factory
    DefaultRendererFactory drf = new DefaultRendererFactory();
    vis.setRendererFactory(drf);

    // Create an actionlist called layout
    ActionList layout = new ActionList(1,30);

    // Add the forcedirected layout to it and place it on the visualization
    //layout.add(new ForceDirectedLayout("graph", false, true));
    layout.add(new FM3Layout("graph"));
    layout.add(new RepaintAction());
    vis.putAction("layout", layout);

    final String NODES = PrefuseLib.getGroupName(GROUP, Graph.NODES); //
"NODES" = String for the group containing all nodes

    FxDisplay display = new FxDisplay(vis);

    // Control Listeners
    display.addControlListener(new DragControl());
    display.addControlListener(new ZoomControl(display));

    // Handler for empty space events
    root.addEventHandler(Event.ANY, display);

    // Node Features
    ActionList nodeActions = new ActionList(); // List of actions performed
on nodes
    SizeAction size = new SizeAction(NODES, 2.0); // Set size for nodes
    int[] class_color = new int[] {ColorLib.rgb(240,200,20)}; // Palette
with gold color
    DataColorAction node_color = new DataColorAction( NODES, "id",
Constants.NOMINAL, VisualItem.FILLCOLOR, class_color );
    nodeActions.add(node_color);
    nodeActions.add(size);

    // -----FocusControl-----
    int[] testPalette = new int[] {ColorLib.rgb(255,0,0)}; // Palette with
red color
    InGroupPredicate inFocusGroup = new InGroupPredicate(
Visualization.FOCUS_ITEMS ); // Predicate, only applies action to nodes in
FocusGroup
    // New data-color action (Nodes in focus will be colored red)
    DataColorAction FocusColor = new DataColorAction(NODES, inFocusGroup,
"id", Constants.NOMINAL, VisualItem.FILLCOLOR, testPalette);
    nodeActions.add(FocusColor); // Add the color-change action to the list
of actions.
    vis.putAction("nodes", nodeActions);
    display.addControlListener(new FocusControl(1,"nodes")); // add a
control listener for FocusControl
    //-----End of FocusControl-----

    root.setCenter(display);

```

```
        vis.run("nodes");
        vis.run("layout");
    }
    catch (DataIOException e)
    {
        e.printStackTrace();
        System.err.println("Error loading graph. Exiting...");
        System.exit(1);
    }
}
}
```

Appendix B. GraphOWLReader

```
/*
 * Author: Daniel Sund
 * Read .owl file into a Graph
 */
package prefix.data.io;

import java.util.HashMap;
import java.util.ArrayList;
import java.util.Map;
import java.util.Iterator;

// prefix
import prefix.data.Graph;
import prefix.data.Node;
import prefix.data.Edge;

// Jena
import com.hp.hpl.jena.ontology.OntModel;
import com.hp.hpl.jena.ontology.OntModelSpec;
import com.hp.hpl.jena.ontology.OntClass;
import com.hp.hpl.jena.ontology.ObjectProperty;
import com.hp.hpl.jena.ontology.OntResource;
import com.hp.hpl.jena.rdf.model.ModelFactory;
import com.hp.hpl.jena.rdf.model.Statement;
import com.hp.hpl.jena.rdf.model.StmtIterator;

/*
   This class should read an .owl file into a Jena OntModel
   and then convert that OntModel into a Prefux Graph.
*/
public class GraphOWLReader
{
    // Jena OntModel to be converted
    private final OntModel jena_ontology_model =
ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM, null);

    // Prefux Graph to be returned
    private Graph prefix_graph = null;

    // Array of the prefix graph's nodes
    private final Map<String,Node> graph_nodes = new HashMap<>();

    // Array of the prefix graph's edges
    private final ArrayList<String[]> graph_edges = new ArrayList<>();

    public Graph get_graph(){ return prefix_graph; }

    public OntModel get_model(){ return jena_ontology_model; }

    // Read OWL-files and create a prefix graph
    public Graph readOWL( String owl_path, boolean directed ) throws
DataIOException
    {
        try
        {
            // 1. Read .owl file to the OntModel
            jena_ontology_model.read("file:///"+owl_path);

            // 2. Create the Graph from the OntModel
            create_graph(directed);

            return prefix_graph;
        }
        catch ( Exception e )
        {
            if ( e instanceof DataIOException )

```

```

        throw (DataIOException)e;
    else
        throw new DataIOException(e);
    }
}

// Create the graph
private void create_graph( boolean directed )
{
    // Initialize the graph
    initialize_graph( directed );

    // Create the class nodes
    Iterator<OntClass> cls_it = jena_ontology_model.listNamedClasses();
    while( cls_it.hasNext() )
    {
        OntClass cls = cls_it.next();

        // Create this node
        Node class_node = prefix_graph.addNode();
        class_node.setString( "id", cls.getURI() );
        class_node.setString( "name", cls.getLocalName() );
        class_node.setString( "type", "Class" );
        graph_nodes.put( cls.getURI(), class_node );

        // Get data from this class, for creating edges later
        get_edge_data( cls );
    }

    // Create the ObjectProperty edges
    Iterator<ObjectProperty> o_prop_it =
jena_ontology_model.listObjectProperties();
    while( o_prop_it.hasNext() )
    {
        ObjectProperty o_prop = o_prop_it.next();

        if( o_prop.getURI() != null && o_prop.getDomain() != null &&
o_prop.getRange() != null
            && o_prop.getDomain().getURI() != null &&
o_prop.getRange().getURI() != null )
        {
            String source_id = o_prop.getDomain().getURI();
            String name = o_prop.getLocalName();
            String target_id = o_prop.getRange().getURI();

            // Edge
            String[] edge_data = new String[3];
            edge_data[0] = source_id;
            edge_data[1] = target_id;
            edge_data[2] = name;
            graph_edges.add(edge_data);
        }
    }

    create_edges();
}

// Get data necessary for the creation of edges
private void get_edge_data( OntResource res )
{
    if( res.canAs( OntClass.class ) ) // Handle OntClass
    {
        // One edge for every sub-class relation
        String source_id = res.getURI();
        String name;
        StmtIterator stmt_it = res.listProperties();
        while( stmt_it.hasNext() )
        {

```

```

Statement property = stmt_it.next();
name = property.getPredicate().getLocalName();

if( name.equals("subClassOf") && property.getResource().getURI() !=
null )
{
    String target_id = property.getResource().getURI();
    String[] edge_data = new String[3];
    edge_data[0] = source_id;
    edge_data[1] = target_id;
    edge_data[2] = name;
    graph_edges.add(edge_data);
}
}
}

// Create the edges of the graph
private void create_edges()
{
    for( String[] edge_data : graph_edges )
    {
        Node src = graph_nodes.get(edge_data[0]);
        Node trg = graph_nodes.get(edge_data[1]);

        if( src != null && trg != null )
        {
            Edge edge = prefix_graph.addEdge(src, trg);
            edge.setString("name", edge_data[2]);
            edge.setString("type", "Property");
        }
    }
}

// Initialize the graph
private void initialize_graph( boolean directed )
{
    prefix_graph = new Graph( directed );
    prefix_graph.addColumn( "id", String.class ); // URI for nodes and edges
    prefix_graph.addColumn( "src_id", String.class ); // Source URI for edges
(null for node entries)
    prefix_graph.addColumn( "trg_id", String.class ); // Target URI for edges
(null for node entries)
    prefix_graph.addColumn( "name", String.class ); // Local Name of Class or
Property
    prefix_graph.addColumn( "type", String.class ); // Type: "Class" or
"Property"
}
}

```

Appendix C. FM³ Algorithm

```
package prefix.action.layout.graph;

import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Random;

import prefix.action.layout.Layout;
import prefix.data.Graph;
import prefix.util.PrefuseLib;
import prefix.data.Node;
import prefix.data.Edge;
import prefix.data.util.Point2D;
import prefix.data.util.Rectangle2D;
import prefix.visual.VisualItem;

public class FM3Layout extends Layout
{
    protected String          m_nodeGroup;      // Name of the node group
    protected String          m_edgeGroup;      // Name of the edge group
    protected double          uniform_edge_length;
    protected boolean         first_run = true;
    protected double          aspect_ratio = 1.5; // Defined as
(width/height)
    private FM3_Graph         fm3_graph; // original version of the graph
    protected boolean         run_restrict_to_grid = true;
    private Counter           elapsed_time_counter;
    private Edge_Cross_Counter edge_cross_counter;

    protected double          spring_stiffness_factor;
    protected double          repulsion_factor;
    protected double          displacement_factor;
    protected double          force_threshold;

    /* Constructor for creating a new FM3 Layout.
    */
    public FM3Layout(String group)
    {
        this( group, 50.0, 0.01, 100.0, 0.5, 0.2 );
    }

    public FM3Layout(String group, double desired_edge_length, double
spring_stiffness, double repulsion, double displacement, double force_thresh )
    {
        super(group);
        m_nodeGroup = PrefuseLib.getGroupName(group, Graph.NODES);
        m_edgeGroup = PrefuseLib.getGroupName(group, Graph.EDGES);
        uniform_edge_length = desired_edge_length;
        elapsed_time_counter = new Counter();
        edge_cross_counter = new Edge_Cross_Counter();
        spring_stiffness_factor = spring_stiffness;
        repulsion_factor = repulsion;
        displacement_factor = displacement;
        force_threshold = force_thresh;
    }

    public void run(double frac)
    {
        if( first_run )
        {
            fm3_graph = new FM3_Graph();

            System.out.println("Run started...");
            elapsed_time_counter.start();
        }
    }
}
```

```

        fm3_preprocessing_step();

        fm3_divide_et_impera();

        first_run = false;

        elapsed_time_counter.stop();
        System.out.println("Done. Total runtime:
"+elapsed_time_counter.get_elapsed_time());
        edge_cross_counter.calculate();
    }
}

// FM3 specific classes
private class FM3_Graph
{...92 lines }
private class FM3_Node
{...101 lines }
private class FM3_Edge
{...29 lines }

private void fm3_preprocessing_step()
{
    // Calculate the desired edge lengths
    for( Iterator<FM3_Edge> edge_it = fm3_graph.edges(); edge_it.hasNext() ; )
    {
        FM3_Edge edge = edge_it.next();
        edge.set_desired_length(uniform_edge_length);
    }
}

private void fm3_divide_et_impera()
{
    ArrayList<FM3_Graph> subgraphs = new ArrayList<>();

    fm3_divide_step( subgraphs );

    for( FM3_Graph subgraph : subgraphs )
    {
        fm3_multilevel_step( subgraph );
    }

    fm3_impera_step( subgraphs );
}

private void fm3_divide_step( ArrayList<FM3_Graph> subgraphs )
{
    // divide the disconnected graph into an array of connected graphs
    ArrayList<FM3_Node> already_visited = new ArrayList<>();
    Map<FM3_Node,Integer> component_id = new HashMap<>();
    int id = 0;
    for( Iterator<FM3_Node> node_it = fm3_graph.nodes(); node_it.hasNext() ; )
    {
        FM3_Node node = node_it.next();
        if( !already_visited.contains(node) )
        {
            ArrayList<FM3_Node> component_nodes = new ArrayList<>();
            fm3_find_component(node, id, component_id, already_visited,
component_nodes);
            subgraphs.add( fm3_subgraph( fm3_graph, component_nodes) );
            id++;
        }
    }

    // Assign Desired Edge Lengths
    for( FM3_Graph subgraph : subgraphs )
    {

```

```

        for( Iterator<FM3_Edge> edge_it = subgraph.edges() ; edge_it.hasNext()
; )
        {
            FM3_Edge edge = edge_it.next();
            edge.set_desired_length( edge.get_ancestor().get_desired_length()
);
        }
    }

private void fm3_find_component( FM3_Node node, int id, Map<FM3_Node,Integer>
component_id, ArrayList<FM3_Node> already_visited, ArrayList<FM3_Node>
component_nodes )
{...13 lines }

private void fm3_multilevel_step( FM3_Graph graph )
{
    // Coarsening Phase
    int i=0, sum=0, d=5, c=50;
    double s = 1.25;
    ArrayList<FM3_Graph> graph_array = new ArrayList<>();
    graph_array.add(graph);
    do
    {
        // Create a galaxy partitioning & collapse to a new graph
        fm3_select_by_star_mass( graph_array.get(i) );
        graph_array.add( fm3_collapse( graph_array.get(i) ) );
        i++;
        if( graph_array.get(i).edge_count() > (graph_array.get(i-
1).edge_count())/s )
            sum++;
    }
    while( graph_array.get(i).node_count() >= c && sum <= d );
    int last_level = i;

    // Refinement Phase

    // Generate a random initial placements for the nodes of the smallest (last)
multilevel
    fm3_generate_random_initial_placement( graph_array.get(i),
fm3_graph.node_count() );

    // Force Calculation for this multilevel
    fm3_grid_embedder( graph_array.get(i), i, last_level );

    // Loop through the remaining multilevels
    i--;
    while( i >= 0 )
    {
        // Generate initial placement for this multilevel
        fm3_generate_initial_placement( graph_array.get(i) );

