TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Institute of Computer Science
Chair of Theoretical Informatics

Automatic layout of large-scale graphs

Bachelor thesis

Student: Joonas Vali
Student code: 073857 IAPB
Advisor: Pavel Grigorenko

Tallinn
2013

I declare that this thesis is the result of my own research except as cited in the references. The thesis has not been accepted for any degree and is not concurrently submitted in candidature of any other degree.

Signature	
Name	Joonas Vali
Date	May 30, 2013

Abstract

In this thesis the author researches a problem of layouting large-scale graphs in two-dimensional environment. The author explains the basics of graphs and their real-world applications, defines the problem being researched, brings focus to major issues and describes some of the existing related work in this area. The researched problems that automatic layout algorithms face include algorithm performance and layout visual aesthetics. In this thesis the author defines large-scale graph as a graph with over 1000 nodes.

Due to the large amount of applications in the real world using graph theory, and authors' wish to keep the research on generic level, the algorithms researched and a solution implementation are not specific to one certain domain, but may be applied to several different domains.

The graph layout algorithms under focus are force-based. Force-based layout algorithms simulate the graph as a physical system with forces between the nodes that is being run iteratively until the system achieves an equilibrium state. There are descriptions of well known force-based algorithms and author's implementation of force-based algorithm is based on the principles researched and found appropriate.

The experimental part is done in two phases. The first phase is the implementation of a generic layout engine applicable to any Java-based application. Second, the engine integration with CoCoViLa, a visual model-based software development platform [1] in order to demonstrate the capability of the implemented layout engine to solve the real world graph visual layout problems.

Annotatsioon

Käesolevas töös kirjeldab autor suuremõõtmeliste graafide automaatse paigutusega seotud probleeme kahemõõtmelises keskkonnas, seletab lahti graafide elementaartõed ning tutvustab graafide kasutusvaldkondi. Autor toob esile graafide paigutamisel tekkivad suuremad probleemid ning kirjeldab lühidalt selles valdkonnas eelnevalt tehtud töid. Selle töö raames on suuremõõtmeline graaf defineeritud graafina, mis koosneb rohkem kui 1000 tipust.

Kuna graafidel on erinevates valdkondades väga palju rakendusi ja vastavalt autori soovile hoida vastav uurimus võimalikult abstraktsena, on uuritud algoritmid ja implementatsioon rakendatavad mitmele valdkonnale.

Graafi paigutamise algoritmid, mida selles töös vaadeldakse, põhinevad graafi simuleerimisel füüsikalise süsteemina. Vastavad algoritmid simuleerivad graafi füüsilise süsteemina, mille tipud omavad üksteise vahel tõukavat või tõmbavat jõudu, ning mida rakendatakse graafile iteratiivselt kuni piisavalt stabiilne olek on saavutatud. Töös on kirjeldatud üldtuntud jõududel põhinevaid algoritme ning autori loodud implementatsiooni, mille tööpõhimõte baseerub vastavatel algoritmidel.

Autori loodud eksperimentaalne implementatsioon jagub kaheks osaks. Esimene osa on üldine graafi paigutamise mootor, mida saab rakendada mistahes Java programmeerimiskeelel põhineval rakenduses. Teiseks, mootori integratsioon CoCoViLa keskkonnaga, mis on visuaalne mudelipõhine tarkvaraarendusplatform [1], et demonstreerida ja testida loodud mootori võimekust reaalsete probleemide lahendamisel.

Contents

1 Introduction

Visualising graphs helps humans to better understand the data presented in the graph form and to easier analyse and detect patterns and anomalies. Unfortunately, the ordinary graph itself never contains the data how it should be drawn to be visually most pleasing and semantically most accurate in the context the graph was designed for. This creates the graph drawing problem, often called *the graph layout problem* [2].

Graphs are used in many different fields and automatic graph layout algorithms are actively researched, with a lot of sufficient existing solutions for most of the cases imaginable. The goal of this work is not to exceed existing implementations, but merely an educational.

Automatic graph layout can be posed as an optimization problem where a sufficiently good layout is found by searching for a configuration of nodes and edges that is optimal with respect to various aesthetic criteria [3, p. 1].

The layout problem is generally not mathematically solvable and there is no right or best layout. The layout fitness is usually dependent on human perception and the graph domain. The actual algorithm complexity and outcome depends on the implementation. Compromisses are necessary between different areas of the algorithm and enhancing or optimizing one part may affect other parts negatively. For example, minimising edge crossings may reduce the graph symmetry [3, p. 1]. The balance between the different areas of the layout algorithm is the key to a visually appealing graph.