        // Calculate forces and apply them on the nodes
        fm3_grid_embedder( graph_array.get(i), i, last_level );
        i--;
    }
}

private void fm3_grid_embedder( FM3_Graph graph, int current_multilevel, int
max_multilevel )
{
    // Init
    int max_iter, i=0;
    double springfactor=spring_stiffness_factor,
repulsionfactor=repulsion_factor, s=displacement_factor, t=force_threshold,
avg_force;

    // Calculate the maximum number of iterations

```

```

max_iter = fm3_calculate_max_iter(current_multilevel, max_multilevel);

// Loop until the stopping criterion is met
do
{
    // Restrict to grid
    if( run_restrict_to_grid )
        fm3_restrict_to_grid( graph, 10, 1, fm3_graph.node_count() );

    // Calculate Spring Forces
    fm3_calculate_spring_forces(graph);

    // Calculate Repulsive Forces
    fm3_force_approx(graph);

    // Calculate Resulting Forces
    fm3_calculate_resulting_force( graph, springfactor, repulsionfactor );

    // Calculate Displacement
    fm3_calculate_displacement( graph, s );

    // Re-position the nodes based on the displacement
    fm3_move_nodes(graph);

    // Calculate avrage total force
    avg_force = 0.0;
    for( Iterator<FM3_Node> iter = graph.nodes() ; iter.hasNext() ; )
    {
        FM3_Node node = iter.next();
        avg_force += Math.abs(node.get_displacement_x()) +
Math.abs(node.get_displacement_y());
    }
    avg_force = avg_force/graph.node_count();

    i++;
}
while( i<=max_iter && avg_force>=t );

// Restrict to grid
if( run_restrict_to_grid )
    fm3_restrict_to_grid( graph, 10, 1, fm3_graph.node_count() );
}

// Calculates and returns the maximum iteration for the grid_embedder
private int fm3_calculate_max_iter( double current_multilevel, double
max_multilevel )
{...8 lines }

private void fm3_calculate_spring_forces( FM3_Graph graph )
{
    for( Iterator<FM3_Node> iter = graph.nodes(); iter.hasNext() ; )
    {
        FM3_Node source = iter.next();
        source.clear_spring_force();
        Point2D src_node = new Point2D( source.get_visual_node().getX(),
source.get_visual_node().getY() );
        for( Iterator<FM3_Edge> edge_iter = source.edges() ;
edge_iter.hasNext(); )
        {
            FM3_Edge edge = edge_iter.next();
            FM3_Node target =
(source==edge.get_source()?edge.get_target():edge.get_source());
            Point2D trg_node = new Point2D( target.get_visual_node().getX(),
target.get_visual_node().getY() );
            double distance = src_node.distance(trg_node);
            double desired_length = edge.get_desired_length();
            if( desired_length != 0 && distance>0 )
            {

```

```

        double temp_force = Math.log10(distance/desired_length) *
distance;
        double distance_x = trg_node.getX() - src_node.getX();
        double distance_y = trg_node.getY() - src_node.getY();
        double temp_force_x = distance_x * temp_force;
        double temp_force_y = distance_y * temp_force;
        source.add_spring_force( temp_force_x, temp_force_y );
    }
}
}

private void fm3_force_approx( FM3_Graph graph )
{
    int crossover_point = 100;

    if( graph.node_count() <= crossover_point )
    {
        // Naive Force Calculation
        fm3_calculate_repulsion_forces( graph );
    }
    else
    {
        // Calculate using multipole method
        // Step 1. Create the Reduced Bucket QuadTree
        FM3_QuadTree reduced_bucket_quadtree = new FM3_QuadTree(graph,2);

        // Step 2. Calculate the repulsion forces, using the QuadTree & the
MultipoleFramework
        fm3_multipole_framework( graph, reduced_bucket_quadtree, 4 );
    }
}

private void fm3_calculate_repulsion_forces( FM3_Graph graph )
{
    for( Iterator<FM3_Node> src_iter = graph.nodes() ; src_iter.hasNext() ; )
    {
        FM3_Node s_node = src_iter.next();
        double force_sum_x = 0.0;
        double force_sum_y = 0.0;

        for( Iterator<FM3_Node> trg_iter = graph.nodes() ; trg_iter.hasNext() ; )
        {
            FM3_Node t_node = trg_iter.next();
            if( t_node == s_node )
                continue;

            double distance_x = s_node.get_visual_node().getX() -
t_node.get_visual_node().getX();
            double distance_y = s_node.get_visual_node().getY() -
t_node.get_visual_node().getY();
            double temp_force_x;
            double temp_force_y;
            Point2D node1 = new
Point2D(s_node.get_visual_node().getX(),s_node.get_visual_node().getY());
            Point2D node2 = new
Point2D(t_node.get_visual_node().getX(),t_node.get_visual_node().getY());
            double distance = node1.distance(node2);
            if( distance != 0 )
            {
                temp_force_x = (distance_x / Math.pow(distance, 2));
                temp_force_y = (distance_y / Math.pow(distance, 2));
            }
            else
            {
                // Same position, random force vector
                Random rand = new Random();

```

```

        int random = rand.nextInt(4);
        if( random == 0 ) // Down & Right
        {
            // random force between 1-5
            temp_force_x = (rand.nextInt(5)+1);
            temp_force_y = (rand.nextInt(5)+1);
        }
        else if( random == 1 ) // Down & Left
        {
            temp_force_x = (-1)*(rand.nextInt(5)+1);
            temp_force_y = (rand.nextInt(5)+1);
        }
        else if( random == 2 ) // Up & Right
        {
            temp_force_x = (rand.nextInt(5)+1);
            temp_force_y = (-1)*(rand.nextInt(5)+1);
        }
        else
        {
            temp_force_x = (-1)*(rand.nextInt(5)+1);
            temp_force_y = (-1)*(rand.nextInt(5)+1);
        }
        force_sum_x = force_sum_x + temp_force_x;
        force_sum_y = force_sum_y + temp_force_y;
    }
    s_node.clear_repulsion_force();
    s_node.add_repulsion_force( force_sum_x, force_sum_y );
}

private void fm3_calculate_resulting_force( FM3_Graph graph, double
springfactor, double repulsionfactor )
{...12 lines }

private void fm3_calculate_displacement( FM3_Graph graph, double s )
{
    for( Iterator<FM3_Node> iter = graph.nodes() ; iter.hasNext() ; )
    {
        FM3_Node node = iter.next();

        // calculate angle (radians)
        double angle, c, displ_factor;
        Point2D resulting_force = new Point2D( node.get_resulting_force_x(),
node.get_resulting_force_y() );
        Point2D former_displ_force = complex_divide( new Point2D(
node.get_displacement_x(), node.get_displacement_y() ), s );
        angle = Math.acos(
resulting_force.normalize().dotProduct(former_displ_force.normalize()) );

        // calculate c based on the angle
        if( angle <= (1.0/6.0)*Math.PI && angle >= (-1.0/6.0)*Math.PI )
            c = 2.0;
        else if( (angle <= (2.0/6.0)*Math.PI && angle > (1.0/6.0)*Math.PI)
            || (angle >= (-2.0/6.0)*Math.PI && angle < (-1.0/6.0)*Math.PI)
        )
            c = 3.0/2.0;
        else if( (angle <= (3.0/6.0)*Math.PI && angle > (2.0/6.0)*Math.PI)
            || (angle >= (-3.0/6.0)*Math.PI && angle < (-2.0/6.0)*Math.PI)
        )
            c = 1.0;
        else if( (angle <= (4.0/6.0)*Math.PI && angle > (3.0/6.0)*Math.PI)
            || (angle >= (-4.0/6.0)*Math.PI && angle < (-3.0/6.0)*Math.PI)
        )
            c = 2.0/3.0;
        else
            c = 1.0/3.0;
    }
}

```

```

        if( complex_norm(resulting_force) >
(c*complex_norm(former_displ_force))
            && (c*complex_norm(former_displ_force)) > 0.0 )
        {
            displ_factor = s * c * complex_norm(former_displ_force);
        }
        else
        {
            displ_factor = s * complex_norm(resulting_force);
        }
        Point2D current_displ_force = complex_multiply(
complex_divide(resulting_force,complex_norm(resulting_force)), displ_factor );
        node.set_displacement( current_displ_force.getX(),
current_displ_force.getY() );
    }
}

// Creates a new graph containing the selected nodes from the given graph
private FM3_Graph fm3_subgraph( FM3_Graph graph, ArrayList<FM3_Node> nodes )
{...34 lines }

private void fm3_select_by_star_mass( FM3_Graph graph )
{...65 lines }

// This function returns a collapsed version of this graph based on galaxy
partitioning
private FM3_Graph fm3_collapse( FM3_Graph graph )
{...149 lines }

// Gives each node of this graph a random initial placement
private void fm3_generate_random_initial_placement( FM3_Graph graph, int size )
{...22 lines }

// Gives each node of this graph a placement based on RPP-lists
private void fm3_generate_initial_placement( FM3_Graph graph )
{...78 lines }

// Re-Positions the nodes based on displacement
private void fm3_move_nodes( FM3_Graph graph )
{...18 lines }

// This function restricts the nodes of a graph to a given boundary
private void fm3_restrict_to_grid( FM3_Graph graph, double d1, double d2,
double amount )
{...56 lines }

private void fm3_impera_step( ArrayList<FM3_Graph> components )
{
    if( components.size() >1 )
    {
        // 1. Find the bounding rectangles of each component.
        ArrayList<Rectangle2D> boundaries = new ArrayList<>();
        Map<Rectangle2D,FM3_Graph> boundary_graph_link = new HashMap<>();
        boolean first_node;
        double offset = 30.0;
        for( FM3_Graph component : components )
        {
            double x_min=0.0, x_max=0.0, y_min=0.0, y_max=0.0;
            first_node = true;
            for( Iterator<FM3_Node> node_it = component.nodes() ;
node_it.hasNext() ; )
            {
                VisualItem node = node_it.next().get_visual_node();
                if( first_node )
                {
                    x_min = node.getX();
                    x_max = node.getX();
                    y_min = node.getY();

```

```

        y_max = node.getY();
        first_node = false;
    }
    else
    {
        if( x_min > node.getX() )
            x_min = node.getX();
        if( x_max < node.getX() )
            x_max = node.getX();
        if( y_min > node.getY() )
            y_min = node.getY();
        if( y_max < node.getY() )
            y_max = node.getY();
    }
}
// Create boundary + offset
Rectangle2D boundary = new Rectangle2D(x_min-offset, y_min-offset,
(x_max-x_min)+(offset*2.0), (y_max-y_min)+(offset*2.0) );
boundaries.add(boundary);
boundary_graph_link.put(boundary, component);
}

// 2. Foreach component calculate each rectangle's "Alignment".
for( int i=0 ; i<boundaries.size() ; i++ )
{
    Rectangle2D boundary = boundaries.get(i);
    FM3_Graph current_graph = boundary_graph_link.get(boundary);
    double width_height_ratio =
boundary.getWidth()/boundary.getHeight();
    if( !(aspect_ratio < 1 && width_height_ratio < 1)
        && !(aspect_ratio >= 1 && width_height_ratio >=1) )
    {
        boundaries.set( i, fm3_rotate_90( boundary, current_graph ) );
        boundary_graph_link.put(boundaries.get(i), current_graph);
    }
}

// 3. "Sort" the rectangles into decreasing height order
boundaries.sort(new compare_boundary_height());

// 4. Use the sorted array of rectangles to "Pack" them together
boolean first_box = true;
ArrayList<Rectangle2D> rows = new ArrayList<>(); // previous_row =
rows.get(rows.size()-1)
double res_height = 0.0, res_width = 0.0;
for( Rectangle2D bound : boundaries )
{
    double x_coord, y_coord;
    if( first_box )
    {
        // Place at the bottom left corner
        x_coord = bound.getWidth()/2.0;
        y_coord = bound.getHeight()/2.0;
        bound = fm3_move_graph( bound, boundary_graph_link.get(bound),
x_coord, y_coord );
        rows.add( new Rectangle2D(bound.getMinX(), bound.getMinY(),
bound.getWidth(), bound.getHeight() ) );
        res_height = bound.getHeight();
        res_width = bound.getWidth();
        first_box = false;
    }
    else
    {
        // Calculate resulting drawing area for each case with regard
to desired aspect ratio

        // Case 1. Place on existing row to the right of the previous
box.

```

```

        Rectangle2D drawing_area_case_1 = new Rectangle2D(0.0, 0.0,
Math.max( rows.get( rows.size()-1 ).getWidth()+bound.getWidth(), res_width),
res_height );
        if( drawing_area_case_1.getWidth() >
(drawing_area_case_1.getHeight()*aspect_ratio) )
            drawing_area_case_1 = new Rectangle2D( 0.0, 0.0,
drawing_area_case_1.getWidth(), drawing_area_case_1.getWidth()/aspect_ratio );
        else
            drawing_area_case_1 = new Rectangle2D( 0.0, 0.0,
drawing_area_case_1.getHeight()*aspect_ratio, drawing_area_case_1.getHeight() );

        // Case 2. Place on new row to the left above the previous box.
        Rectangle2D drawing_area_case_2 = new Rectangle2D(0.0, 0.0,
Math.max( bound.getWidth(), res_width), (res_height + bound.getHeight()) );
        if( drawing_area_case_2.getWidth() >
(drawing_area_case_2.getHeight()*aspect_ratio) )
            drawing_area_case_2 = new Rectangle2D( 0.0, 0.0,
drawing_area_case_2.getWidth(), drawing_area_case_2.getWidth()/aspect_ratio );
        else
            drawing_area_case_2 = new Rectangle2D( 0.0, 0.0,
drawing_area_case_2.getHeight()*aspect_ratio, drawing_area_case_2.getHeight() );

        // Case 3. Rotate 90deg and place on any existing row with
minimal width.
        Rectangle2D drawing_area_case_3 = null;
        Rectangle2D minimal_width_row = null;
        for( Rectangle2D row : rows )
        {
            if( row.getHeight() < bound.getWidth() )
                break;
            if( minimal_width_row == null ||
minimal_width_row.getWidth() > row.getWidth() )
                minimal_width_row = row;
        }
        if( minimal_width_row != null )
        {
            drawing_area_case_3 = new Rectangle2D( 0.0, 0.0, Math.max(
minimal_width_row.getWidth()+bound.getHeight(), res_width ), res_height );
            if( drawing_area_case_3.getWidth() >
(drawing_area_case_3.getHeight()*aspect_ratio) )
                drawing_area_case_3 = new Rectangle2D( 0.0, 0.0,
drawing_area_case_3.getWidth(), drawing_area_case_3.getWidth()/aspect_ratio );
            else
                drawing_area_case_3 = new Rectangle2D( 0.0, 0.0,
drawing_area_case_3.getHeight()*aspect_ratio, drawing_area_case_3.getHeight() );
        }

        // Choose the case that produces the smallest drawing area
        if( drawing_area_case_1.getWidth() <=
drawing_area_case_2.getWidth() &&
            (drawing_area_case_3 == null ||
drawing_area_case_1.getWidth() <= drawing_area_case_3.getWidth() ) )
        {
            // Go with case 1
            x_coord = rows.get( rows.size()-1 ).getCenterX() +
rows.get( rows.size()-1 ).getWidth()/2 + bound.getWidth()/2;
            y_coord = rows.get( rows.size()-1 ).getCenterY();
            bound = fm3_move_graph( bound,
boundary_graph_link.get( bound ), x_coord, y_coord );

            // Update rows array (extend the previous row), res_height
& res_width
            int row_index = ( rows.size()-1 );
            rows.set( row_index, new Rectangle2D(
rows.get( row_index ).getMinX(), rows.get( row_index ).getMinY(),
                ( rows.get( row_index ).getWidth()+bound.getWidth() ),
rows.get( row_index ).getHeight() ) );

```

```

        res_width = Math.max( res_width,
rows.get(row_index).getWidth() );
    }
    else if( drawing_area_case_2.getWidth() <
drawing_area_case_1.getWidth() &&
        (drawing_area_case_3 == null ||
drawing_area_case_2.getWidth() <= drawing_area_case_3.getWidth() ) )
    {
        // Go with case 2
        x_coord = bound.getWidth()/2;
        y_coord = rows.get(rows.size()-1).getCenterY() +
rows.get(rows.size()-1).getHeight()/2 + bound.getHeight()/2;
        bound = fm3_move_graph( bound,
boundary_graph_link.get(bound), x_coord, y_coord );

        // Update rows array (add a new row to it), res_height &
res_width
        rows.add( new Rectangle2D(bound.getMinX(), bound.getMinY(),
bound.getWidth(), bound.getHeight() ) );
        res_height += bound.getHeight();
        res_width = Math.max( bound.getWidth(), res_width);
    }
    else
    {
        // Go with case 3
        x_coord = minimal_width_row.getCenterX() +
(minimal_width_row.getWidth()/2) + (bound.getHeight()/2);
        y_coord = minimal_width_row.getCenterY();
        FM3_Graph current_graph = boundary_graph_link.get(bound);
        bound = fm3_rotate_90( bound, current_graph );
        boundary_graph_link.put(bound, current_graph);
        bound = fm3_move_graph( bound,
boundary_graph_link.get(bound), x_coord, y_coord );

        // Update rows array (extend an already existing row),
res_height & res_width
        int row_index = rows.indexOf(minimal_width_row);
        rows.set( row_index, new Rectangle2D(
rows.get(row_index).getMinX(), rows.get(row_index).getMinY(),
        (rows.get(row_index).getWidth()+bound.getWidth()),
rows.get(row_index).getHeight() ) );
        res_width = Math.max( rows.get(row_index).getWidth(),
res_width );
    }
}
}
}

// Comparator for Rectangle2D heights coordinate
private class compare_boundary_height implements Comparator<Rectangle2D>
{...7 lines }

// This function takes a graph and rotates it 90 degrees around its center
point, returns the new boundary
private Rectangle2D fm3_rotate_90( Rectangle2D boundary, FM3_Graph graph )
{...28 lines }

// This function takes a graph and moves it to a new position, returns the new
boundary
private Rectangle2D fm3_move_graph( Rectangle2D boundary, FM3_Graph graph,
double new_center_x, double new_center_y )
{...15 lines }

private class FM3_QuadTree
{...620 lines }

// The QuadTree node class

```

```

private class FM3_QuadTreeNode
{...261 lines }

private boolean overlap( Rectangle2D box1, Rectangle2D box2 )
{...12 lines }

private boolean neighbors( Rectangle2D box1, Rectangle2D box2 )
{...17 lines }
private boolean well_separated( FM3_QuadTreeNode node1, FM3_QuadTreeNode node2
)
{...19 lines }
private boolean well_separated( Rectangle2D box1, Rectangle2D box2 )
{...3 lines }

private Rectangle2D cover_region( FM3_QuadTreeNode node1, FM3_QuadTreeNode
node2 )
{...34 lines }

private Point2D complex_add( Point2D complex_1, Point2D complex_2 )
{...6 lines }
private Point2D complex_multiply( Point2D complex_1, Point2D complex_2 )
{...6 lines }
private Point2D complex_multiply( Point2D complex, double real )
{...6 lines }
private Point2D complex_pow( Point2D complex, int pow )
{...15 lines }
private Point2D complex_divide( Point2D complex_1, Point2D complex_2 )
{...15 lines }
private Point2D complex_divide( double real, Point2D complex_2 )
{...15 lines }
private Point2D complex_divide( Point2D complex_2, double real )
{...12 lines }
private Point2D complex_log( double base, Point2D complex )
{...6 lines }
private double complex_norm( Point2D complex )
{...4 lines }

private int binomial_coefficient( int n, int k )
{...9 lines }

private void fm3_multipole_framework( FM3_Graph graph, FM3_QuadTree tree, int
precision )
{
    // Part1, Trivial Case & Initializations
    if( tree.single_node() )
    {
        // Make a call to "calculate_repulsion_forces()" and EXIT
        fm3_calculate_repulsion_forces(graph);
        return;
    }

    // Part2, Bottom-Up Traversal
    // Foreach LeafNode: calculate the coefficients of Mp, due to all particles
    contained in the Sm-region.
    for( FM3_QuadTreeNode node : tree.get_leaf_nodes() )
    {
        node.calculate_Mp(precision);
    }

    // Calculate the Mp-Coefficients of the interior nodes
    for( FM3_QuadTreeNode node : tree.get_leaf_nodes() )
    {
        tree.set_active_node(node);
        while( tree.get_active_node() != tree.get_root() )
        {
            FM3_QuadTreeNode parent = tree.get_active_node().get_parent();
            tree.set_active_node( parent );
            if( tree.get_active_node().is_mp_coef_calculated() )

```

```

        continue;

// Detect if all children have been calculated
boolean mp_coef_calculated = true;
for( FM3_QuadTreeNode child : tree.get_active_node().get_children()
)
{
    if( !child.is_mp_coef_calculated() )
    {
        mp_coef_calculated = false;
        break;
    }
}
if( mp_coef_calculated )
    tree.get_active_node().calculate_Mp(precision);
else
    break;
}
}

// Part3, Top-Down Traversal
// Foreach child of the ROOT: "Calculate_Local_Expansions_and_Node_Sets()"
for( FM3_QuadTreeNode child : tree.get_root().get_children() )
{
    fm3_calculate_local_expansions_and_node_sets( tree, child, precision );
}

// Part4, Obtain The Forces

//      Have a set (C) of particles contained in the Sm-region.
//      Calculate Force_Local using coefficients of Lp.
//      Calculate Force_Direct using "D1-UNION-D3-UNION-C".
//      Calculate Force_Multipole using K.
//      Force_Repulsive = Force_Local + Force_Direct + Force_Multipole
for( FM3_QuadTreeNode node : tree.get_leaf_nodes() )
{
    // Obtain forces for each particle
    for( FM3_Node particle : node.get_Cx() )
    {
        // Calculate Force_Local[]
        Point2D local_z = new
Point2D(particle.get_visual_node().getX(),particle.get_visual_node().getY());
        Point2D local_z1 = new
Point2D(node.get_Sm().getCenterX(),node.get_Sm().getCenterY());
        double local_sumx;
        double local_sumy;
        Point2D lp_dist = new Point2D( (local_z.getX()-local_z1.getX()),
(local_z.getY()-local_z1.getY()) );
        Point2D lp_coef;
        Point2D local_sum = new Point2D(0.0,0.0);
        for( int l=1 ; l<=precision ; l++ )
        {
            if( node.get_lp_coef_x().size() == precision+1 )
            {
                lp_coef = new Point2D( node.get_lp_coef_x().get(l),
node.get_lp_coef_y().get(l) );
                local_sum = complex_add( local_sum, complex_multiply(
complex_multiply( lp_coef, complex_pow(lp_dist,(l-1)) ), (double)l ) );
            }
        }
        local_sumx = local_sum.getX();
        local_sumy = (-1.0) * local_sum.getY();

        // Calculate Force_Direct[]
        double direct_sum_x = 0.0;
        double direct_sum_y = 0.0;

```

```

        for( FM3_Node other_particle : node.get_Cx() ) // This node's own
particles
    {
        if( particle == other_particle )
            continue;
        double distance_x = particle.get_visual_node().getX() -
other_particle.get_visual_node().getX();
        double distance_y = particle.get_visual_node().getY() -
other_particle.get_visual_node().getY();
        Point2D node1 = new
Point2D(particle.get_visual_node().getX(),particle.get_visual_node().getY());
        Point2D node2 = new
Point2D(other_particle.get_visual_node().getX(),other_particle.get_visual_node().ge
tY());

        double distance = node1.distance(node2);
        if( distance != 0 )
        {
            double temp_force = 1/distance;
            direct_sum_x = direct_sum_x + (distance_x / distance) *
temp_force;
            direct_sum_y = direct_sum_y + (distance_y / distance) *
temp_force;
        }
    }
    for( FM3_QuadTreeNode other_node : node.get_D1() ) // Particles of
nodes in set D1
    {
        for( FM3_Node other_particle : other_node.get_Cx() )
        {
            double distance_x = particle.get_visual_node().getX() -
other_particle.get_visual_node().getX();
            double distance_y = particle.get_visual_node().getY() -
other_particle.get_visual_node().getY();
            Point2D node1 = new
Point2D(particle.get_visual_node().getX(),particle.get_visual_node().getY());
            Point2D node2 = new
Point2D(other_particle.get_visual_node().getX(),other_particle.get_visual_node().ge
tY());

            double distance = node1.distance(node2);
            if( distance != 0 )
            {
                double temp_force = 1/distance;
                direct_sum_x = direct_sum_x + (distance_x / distance) *
temp_force;
                direct_sum_y = direct_sum_y + (distance_y / distance) *
temp_force;
            }
        }
    }
    for( FM3_QuadTreeNode other_node : node.get_D3() ) // Particles of
nodes in set D3
    {
        for( FM3_Node other_particle : other_node.get_Cx() )
        {
            double distance_x = particle.get_visual_node().getX() -
other_particle.get_visual_node().getX();
            double distance_y = particle.get_visual_node().getY() -
other_particle.get_visual_node().getY();
            Point2D node1 = new
Point2D(particle.get_visual_node().getX(),particle.get_visual_node().getY());
            Point2D node2 = new
Point2D(other_particle.get_visual_node().getX(),other_particle.get_visual_node().ge
tY());

            double distance = node1.distance(node2);
            if( distance != 0 )
            {
                double temp_force = 1/distance;

```

```

        direct_sum_x = direct_sum_x + (distance_x / distance) *
temp_force;
        direct_sum_y = direct_sum_y + (distance_y / distance) *
temp_force;
    }
}