The major issue facing the automatic graph layouts is the diversity of the possible graphs and the areas of use. A certain graph layout algorithm might produce good results for one domain, but completely wrong, confusing or inapplicable result for another domain. Some application areas, like electronic circuit boards and VLSI(Very-large-scale integration) require very specific approach considering the length of edges, grouping, node placing discipline and, possibly, a total absence of edge crossings, while other graphs, used for representing relations between persons, may look better with organic placing and no bending of edges.

In visual modelling environments, such as CoCoViLa [1], schemes specifying domain-specific computational problems can be viewed as graphs with objects on a scheme as nodes of a graph and connections binding attributes of objects as edges. The edges between nodes may not be necessarily linear and could require breakpoint for edge bends to look meaningful and good to the observer. Nodes in a scheme occupy rectangular spaces and the location of the ports, the connection points for the edges, should be taken into account when placing nodes to minimize crossings, while not placing the edges over any adjacent nodes or the source nodes themselves. Figure 1 demonstrates a graph in CoCoViLa from the domain of web services.

On a mathematical level, graphs can have unlimited nodes and edges, while two-dimensional space cannot fit unlimited edges next to each other

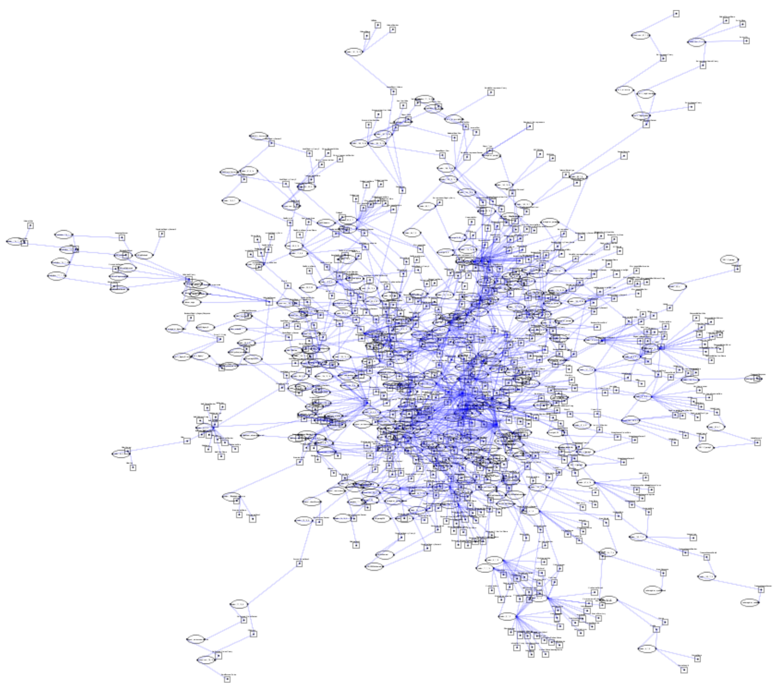


Figure 1: Graph in CoCoViLa

and still look aesthetically correct or trivially meaningful. This thesis does not investigate such cases, and assumes the graphs can be rendered on two dimensional plane without looking too cluttered.

The problem of graph layout can be split to two or more smaller subproblems, but is not required to, as solving the problem at once may result in a nicer look, but is more complex to solve, having more types of objects to depend on each other, requiring more dynamic approach. The subproblems include placing nodes into appropriate positions and placing breakpoints for the bends in edges. The research documented in this thesis tackles both these subproblems as different phases of the graph layout process.

Studies have shown that layouts generated by force based algorithms are preferred by users over any other layout algorithm implementations and even over user-generated layouts [3, p. 1]. Considering previous and the capacity limits of this document, the main algorithm under the focus in this thesis is force-based.

1.1 Motivation

CoCoViLa allows users to create and manipulate schemes in the visual manner and good placement is required for the graphs to look meaningful and reflect the data present at visual observation. The fundamental problem of drawing graphs was something that author of the thesis considered interesting.

The subject of this work was selected from the list of proposed topics by the Institute of Cybernetics at Tallinn University of Technology.

1.2 Goal

The goal of the work is to research a problem of automatic layout on large-scale graphs exceeding 1000 nodes, investigate and implement a convenient layout algorithm for the purpose to use it in the CoCoViLa environment.

A graph layout problem can be defined as follows [4]:

> "A fundamental and classical aesthetic is the minimisation of crossings between edges. In polyline drawings it is desirable to avoid bends in edges. In grid drawings, the area of the smallest rectangle covering the drawing should be minimal. In all graphic standards, the display of symmetries is desirable. It should be noted that aesthetics are subjective and may need to be tailored to suit personal preferences, traditions and culture."

1.3 Graphs

1.3.1 Introduction to graphs

A graph is a way of specifying relationships among a collection of items. A graph G is a pair of two sets, first is a set of objects (V), called nodes (vertices), with certain pairs of these objects connected by links called edges (set E) [5]:

$$G = (V, E)$$

Graphs can be divided to directed and undirected graphs, based whether their relations apply equally to both connected nodes or in case of the directed graph, the relation applies from one node to other, but not from the second to first node. The actual meaning of the relation is completely subjective to interpretator. The solutions researched, do not consider directed graphs, having little or no semantic meaning in general context this thesis is oriented to.

Graphs that allow multiple edges between a pair of nodes are called *multigraphs*. These type of graphs sometimes allow edge loops from a node to itself [6]. All graphs in this thesis are handled as multigraphs.

In exceptional situations it is even necessary to have edges with only one end, called half-edges, or no ends (loose edges). This thesis does not consider such cases [6].

Planar graphs are graphs that can be drawn on a plane in such way that none of its edges cross each other [7].

Real life applications for graphs may require nodes having limited edge connection points (ports) and in predefined positions relative to the node. Graphs considered in this thesis all have predefined ports. This property may cause graphs normally considered planar no longer be drawable on a plane with no edge crossings.

1.3.2 Applications of graphs

Modern applications of graphs include computer networks, maps, artificial intelligence, social networks, data mining, electronic circuits, virtual modeling, computer graphics, flow diagrams, class diagrams, use-case diagrams and many more.

The semantic meaning of the graph is usually domain specific. Some problems consider only the connection fact itself, leaving aside weights of the edges and nodes. A sample graph describing human relations can be found in Figure 2.

Other problems may require graphs having very specific edge weights and/or node locations, such as the Travelling Salesman Problem [8].

Graphs that have a fixed node locations are naturally not subjects for the graph drawing problem.

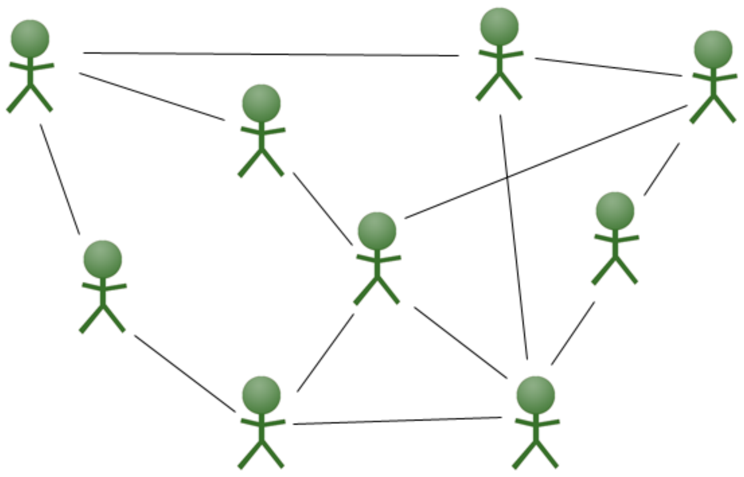


Figure 2: A graph representing relations between people

1.3.3 Graph drawing

Graph drawing addresses the problem of visualising the data present in graphs in most meaningful and understandable way to visual observation. Automatic graph layout algorithms are used to visualize the graphs in two-dimensional or three-dimensional environment. Different algorithms are used for different use cases. Layout algorithms are usually computationally expensive and, in most cases, the outcome fitness is subjective to the observer.

Research on graph drawing algorithms is spread over the broad spectrum of Computer Science, from VLSI to database design [4].

The edges between nodes may require additional algorithm for break-points if the use case prohibits diagonal straight lines. In this thesis the placing of edges is considered a second subproblem after the initial positioning of nodes is executed and sufficient node layout is found.