// Calculate Force_Multipole[]
Point2D multipole_z = new
Point2D(particle.get_visual_node().getX(),particle.get_visual_node().getY());
double multipole_sum_x;
double multipole_sum_y;
Point2D mp_dist;
Point2D multipole_sum = new Point2D(0.0,0.0);
Point2D other_node_coef;
for( FM3_QuadTreeNode other_node : node.get_K() )
{
    if( other_node != node )
    {
        Point2D multipole_z0 = new
Point2D(other_node.get_Sm().getCenterX(),other_node.get_Sm().getCenterY());
mp_dist = new Point2D( (multipole_z.getX()-
multipole_z0.getX()), (multipole_z.getY()-multipole_z0.getY()) );
other_node_coef = new Point2D(
other_node.get_mp_coef_x().get(0), other_node.get_mp_coef_y().get(0) );
multipole_sum = complex_add( multipole_sum, complex_divide(
other_node_coef, mp_dist ) );
for( int k=1 ; k<=precision ; k++ )
{
    other_node_coef = new Point2D(
other_node.get_mp_coef_x().get(k), other_node.get_mp_coef_y().get(k) );
multipole_sum = complex_add( multipole_sum,
complex_multiply(
complex_divide(complex_multiply(other_node_coef, (double)k), complex_pow( mp_dist,
(k+1))), (-1.0) ) );
}
}
multipole_sum_x = multipole_sum.getX();
multipole_sum_y = (-1.0) * multipole_sum.getY();

// Calculate Force_Repulsive[]
double result_x = local_sumx + direct_sum_x + multipole_sum_x;
double result_y = local_sumy + direct_sum_y + multipole_sum_y;
particle.clear_repulsion_force();
particle.add_repulsion_force(result_x, result_y);
}

// Exception handling
if( node.is_collapsed )
{
    FM3_Node group_particle = node.get_Cx().get(0);

    // Each collapsed particle is assigned the same forces affecting
the group_particle
for( FM3_Node particle : node.get_Cx_collapsed() )
{
    double result_x = group_particle.get_repulsion_force_x();
    double result_y = group_particle.get_repulsion_force_y();

    // Separate the particles with random force-vectors
Random rand = new Random();
if( rand.nextInt(2)==0 )
    result_x = result_x + 2.0;
else
    result_x = result_x - 2.0;
}
}

```

```

        if( rand.nextInt(2)==0 )
            result_y = result_y + 2.0;
        else
            result_y = result_y - 2.0;

        particle.clear_repulsion_force();
        particle.add_repulsion_force(result_x, result_y);
    }
}

private void fm3_calculate_local_expansions_and_node_sets( FM3_QuadTree T,
FM3_QuadTreeNode node, int precision )
{
    // Part 3.1 ( Find sets R(v), I(v), D1(v), D2(v) )
    ArrayList<FM3_QuadTreeNode> E = new ArrayList<>();
    if( node.get_parent() == T.get_root() )
        E.add(node.get_parent());
    else
    {
        for( FM3_QuadTreeNode add_node : node.get_parent().get_R() )
            E.add(add_node);
        for( FM3_QuadTreeNode add_node : node.get_parent().get_D1() )
        {
            if( !E.contains(add_node) )
                E.add(add_node);
        }
    }
    while( !E.isEmpty() )
    {
        FM3_QuadTreeNode temp_node = E.get(0);
        if( well_separated(node,temp_node) )
        {
            node.get_I().add(temp_node);
        }
        else if( node.get_Sm().getWidth() >= temp_node.get_Sm().getWidth() )
        {
            node.get_R().add(temp_node);
        }
        else if( !temp_node.get_children().isEmpty() )
        {
            for( FM3_QuadTreeNode child : temp_node.get_children() )
            {
                E.add(child);
            }
        }
        else if( neighbors(node.get_Sm(), temp_node.get_Sm()) )
        {
            node.get_D1().add(temp_node);
        }
        else
        {
            node.get_D2().add(temp_node);
        }
        E.remove(temp_node);
    }

    // Part 3.2 ( Calculate the coefficients of Lp )
    for( FM3_QuadTreeNode x : node.get_I() )
    {
        fm3_calculate_and_add_Lp(precision,x,node);
    }
    for( FM3_QuadTreeNode x : node.get_D2() )
    {
        fm3_calculate_and_add_Lp_of_leaf(precision,x,node);
    }
    if( node.get_parent().is_lp_coef_calculated() )

```

```

    {
        fm3_calculate_and_add_shifted_Lp(precision,node);
    }

// Part 3.3 ( Find D3(v), K(v) for leafNodes )
if( node.get_children().isEmpty() ) // node is a LeafNode
{
    E = (ArrayList<FM3_QuadTreeNode>) node.get_R().clone();
    while( !E.isEmpty() )
    {
        FM3_QuadTreeNode temp_node = E.get(0);
        if( !neighbors(node.get_Sm(), temp_node.get_Sm()) )
        {
            node.get_K().add(temp_node);
        }
        else if( neighbors(node.get_Sm(), temp_node.get_Sm())
            && temp_node.get_children().isEmpty() )
        {
            node.get_D3().add(temp_node);
        }
        else
        {
            for( FM3_QuadTreeNode child : temp_node.get_children() )
            {
                E.add(child);
            }
        }
        E.remove(temp_node);
    }

// Part 3.4 ( Recursion )
// Re-Run this function foreach child of v, if v!=LeafNode
if( !node.get_children().isEmpty() )
{
    for( FM3_QuadTreeNode child : node.get_children() )
        fm3_calculate_local_expansions_and_node_sets( T, child, precision
);
}

// Converts interaction_node's Mp_coefficients to Lp_coefficients and then adds
these to add_to_node's Lp_coefficients
private void fm3_calulate_and_add_Lp( int precision, FM3_QuadTreeNode
interaction_node, FM3_QuadTreeNode add_to_node )
{...57 lines }

// Coefficients of Lp for leafs
private void fm3_calulate_and_add_Lp_of_leaf( int precision, FM3_QuadTreeNode
interaction_node, FM3_QuadTreeNode add_to_node )
{...39 lines }

// Coefficients of Lp for parent
private void fm3_calulate_and_add_shifted_Lp( int precision, FM3_QuadTreeNode
add_to_node )
{...34 lines }
}

```

Appendix D. Touch Controls

DragControl.java

```
package prefix.controls;

import javafx.event.Event;
import javafx.event.EventType;
import javafx.scene.Cursor;
import javafx.scene.input.MouseButton;
import javafx.scene.input.MouseDragEvent;
import javafx.scene.input.MouseEvent;

// Touch-functionality
import javafx.scene.input.TouchEvent;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import prefix.data.Table;
import prefix.data.event.EventConstants;
import prefix.data.event.TableListener;
import prefix.data.util.Point2D;
import prefix.util.PrefuseLib;
import prefix.visual.VisualItem;

public class DragControl extends ControlAdapter implements TableListener {

    private VisualItem          activeItem;
    protected String           action;
    protected Point2D          down      = new Point2D();
    protected Point2D          temp     = new Point2D();
    protected boolean          dragged, wasFixed, resetItem;
    private boolean            fixOnMouseOver      = true;

    private static final Logger log =
LogManager.getLogger(DragControl.class);
    private final Delta          delta          = new Delta();

    public DragControl() {
    }

    public DragControl(String action) {
        this.action = action;
    }

    public DragControl(String action, boolean fixOnMouseOver) {
        this.fixOnMouseOver = fixOnMouseOver;
        this.action = action;
    }

    public void setFixPositionOnMouseOver(boolean s) {
        fixOnMouseOver = s;
    }

    @Override
    public void itemEvent(VisualItem item, Event e)
    {
        activeItem = item;
        if (e.getEventType() == MouseEvent.MOUSE_PRESSED)
        {
            MouseEvent ev = (MouseEvent) e;
            if( ev.getButton() == MouseButton.PRIMARY )
            {
                delta.x = item.getX() - ev.getSceneX();
                delta.y = item.getY() - ev.getSceneY();
            }
        }
    }
}
```

```

        item.getNode().setCursor(Cursor.MOVE);
    }
}
else if (e.getEventType() == MouseEvent.DRAG_DETECTED)
{
    MouseEvent ev = (MouseEvent) e;
    if( ev.getButton() == MouseButton.PRIMARY )
    {
        log.info("Drag Event detected");
        wasFixed = item.isFixed();
        resetItem = true;
        activeItem = item;
        item.setFixed(true);
        item.getTable().addTableListener(this);
        PrefuseLib.setX(activeItem, null, ev.getSceneX() +
delta.x);
        PrefuseLib.setY(activeItem, null, ev.getSceneY() +
delta.y);
    }
}
else if (e.getEventType() == MouseEvent.MOUSE_DRAGGED)
{
    MouseEvent ev = (MouseEvent) e;
    if (activeItem != null && ev.getButton() ==
MouseButton.PRIMARY)
    {
        item.setFixed(true); // fixing the item to the cursors
        position
        PrefuseLib.setX(activeItem, null, ev.getSceneX() +
delta.x);
        PrefuseLib.setY(activeItem, null, ev.getSceneY() +
delta.y);
    }
}
else if (e.getEventType() == MouseDragEvent.MOUSE_DRAG_RELEASED ||
MouseEvent.MOUSE_RELEASED)
{
    MouseEvent ev = (MouseEvent) e;
    if (activeItem != null && ev.getButton() == MouseButton.PRIMARY)
    {
        activeItem.setFixed(false);
        activeItem = null;
    }
}
// Touch-functionality
else if( e.getEventType() == TouchEvent.TOUCH_PRESSED )
{
    // Interpret just like a press with a mouse
    TouchEvent ev = (TouchEvent) e;
    if ( ev.getTouchCount()!=1 ) return; // Exit if more than 1
touchpoint are used
    delta.x = item.getX() - ev.getTouchPoint().getSceneX();
    delta.y = item.getY() - ev.getTouchPoint().getSceneY();
    item.getNode().setCursor(Cursor.MOVE);
}
else if( e.getEventType() == TouchEvent.TOUCH_RELEASED )
{
    // Interpret just like a release with a mouse
    if (activeItem != null)
    {
        activeItem.setFixed(false);
        activeItem = null;
    }
}
else if( e.getEventType() == TouchEvent.TOUCH_MOVED )
{

```

```

        // Interpret just like a drag event with a mouse. Combine
detection and dragged above?
        TouchEvent ev = (TouchEvent) e;
        if ( ev.getTouchCount()!=1 ) return; // Exit if more than 1
touchpoint are used
        if (activeItem != null) // DRAGGED
        {
            item.setFixed(true); // fixing the item to the cursors
position
            PrefuseLib.setX(activeItem, null,
ev.getTouchPoint().getSceneX() + delta.x);
            PrefuseLib.setY(activeItem, null,
ev.getTouchPoint().getSceneY() + delta.y);
        }
    }

    @Override
    public EventType<? extends Event> getEventType() {
        return MouseEvent.ANY;
    }

    public void tableChanged(Table t, int start, int end, int col, int
type) {
        if (activeItem == null || type != EventConstants.UPDATE
            || col != t.getColumnNumber(VisualItem.FIXED))
            return;
        int row = activeItem.getRow();
        if (row >= start && row <= end)
            resetItem = false;
    }

} // end of class DragControl

```

FocusControl.java

```
package prefix.controls;

import javafx.event.Event;
import javafx.event.EventType;
import javafx.scene.input.MouseButton;
import javafx.scene.input.MouseEvent;

// Touch-functionality
import javafx.scene.input.TouchEvent;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import prefix.Visualization;
import prefix.data.expression.Predicate;
import prefix.data.tuple.TupleSet;
import prefix.util.StringLib;
import prefix.visual.VisualItem;

public class FocusControl extends ControlAdapter {

    private String group = Visualization.FOCUS_ITEMS;
    protected String activity;
    protected VisualItem curFocus;
    protected int ccount;
    private static final Logger log = LogManager.getLogger(FocusControl.class);
    //protected int button = Control.LEFT_MOUSE_BUTTON;
    protected Predicate filter = null;

    private boolean focus_mode = false; // true if the previous event was a touch
    press
    private boolean focus_mode2 = false;
    private Visualization focus_vis = null;

    public FocusControl() {
        this(1);
    }

    public FocusControl(String focusGroup) {
        this(1);
        group = focusGroup;
    }

    public FocusControl(int clicks) {
        ccount = clicks;
    }

    public FocusControl(String focusGroup, int clicks) {
        ccount = clicks;
        group = focusGroup;
    }

    public FocusControl(int clicks, String act) {
        ccount = clicks;
        activity = act;
    }

    public FocusControl(String focusGroup, int clicks, String act) {
        ccount = clicks;
        activity = act;
        this.group = focusGroup;
    }

    public void setFilter(Predicate p) {
        this.filter = p;
    }
}
```

```

public Predicate getFilter() {
    return filter;
}

protected boolean filterCheck(VisualItem item) {
    if ( filter == null )
        return true;
    try
    {
        return filter.getBoolean(item);
    }
    catch ( Exception e )
    {
        log.info( e.getMessage() + "\n" + StringLib.getStackTrace(e) );
        return false;
    }
}