1.3.4 Problem specification

The graphs in this thesis have several properties usually not present in graph drawing problem. These properties make the algorithms to require additional time due to the calculations needed to produce better results for modeling software.

The use case expects the nodes to have rectangular bounds and to have dimensions, while force-based algorithms usually handle the nodes as singularities. Force-based algorithms use distance between the nodes to calculate the repulsive force between the nodes. Usually the distance is calculated from one node's center to another node center. In our case, the nodes might have large dimensions and applying the same kind of force to every node, might cause larger nodes to overlap. This requires redirection of the force hooks to prevent overlapping of geometrically massive nodes.

The second disparity with the typical force-based algorithms is due to the fact that instead of considering edges without bendings, the work in this thesis expects the edges to have breakpoints for connecting two unlevelled nodes, which creates a whole separate problem, independent from the classical graph drawing problem.

The problem is even harder to solve satisfactorily given that the implementation should produce aesthetically good looking graphs on many different areas of use, while having no special knowledge about the semantics of the graph. While it is obviously near impossible to consider all application domains and produce perfectly aesthetic layouts for any input, the force-based approach is the most suitable and likely to achieve good results without requiring enormous amount of programming and conditions to detect the domain from the graph structure, exceeding the limits of this work. This is the main reason the present thesis only considers the force-based algorithms.

2 Force-based algorithms

2.1 Introduction

Force-based algorithms treat nodes as spring connected objects, hold together by forces remotely or directly based on Hooke's Law. Hooke's law states: "stress is directly proportional to strain within the proportional limit" [9]. The algorithm is often extended by adding global forces, like gravitational force and electrically charged springs to avoid overlapping between different branches of nodes, which otherwise would not be aware of each other. Force-directed methods consider the edges to be straight and reduce the problem to only placing nodes. The purpose of springs is to position the nodes of a graph in two-dimensional or three-dimensional space so that all the edges are of more or less equal length and there are as few crossing edges as possible. The entire graph is then simulated as if it was a physical system. The forces are applied to the nodes, pulling them closer together or pushing them further apart. This is repeated iteratively until the system comes to an equilibrium state, i.e., their relative positions do not change anymore from one iteration to the next. The physical interpretation of this equilibrium state is that all the forces are in mechanical equilibrium [10]. The first force-directed algorithms to produce good layouts for graphs with over 1000 nodes is the algorithm of Hadany and Harel from 1999 [9].

Often the problem of drawing graphs is optimized by using heuristical approach, called multilevel or multiscale technique. This approach groups certain sets of adjacent nodes and simulates these as subsets, then recursion is used to move higher in the hierarchy, placing larger groups of nodes the same way. The resulting layout is described to have more global quality. Multilevel approach was first suggested by Reingold & Fruchterman in 1991 [2].

Experience shows that force-based algorithms produce good results with well known graphs in Graph Theory, such as the skeletons of the Platonic solids. They often produce highly symmetric results [11].

2.2 The advantages of force-based algorithms

Force-based algorithm works best on medium sized (50-100 nodes) graphs, being intuitive, flexible and simple. Algorithms developed for large graphs are also often force-based, because these algorithms adapt very well to unpredictable situations, but tend to get stuck in local minima. Force-based algorithms, being physical simulations, usually require no special knowledge about graph theory such as planarity. The resulting graphs have good quality and organic look. The force-based algorithms are well studied and have strong theoretical foundations [10].

2.3 The disadvantages of force-based algorithms

There are two limiting factors for standard force-directed algorithms in drawing large-scale graphs. Physical models often have many local minimums and for a large graph it is likely that the algorithm will settle for one first encountered. This may be improved to a limited extent by introducing slower cooling rate for the system, but for very large graphs it is near impossible to find a good solution using a force-based algorithm [12]. The found local minimum can be, in many cases, considerably worse than a global minimum, which translates into a low-quality drawing [10].

The second limiting factor is computational expense. In the algorithm of Fruchterman and Reingold [13], for any given node, repulsive force must be calculated for every other vertex per every iteration. The typical force-directed algorithms are in general considered to have a $O(n^3)$ running time [10], where n is the number of nodes of the input graph [12]. A limited optimization is possible by computing the repulsive force for only nearby nodes, but this approach still needs the detection of nearby nodes, which may be computationally as expensive as the full iteration over all nodes.

2.4 Application areas

The force-based layout style presents a multi-purpose layout for undirected graphs. It produces clear representations of complex networks and is especially suitable for application areas such as [14]:

- Bioinformatics
- Enterprise networking
- Knowledge representation
- System management
- WWW visualization
- Mesh visualization

2.5 Force-based algorithm suitability for CoCoVila

With additional properties of graphs stated above, the force-based algorithm should produce sufficient layouts for most cases in CoCoViLa.

In the case of a planar graph or a nearly planar graph, the force-based algorithm will provide a high quality layout, further improved by appropriate edge placing algorithm. The number of overlapping edges will be minimised, depending on the locations of ports on the nodes.

3 Implementation

3.1 Introduction

CoCoViLa allows users to automatically synthesize programs from declarative visual specifications (schemes), which are, in essence, multigraphs. Such domain-specific schemes consist of visual objects that need to be placed manually by the user for aesthetic look. Visual objects have predefined shape and dimension and contain ports for connecting with other objects. Connections (edges) may optionally have bendings (breakpoints).

The implemented solution discussed in the present section was done keeping in mind that the layout algorithm could be used on many different domains, thus any special information about the graph is not revealed to the layout. The information accessible by the algorithm is the size of nodes, the location of ports on a specific node and the connections between nodes without any specific limitations or predefined information about the domain. For some domains that require precise placing of nodes and edges the generic approach might not be suitable, but the practice has shown that for automatically generated graphs a generic layout is still a good starting point for further manual adjustments.

The interface providing the algorithm functionality is designed to be easily extendable, and considering any future work in this area, additional algorithms for specific domains may be easily added for improvement later.

3.1.1 Geometrical properties of nodes

The main geometrical property of a node that needs to be taken into account for layout algorithm is node's dimension, which defines its the bounds. The size is a rectangle shaped area defined by width and height. Figure6 depicts node's bounds with outermost rectangle. The algorithm drawing graphs faces the problem of considering the size of the nodes while placing them. Nodes, which have one dimension exceeding the other dimension a number of times create a problem of centered uniformly distributed force, leaving out the farther edge parts or making the force there significantly lower (Figure 3.A). The other extremum is creating too much force, which produces unnecessary hole in the layout for theoretically extra long nodes (Figure 3.B). From the Figure 3 it is possible to determine that there is no right amount of uniformly distributed force, which could solve this problem. By increasing or decreasing the force one of the stated problems is always magnified.

The solution this thesis proposes is to redirect the force vector for nodes to the nearest edge of the node, which can be seen in Figure 4. The imaginary line from the center of one node to the center of the other node creates a cutting point with the two edges of the nodes, and the force strength is

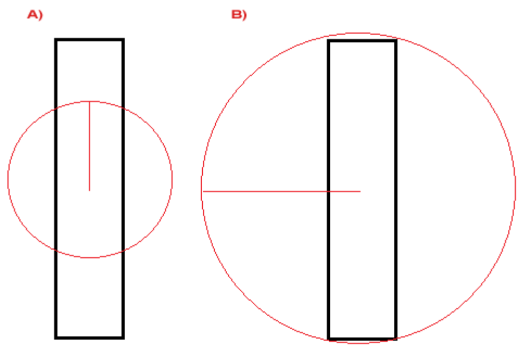


Figure 3: A) Node having too small forcefield, creating possible overlapping. B) Node having too big forcefield, creating excessive force on sides.

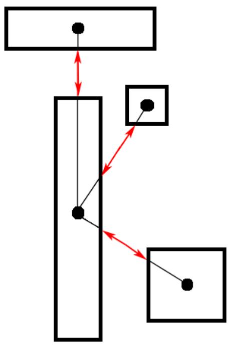


Figure 4: Redirecting force vectors

calculated from the length and direction of the resulting segment between the nodes.

3.1.2 Central node

An invisible central node is added to the graph, which is treated like it is connected to every other node. It acts like a real node, and moves itself according to the forces as every other node. Its purpose is to keep unconnected graph parts together, which would otherwise be pushed farther and farther away from each other. It also enables to keep satellite groups from being pushed too far from the main group. See Figure 18 on page 36 for an example of satellite groups.

3.1.3 Ports

Abstract graphs in mathematical sense connect edges to nodes, with no predefined location for the connecting point, there is just the connection fact.