@Override
public void itemEvent(VisualItem item, Event e)
{
    if( e.getEventType() == MouseEvent.MOUSE_CLICKED )
    {
        MouseEvent ev = (MouseEvent)e;
        if ( !filterCheck(item) ) return;
        if ( ev.getButton()==MouseButton.PRIMARY && ev.getClickCount() ==
ccount )
        {
            if ( item != curFocus )
            {
                Visualization vis = item.getVisualization();
                TupleSet ts = vis.getFocusGroup(group);

                boolean ctrl = ev.isControlDown();
                if ( !ctrl )
                {
                    curFocus = item;
                    ts.setTuple(item);
                }
                else if ( ts.containsTuple(item) )
                {
                    ts.removeTuple(item);
                }
                else
                {
                    ts.addTuple(item);
                }
                runActivity(vis);
            }
            else if ( ev.isControlDown() )
            {
                Visualization vis = item.getVisualization();
                TupleSet ts = vis.getFocusGroup(group);
                ts.removeTuple(item);
                curFocus = null;
                runActivity(vis);
            }
        }
    }
    // Touch-Functionality
    // Focus the item when a tap is made on the item.
    // Focus multiple items by stationary (touch+hold).
    // Focus is cleared when another tap is made on another item, or stationary
on an already focused item.
    else if ( e.getEventType() == TouchEvent.TOUCH_PRESSED ) // a press
    {
        if ( !filterCheck(item) ) return;
        TouchEvent ev = (TouchEvent)e;

```

```

        if ( ev.getTouchCount()!=1 ) return; // Exit if more than 1 touchpoint
are used
        focus_mode = true;
    }
    else if ( focus_mode && e.getEventType() == TouchEvent.TOUCH_RELEASED ) //
a tap
    {
        if ( !filterCheck(item) ) return;
        TouchEvent ev = (TouchEvent)e;
        if ( ev.getTouchCount()!=1 ) return; // Exit if more than 1 touchpoint
are used
        if ( item != curFocus )
        {
            Visualization vis = item.getVisualization();
            TupleSet ts = vis.getFocusGroup(group);
            focus_vis = vis;
            curFocus = item;
            ts.setTuple(item);
            runActivity(vis);
        }
    }
    else if( e.getEventType() == TouchEvent.TOUCH_STATIONARY )
    {
        focus_mode = false;
        if ( !filterCheck(item) ) return;
        TouchEvent ev = (TouchEvent)e;
        if ( ev.getTouchCount()!=1 ) return; // Exit if more than 1 touchpoint
are used
        Visualization vis = item.getVisualization();
        TupleSet ts = vis.getFocusGroup(group);
        focus_vis = vis;
        if(ts.getTupleCount()==0) return; // Exit if nothing is currently being
focused
        if ( item == curFocus )
        {
            ts.removeTuple(item);
            curFocus = null;
        }
        else if( !ts.containsTuple(item) )
        {
            ts.addTuple(item);
        }
        runActivity(vis);
    }
    else if( e.getEventType() == TouchEvent.TOUCH_MOVED )
    {
        focus_mode = false;
    }
    else
    {
        focus_mode = false;
    }
}

@Override
public void event(Event e)
{
    if( e.getEventType() == TouchEvent.TOUCH_PRESSED )
    {
        focus_mode2 = true;
    }
    else if( focus_mode2 && e.getEventType() == TouchEvent.TOUCH_RELEASED ) //
tap on empty space
    {
        if( focus_vis!=null )
        {
            TupleSet focus_group = focus_vis.getFocusGroup(group);

```

```

        if( focus_group.getTupleCount(>0 ) // If at least 1 item is
focused
        {
            focus_group.clear();
            curFocus = null;
            runActivity(focus_vis);
        }
    }
    else
        focus_mode2 = false;
}

@Override
public EventType<? extends Event> getEventType() {
    return MouseEvent.ANY;
}

private void runActivity(Visualization vis) {
    if ( activity != null ) {
        vis.run(activity);
    }
}
} // end of class FocusControl

```

ZoomControl.java

```
package prefux.controls;

import javafx.event.Event;
import javafx.scene.input.MouseEvent;
import javafx.scene.input.ZoomEvent;
import prefux.FxDisplay;
import prefux.data.util.Point2D;
import prefux.visual.VisualItem;

public class ZoomControl extends ControlAdapter {

    private double yLast;
    private double zoomValue = 1;
    private Point2D anchor;
    //private Point2D down = new Point2D.Float();
    //private int button = RIGHT_MOUSE_BUTTON;
    private FxDisplay display;

    public ZoomControl() {
        // do nothing
    }

    public ZoomControl(FxDisplay _display) {
        display = _display;
    }

    @Override
    public void itemEvent(VisualItem item, Event e)
    {
        //event(e);
    }

    @Override
    public void event(Event e)
    {
        if( e.getEventType() == MouseEvent.MOUSE_PRESSED )
        {
            MouseEvent ev = (MouseEvent)e;
            if ( ev.isSecondaryButtonDown() )
            {
                yLast = ev.getY();
                anchor = new Point2D(ev.getX(),ev.getY());
            }
        }
        else if( e.getEventType() == MouseEvent.MOUSE_DRAGGED )
        {
            MouseEvent ev = (MouseEvent)e;
            if ( ev.isSecondaryButtonDown() )
            {
                // Drag up = Zoom out
                // Drag down = Zoom in
                double y = ev.getY();
                double dy = y-yLast;
                zoomValue += dy/100;
                if( zoomValue > 2 )
                    zoomValue = 2;
                else if( zoomValue < 0.05 )
                    zoomValue = 0.05;
                display.zoom( anchor, zoomValue);
                yLast = y;
            }
        }
        else if( e.getEventType() == ZoomEvent.ZOOM_STARTED)
        {
            ZoomEvent ev = (ZoomEvent)e;
            anchor = new Point2D(ev.getX(),ev.getY());
        }
    }
}
```

```
else if( e.getEventType() == ZoomEvent.ZOOM)
{
    ZoomEvent ev = (ZoomEvent)e;
    zoomValue = ev.getTotalZoomFactor();
    display.zoom(anchor, zoomValue);
}
}
} // end of class ZoomControl
```

Appendix E. Time & Edge-Crossings Counter

Time Counter

```
private class Counter
{
    Counter(){};

    public void start(){ time_start = System.nanoTime(); }
    public void stop()
    {
        time_end = System.nanoTime();
        time_nanos = time_end - time_start;
        time_micros = time_nanos / 1000.0;
        time_millis = time_micros / 1000.0;
        time_seconds = time_millis / 1000.0;
    }
    public double get_elapsed_time_nano(){ return time_nanos; }
    public double get_elapsed_time_micro(){ return time_micros; }
    public double get_elapsed_time_milli(){ return time_millis; }
    public double get_elapsed_time(){ return time_seconds; }

    private long time_start;
    private long time_end;
    private double time_nanos;
    private double time_millis;
    private double time_micros;
    private double time_seconds;
}
```

Edge-Crossings Counter (for FM³ Algorithm)

```
private class Edge_Cross_Counter
{
    Edge_Cross_Counter(){};

    public void calculate()
    {
        for( Iterator<FM3_Edge> iter = fm3_graph.edges() ; iter.hasNext() ; )
        {
            FM3_Edge edge = iter.next();
            FM3_Node src1 = edge.get_source();
            FM3_Node trg1 = edge.get_target();
            x1_line1 = (int)src1.get_visual_node().getX();
            x2_line1 = (int)trg1.get_visual_node().getX();
            y1_line1 = (int)src1.get_visual_node().getY();
            y2_line1 = (int)trg1.get_visual_node().getY();
            A_line1 = y2_line1 - y1_line1;
            B_line1 = x1_line1 - x2_line1;
            C_line1 = A_line1 * x1_line1 + B_line1 * y1_line1;
            for( Iterator<FM3_Edge> iter2 = fm3_graph.edges() ;
iter2.hasNext() ; )
                {
                    FM3_Edge edge2 = iter2.next();
                    if( edge == edge2 )
                        continue;
                    FM3_Node src2 = edge2.get_source();
                    FM3_Node trg2 = edge2.get_target();
                    x1_line2 = (int)src2.get_visual_node().getX();
                    x2_line2 = (int)trg2.get_visual_node().getX();
                    y1_line2 = (int)src2.get_visual_node().getY();
                    y2_line2 = (int)trg2.get_visual_node().getY();
                    A_line2 = y2_line2 - y1_line2;
                    B_line2 = x1_line2 - x2_line2;
                    C_line2 = A_line2 * x1_line2 + B_line2 *
y1_line2;
                }
        }
    }
}
```

```

        det = A_line1*B_line2 - A_line2*B_line1;
        if( det != 0 )
        {
            intersect_x = (B_line2*C_line1 -
B_line1*C_line2)/det;
            intersect_y = (A_line1*C_line2 -
A_line2*C_line1)/det;
            if( Math.min( x1_line1 , x2_line1 )
< intersect_x && Math.max( x1_line1 , x2_line1 ) > intersect_x
            && Math.min( y1_line1 ,
y2_line1 ) < intersect_y && Math.max( y1_line1 , y2_line1 ) > intersect_y
            && Math.min( x1_line2 ,
x2_line2 ) < intersect_x && Math.max( x1_line2 , x2_line2 ) > intersect_x
            && Math.min( y1_line2 ,
y2_line2 ) < intersect_y && Math.max( y1_line2 , y2_line2 ) > intersect_y )
                edge_cross_count++;
        }
    }
    edge_cross_count /= 2;
    System.out.println("Edge Crossings: "+edge_cross_count);
}

private double A_line1;
private double A_line2;
private double B_line1;
private double B_line2;
private double C_line1;
private double C_line2;
private double det;
private double intersect_x;
private double intersect_y;
private double x1_line1;
private double x2_line1;
private double y1_line1;
private double y2_line1;
private double x1_line2;
private double x2_line2;
private double y1_line2;
private double y2_line2;
private int edge_cross_count=0;
}

```

Edge-Crossings Counter (for Prefix's Algorithm)

```

private class Edge_Cross_Counter
{
    Edge_Cross_Counter(){};

    public void calculate()
    {
        for( Iterator<VisualItem> iter = m_vis.visibleItems(m_edgeGroup) ;
iter.hasNext() ; )
        {
            EdgeItem edge = (EdgeItem) iter.next();
            NodeItem src1 = edge.getSourceItem();
            NodeItem trg1 = edge.getTargetItem();
            x1_line1 = (int)src1.getX();
            x2_line1 = (int)trg1.getX();
            y1_line1 = (int)src1.getY();
            y2_line1 = (int)trg1.getY();
            A_line1 = y2_line1 - y1_line1;
            B_line1 = x1_line1 - x2_line1;
            C_line1 = A_line1 * x1_line1 + B_line1 * y1_line1;
            for( Iterator<VisualItem> iter2 =
m_vis.visibleItems(m_edgeGroup) ; iter2.hasNext() ; )
            {

```

```

        EdgeItem edge2 = (EdgeItem) iter2.next();
        if( edge == edge2 )
            continue;
        NodeItem src2 = edge2.getSourceItem();
        NodeItem trg2 = edge2.getTargetItem();
        x1_line2 = (int)src2.getX();
        x2_line2 = (int)trg2.getX();
        y1_line2 = (int)src2.getY();
        y2_line2 = (int)trg2.getY();
        A_line2 = y2_line2 - y1_line2;
        B_line2 = x1_line2 - x2_line2;
        C_line2 = A_line2 * x1_line2 + B_line2 * y1_line2;

        det = A_line1*B_line2 - A_line2*B_line1;
        if( det != 0 )
            {
                intersect_x = (B_line2*C_line1 -
B_line1*C_line2)/det;
                intersect_y = (A_line1*C_line2 -
A_line2*C_line1)/det;
                if( Math.min( x1_line1 , x2_line1 )
< intersect_x && Math.max( x1_line1 , x2_line1 ) > intersect_x
&& Math.min( y1_line1 , y2_line1 ) > intersect_y
&& Math.max( y1_line1 , y2_line1 ) < intersect_y
&& Math.min( x1_line2 , x2_line2 ) <
intersect_x && Math.max( x1_line2 , x2_line2 ) > intersect_x
&& Math.min( y1_line2 , y2_line2 ) <
intersect_y && Math.max( y1_line2 , y2_line2 ) > intersect_y )
                    edge_cross_count++;
            }
        }
        edge_cross_count /= 2;
        System.out.println("Edge Crossings: "+edge_cross_count);
    }