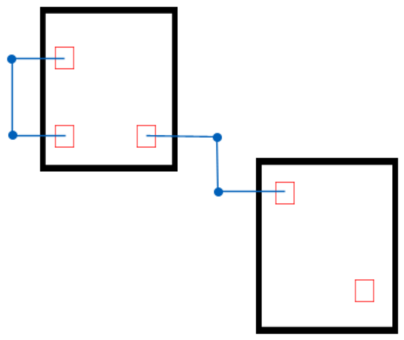


Figure 5: Example of ports and edges

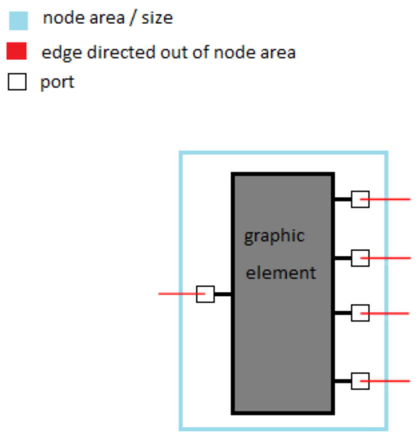


Figure 6: Edges directed from ports to nearest edge of the area

The present work deals with nodes that have shapes and zero or more ports, which handle the connections to other ports, either owned by the same or foreign node. Example of ports can be seen in Figure 5. Edges are connected to a port in node. Port have predefined locations and one port connects to another with an edge or is unconnected. The locations of the ports can be anywhere within the area of a node.

In most cases it would look aesthetic to draw the edge from the port to nearest bound of the node area to keep it from drawing over the node graphic, and due to the unforeseen cases possibly arising in real life situations, this is the strict policy chosen throughout the work examined in this thesis. An explanatory scheme can be seen on Figure 6.

3.2 Breakpoints

3.2.1 Introduction to breakpoints

To connect two unaligned nodes with a non-diagonal edge, that is, the edge with only horisontal and vertical segments, there is a need for one or more breakpoints to connect the different segments of the edge. Breakpoints act as invisible nodes connecting two edges for a bend.

Positioning breakpoints creates a similarly difficult problem of positioning nodes. The obvious restriction is that the two breakpoints connected by edges, should be positioned on the same vertical or horizontal level, if the aesthetic criteria defines that the edges should not be diagonal.

The other restriction is that edges should never cross a node. It would also make sense to avoid previously placed breakpoints, because the result of edges intersecting at corners may create visually complex drawing, where several edges connect. In the current implementation, the former is not implemented and edges often join graphically. This restriction is excluded from implementation because graphs in CoCoViLa often allow multiple edges to connect to a single port and it would create unnecessary complications and a confusingly looking result.

3.2.2 Preparation stage

Internally, the layout engine holds nodes of a graph in the memory as a linked list of node objects, which have predefined dimensions and location on absolute scale. When the algorithm places breakpoints, it needs to search for clear paths so it would not place edges over any nodes.

Detecting if a certain coordinate is vacant using linked list is computationally costly and requires iterating over every node and calculating if the coordinate is a part of any known node area. Doing this routine for every coordinate, which could be considered a part of a potential edge, requires a lot of computing power.

The algorithm accounts this by precomputing a two-dimensional boolean array, what is called a *local map* (Figure 7), surrounding two nodes connected by an edge. The map is computed with a reasonable buffer zone around both nodes, such that the potential edge has a possible path around both of the nodes, if the port placement requires it. The map includes every node that intersects the observed area. Also the bounds of nodes placed on the map are slightly expanded, so there would not be the edges placed too close to the side of the node later on. Two-dimensional array has a very fast access, which allows the breakpoint placing algorithms to make more attempts to place the edge successfully later on, without having a large performance overhead on the algorithm.

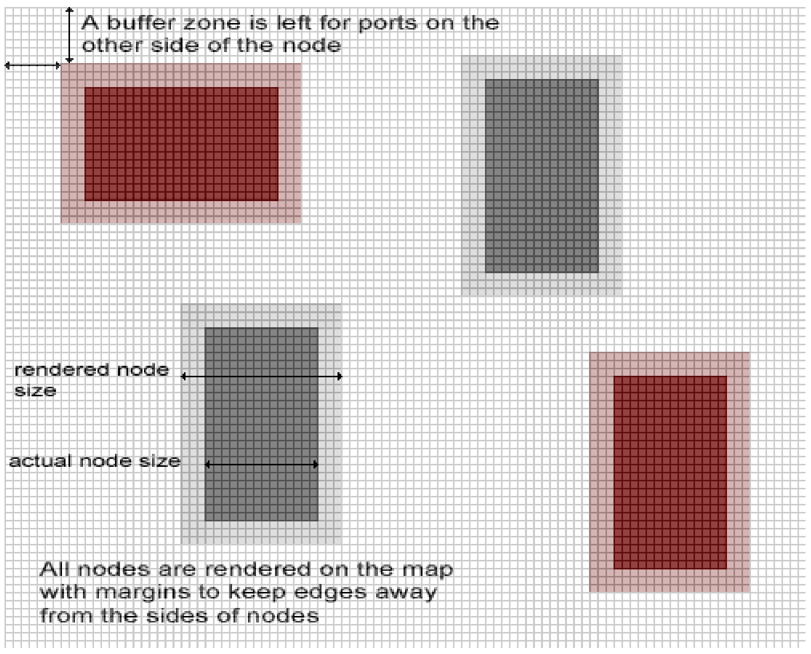

A buffer zone is left for ports on the other side of the node
rendered node size
actual node size
All nodes are rendered on the map with margins to keep edges away from the sides of nodes

Figure 7: A computed local map of two selected nodes shown as red.

3.2.3 Placing breakpoints

The strategy of placing breakpoints consists of searching for vacant "L-shaped" routes. It starts with two coordinates that are the coordinates calculated by directing the edges out of the source node area to nearest vacant area.

First, the algorithm tries to detect if the coordinates are aligned and have a vacant route between them. If these conditions are met, then no breakpoints are needed, and the operation is successfully terminated for this edge.

Second, it tries to detect, if the two coordinates can be connected with an "L-shaped" edge. For this the algorithm checks if one of the two possible routes is free from one coordinate to other. If there is a free route, then a breakpoint is placed in the calculated corner, and operation is successfully terminated.

Third, the absolute middle point for the two coordinates is computed. Then, the algorithm tries to reapply the second routine to both coordinates again, but using the middle point as the goal this time, which might result in two "L-shaped" edges connected to each other. The three breakpoints are applied only if the free route was found.

If all previous steps failed to find a free route, the algorithm tries to apply the third routine in a nested loop, this time placing the middle point to all over the local map area, with small gaps between placing it. This is the last attempt and is very likely to succeed in the most cases.

In rare cases, when the route couldn't be found, the edge is left without any breakpoints, resulting in a diagonal line, indicating that the route would probably look better if the user placed it manually. Sometimes when the graph is complex and the layout algorithm has generated a tightly packed result, a single diagonal line over several nodes might even give a clearer image to the observer than the one with many breakpoints.

Final option would be to apply a kind of the path searching algorithm to the edge, which couldn't be placed otherwise. This step is left as a future work on the project as it would not give any reasonable advantage for the current implementation.

3.3 Layout Engine architecture

In this section two class diagrams are shown and elaborated. Class diagrams represent the structure and relations between different parts of the algorithm used in the implementation.

3.3.1 Nodes layout module

First, a class diagram of nodes and algorithms involved placing them is shown in Figure 8.

- `Node` class contains information about the `Node` basic information like dimension, location, and list of ports.

- `PhysicalNode` is a wrapper class for `Node` class, and contains some additional information specific to force-layout, like velocity and mass of the node.

- `Port` class is referencing its owner node and the port it is connected to or null if port is not connected to any other port. It also contains a list of breakpoints for this connection between the two ports.

- `Force` class is a simple class containing two doubles for force directions on a plane and a utility method, which allows to calculate force absolute value.

- `Graph` class is a container for the collection of nodes.

- `Layout` interface is designed to allow generic interaction with the layout objects, which is necessary if more layouts are to be added in the future.

- `ForceLayout` class is an implementation of layout interface and contains the force based algorithm implementation described in this thesis.