    private double A_line1;
    private double A_line2;
    private double B_line1;
    private double B_line2;
    private double C_line1;
    private double C_line2;
    private double det;
    private double intersect_x;
    private double intersect_y;
    private double x1_line1;
    private double x2_line1;
    private double y1_line1;
    private double y2_line1;
    private double x1_line2;
    private double x2_line2;
    private double y1_line2;
    private double y2_line2;
    private int edge_cross_count=0;
}

```

Appendix F. Tested Graphs

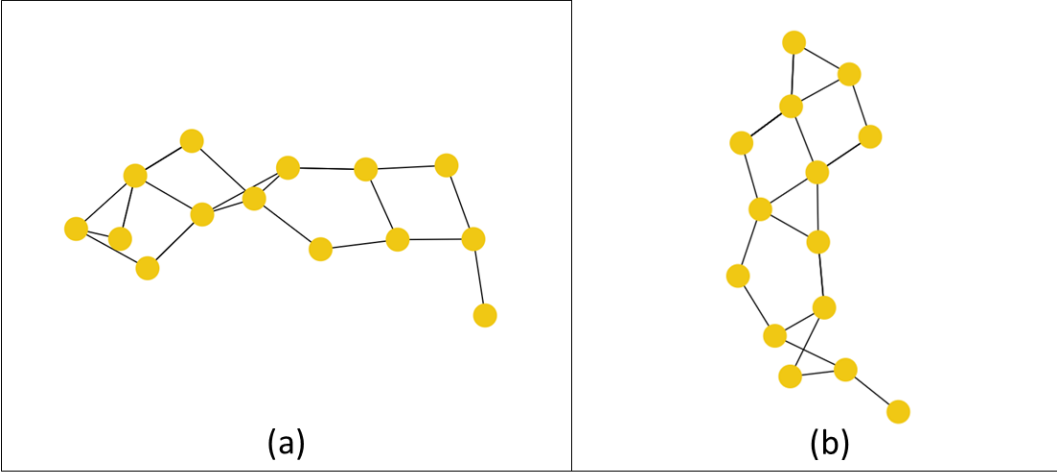


Figure 32. Graph "crs_dr.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

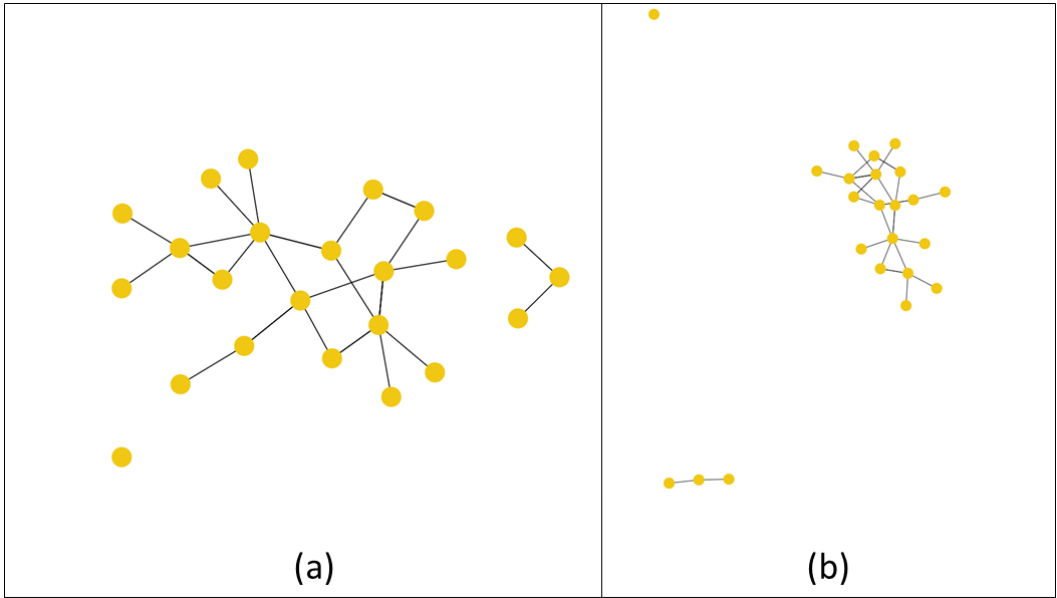


Figure 33. Graph "PCS.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

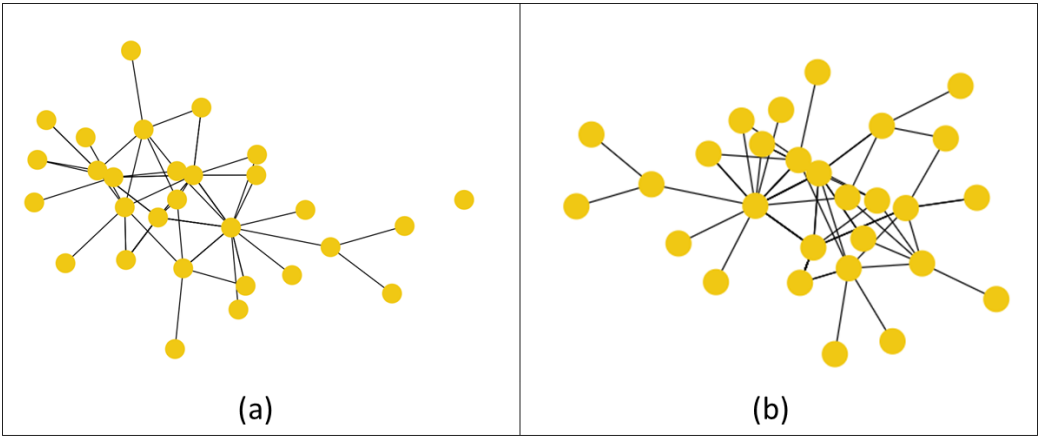


Figure 34. Graph "cmt.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

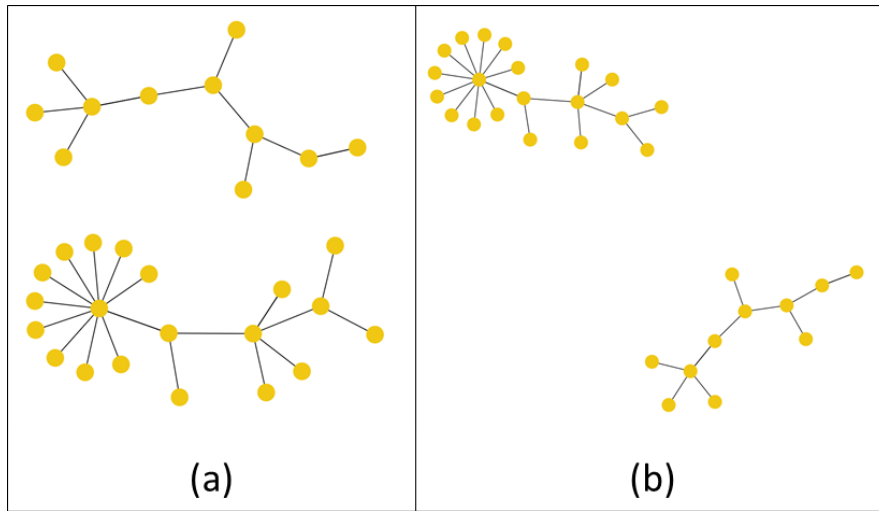


Figure 35. Graph "MICRO.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

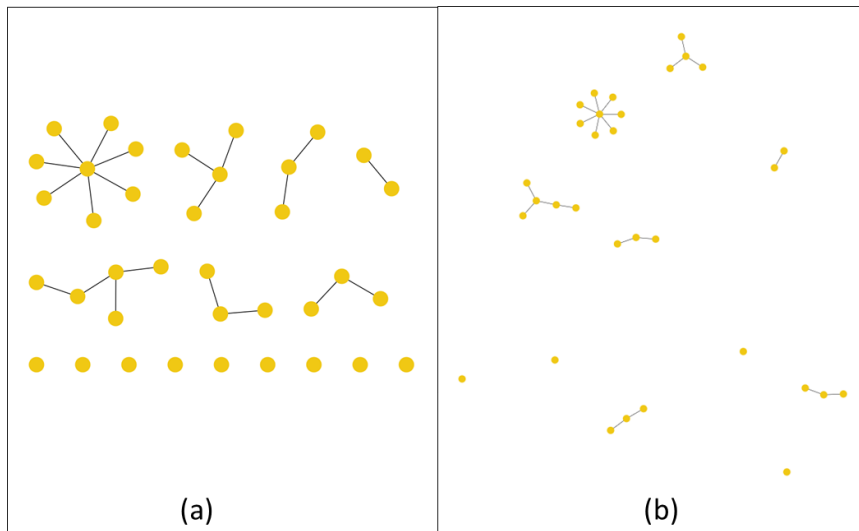


Figure 36. Graph "linkings.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

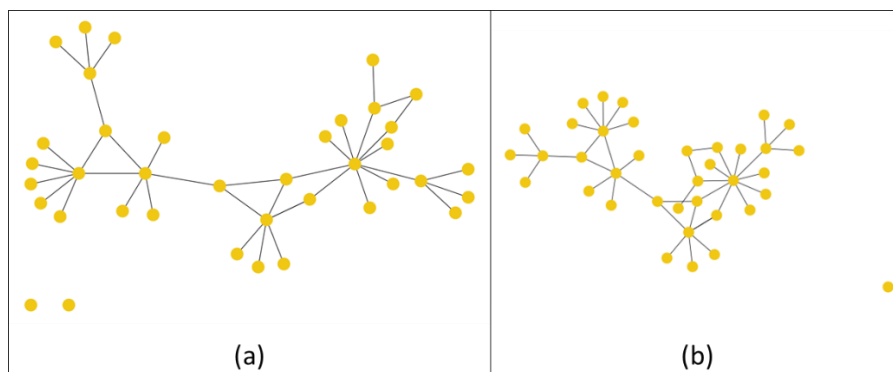


Figure 37. Graph "confOf.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

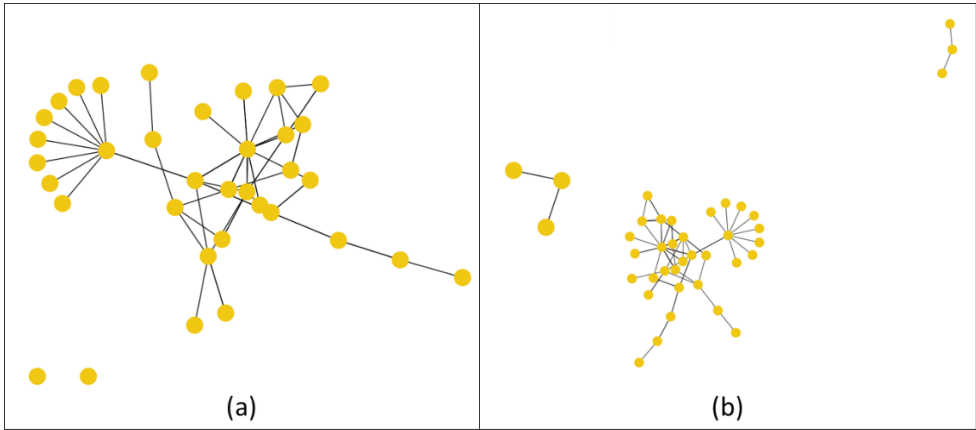


Figure 38. Graph "MyReview.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

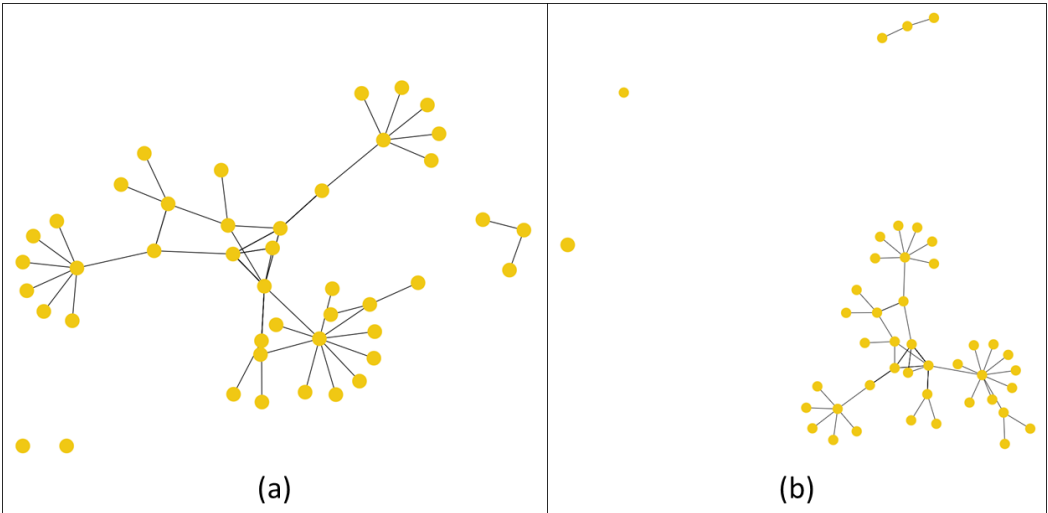


Figure 39. Graph "paperdyne.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

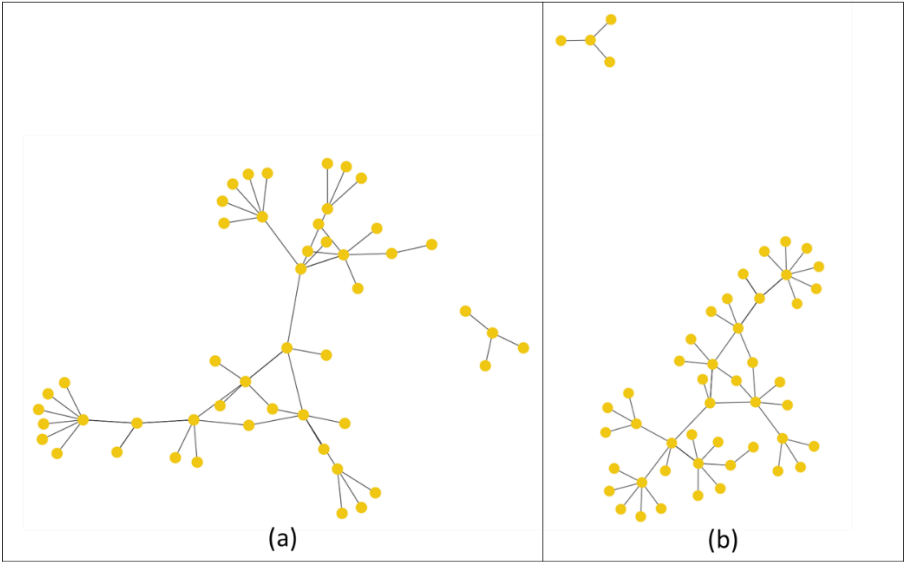


Figure 40. Graph "sigkdd.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

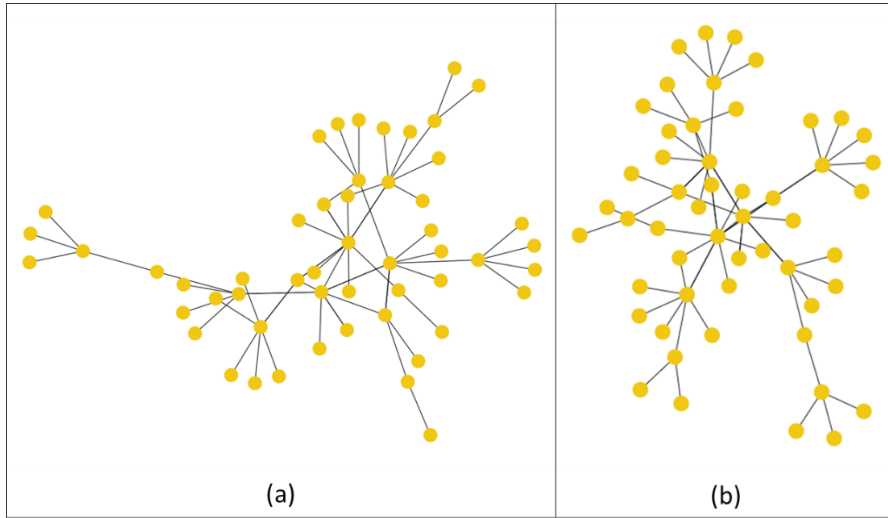


Figure 41. Graph "Cocus.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

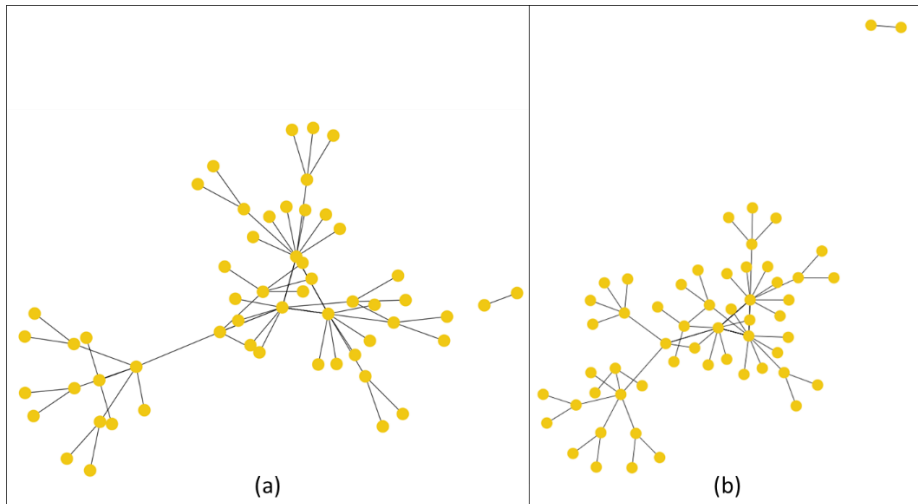


Figure 42. Graph "confious.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

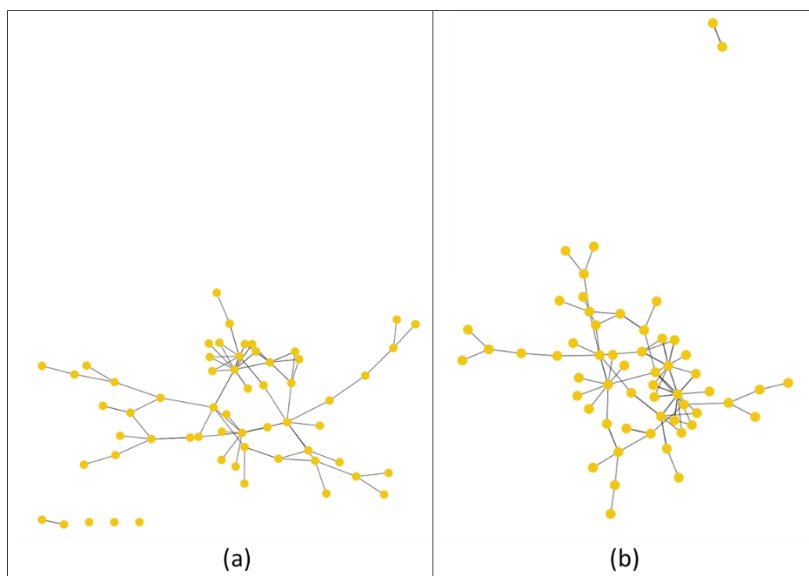


Figure 43. Graph "Conference.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

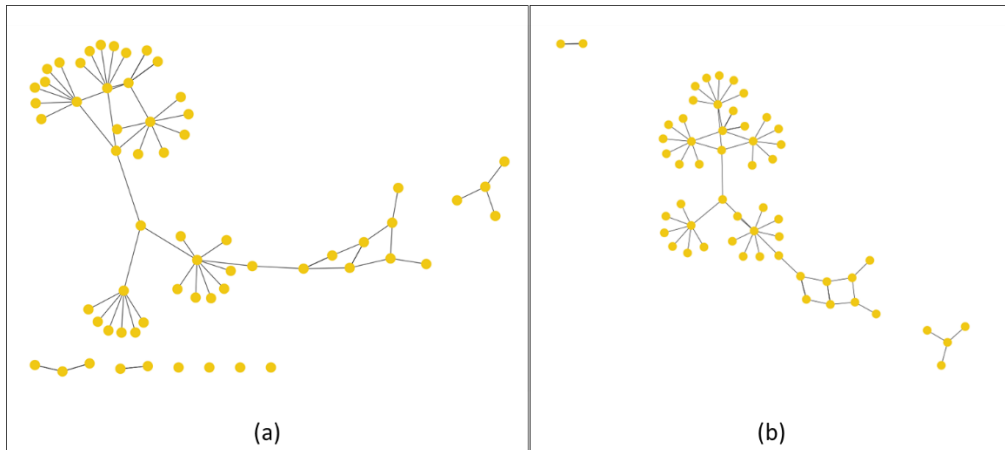


Figure 44. Graph "OpenConf.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

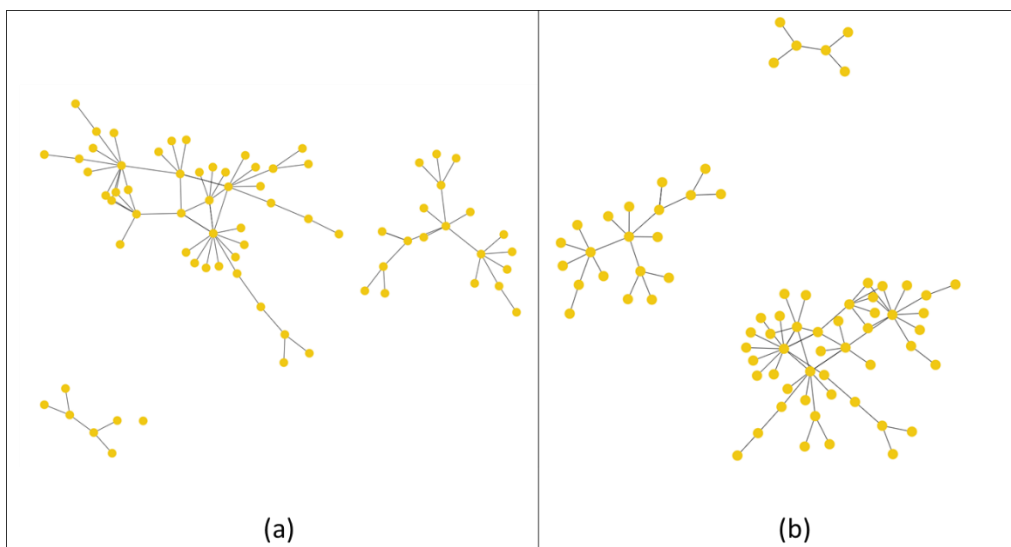


Figure 45. Graph "ekaw.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

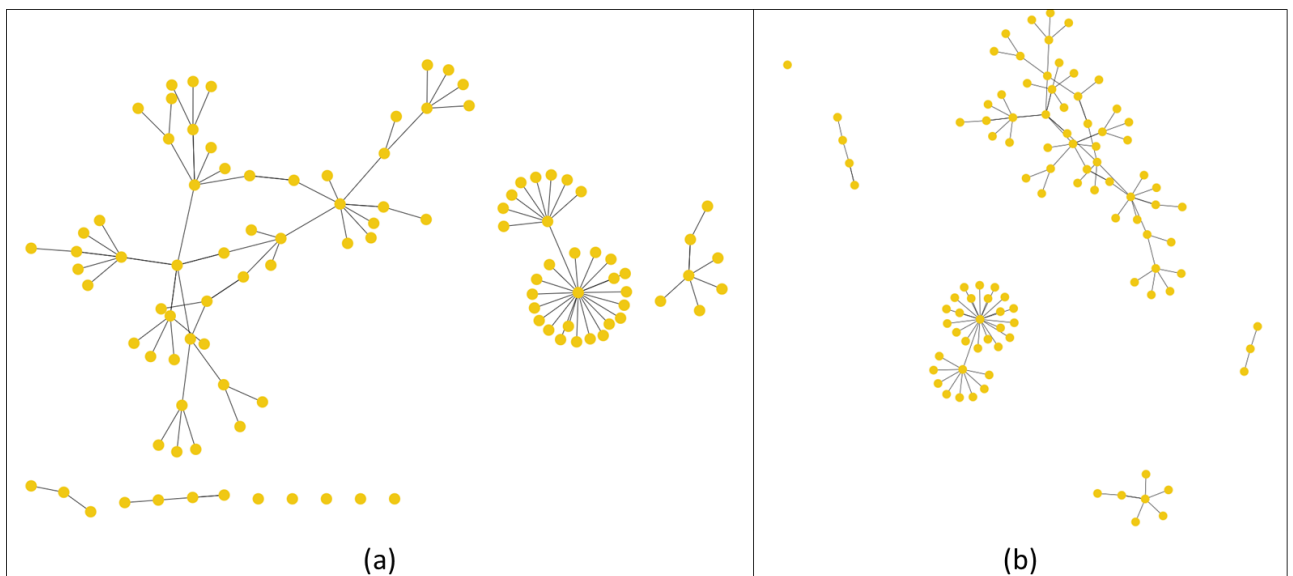


Figure 46. Graph "edas.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

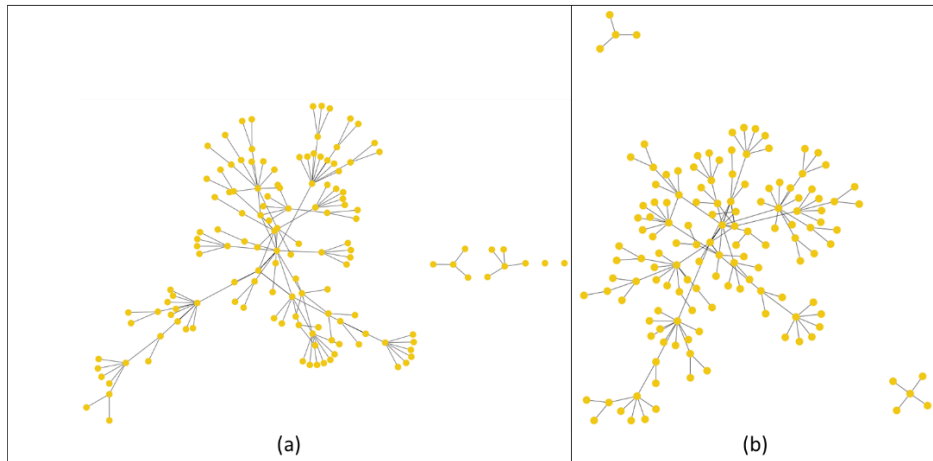


Figure 47. Graph "iasted.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

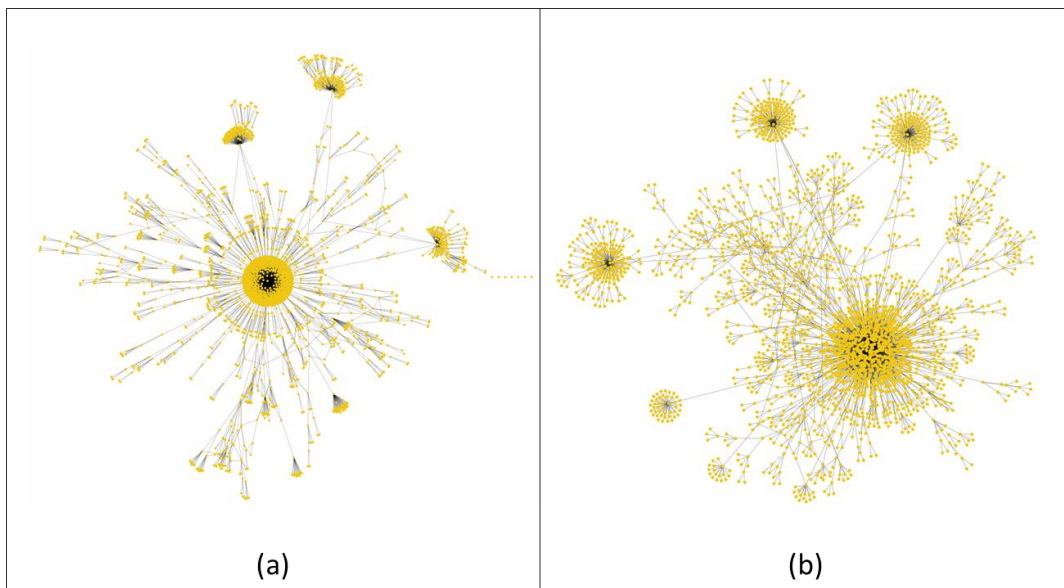


Figure 48. Graph "mouse.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.

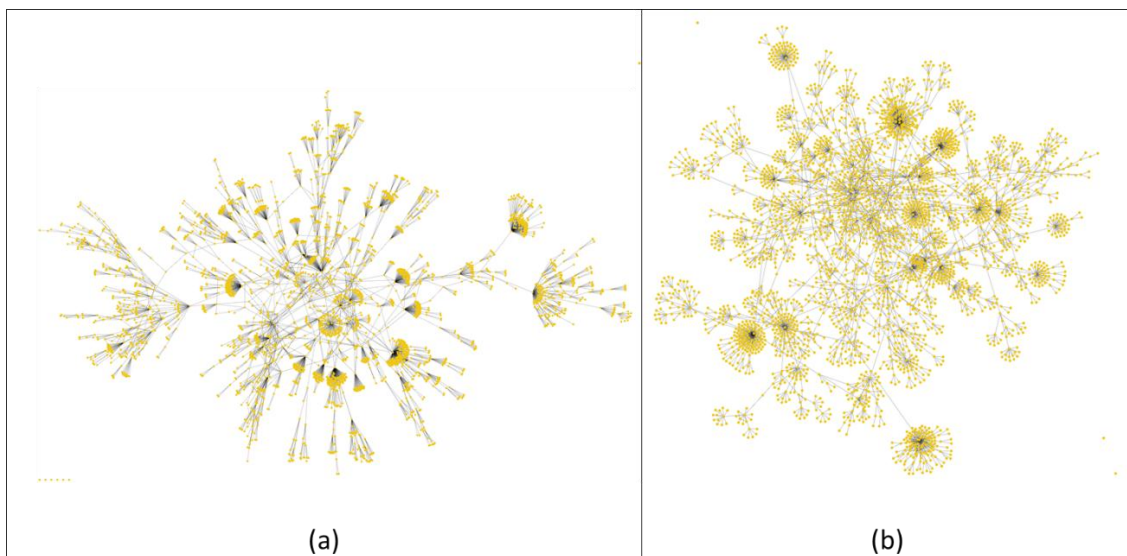


Figure 49. Graph "human.owl", (a) FM³ Algorithm, (b) Prefux's Algorithm.



Figure 50. Graph "FMA_small_overlapping_nci.owl", drawn with FM³ Algorithm.

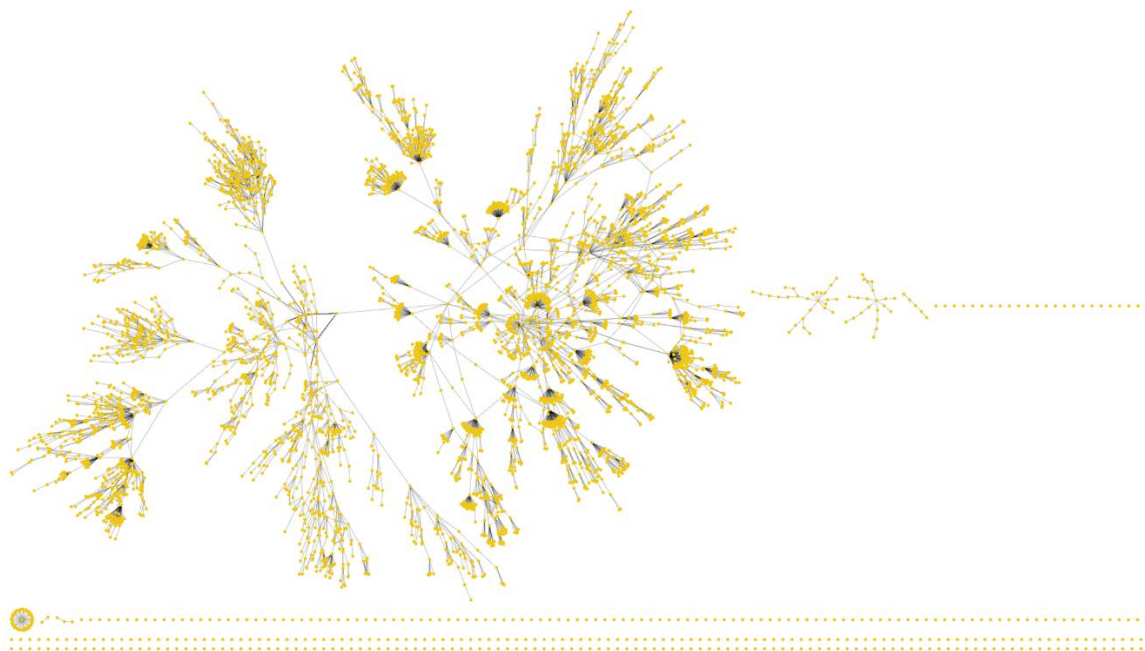


Figure 51. Graph "NCI_small_overlapping_fma.owl", drawn with FM³ Algorithm.



Figure 52. Graph "FMA_small_overlapping_snomed.owl", drawn with FM³ Algorithm.

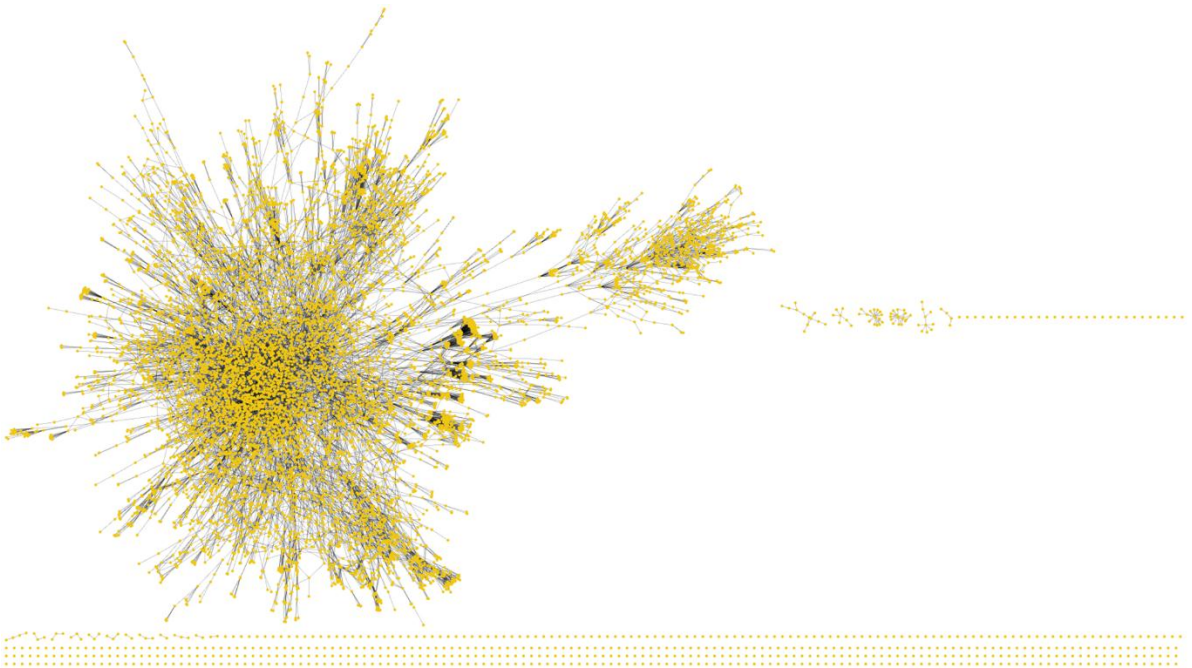


Figure 53. Graph "SNOMED_small_overlapping_fma.owl", drawn with FM³ Algorithm.



Figure 54. Graph "NCI_small_overlapping_snomed.owl", drawn with FM³ Algorithm.



Figure 55. Graph "SNOMED_small_overlapping_nci.owl", drawn with FM³ Algorithm.

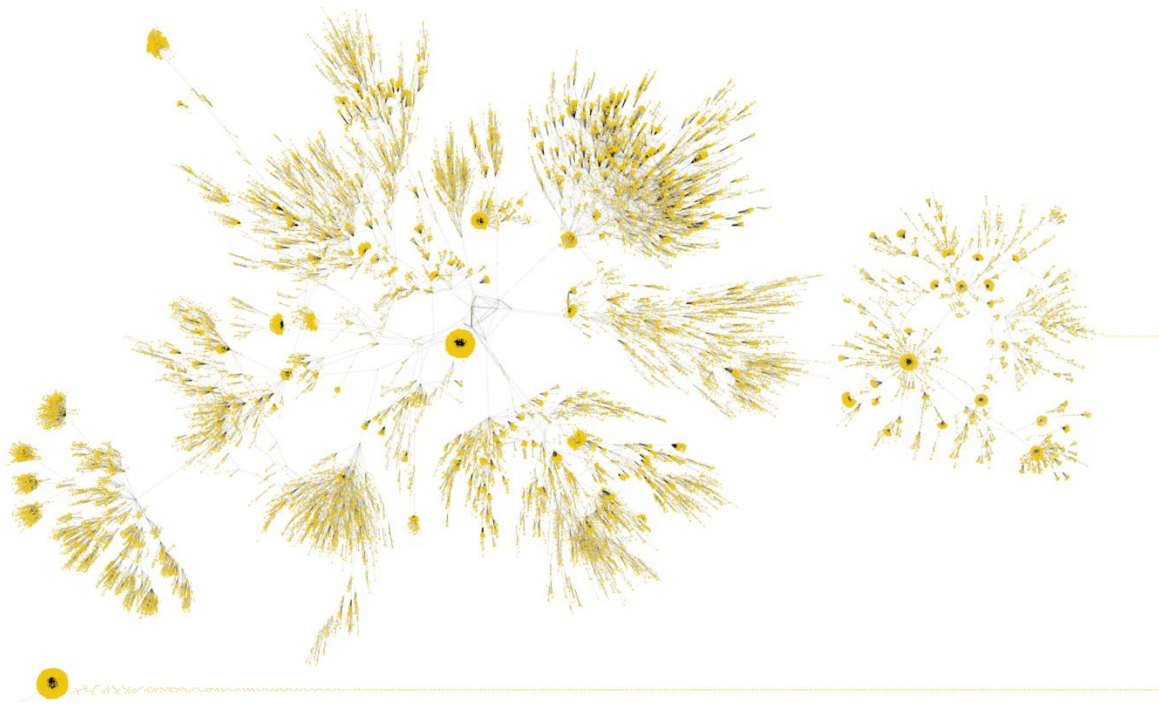


Figure 56. Graph "NCI_whole_ontology.owl", drawn with FM³ Algorithm.



Figure 57. Partial graph "FMA_whole_ontology.owl", drawn with FM³ Algorithm.