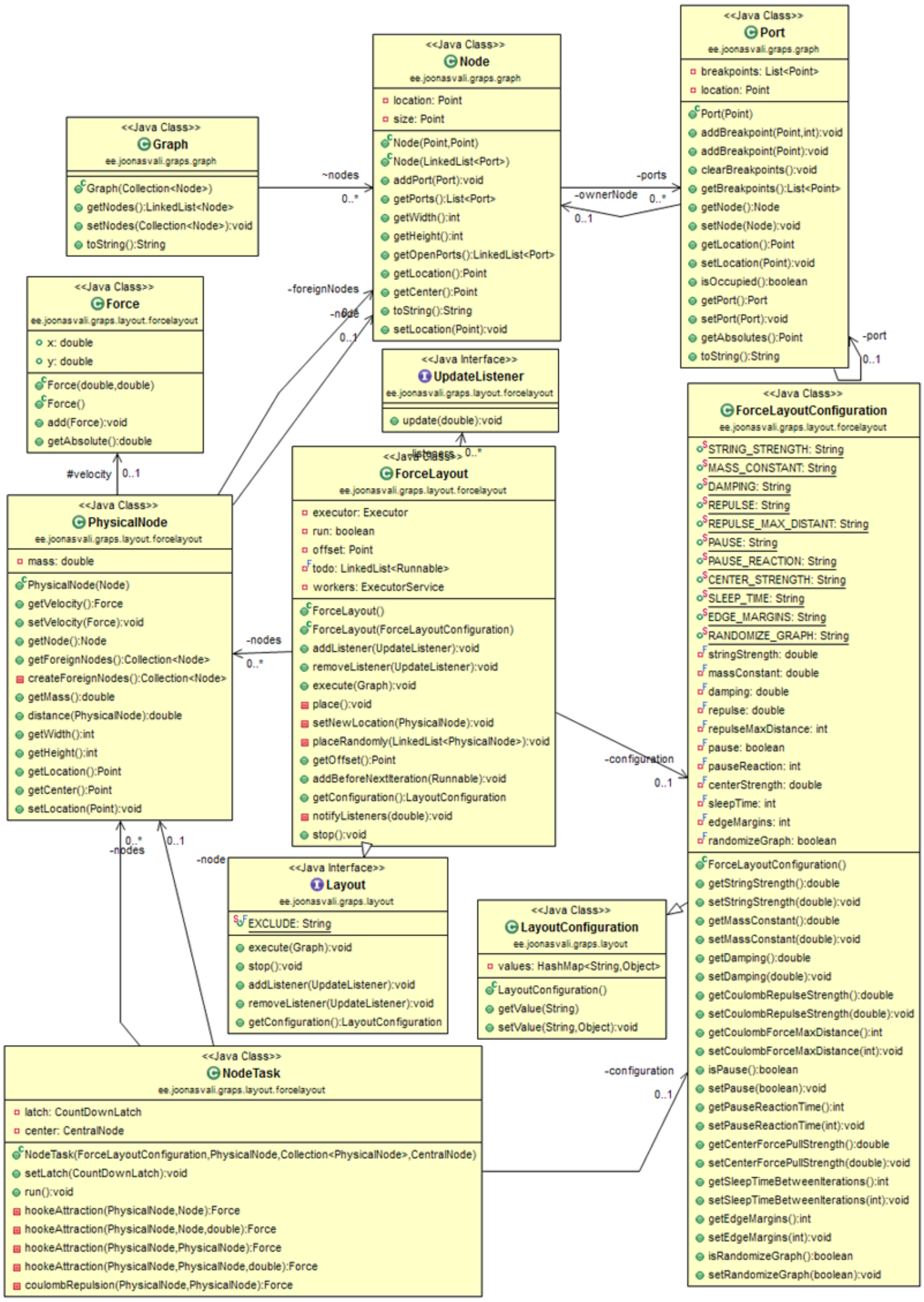


Figure 8: Class Diagram: Placing nodes using Force Layout

- `LayoutConfiguration` Class allows to view and modify the parameters used by the layout algorithm.

- `ForceLayoutConfiguration` class is a control object for the `ForceLayout`, which allows to modify the values used by the `ForceLayout`.

- `NodeTask` class is a reusable Runnable object used by the ForceLayout and generated one per node, to calculate the forces for the node. This is required for multithreaded execution, which means several `NodeTask`s can be executed simultaneously on multiple processor cores.

- `UpdateListener` interface allows to add custom code to `ForceLayout` which method 'update' is intended to be invoked after every iteration. The related methods are also available in the super class, `Layout`, but the actual moment, when the listener is notified, is dependent on the specific implementation.

3.3.2 Breakpoints placement module

Figure 9 describes the classes and their relations involved in breakpoint placing.

- `BreakpointManager` class is the access provider to the breakpoint placement functionality.

- `BreakpointPlacer` interface contains a method that takes a port as an argument and the implementation of this class should place the breakpoints to the connection if needed.

- `PathCalculator` interface has a single method, calculatePath, and its properties are inherited from its factory, based on the specific arguments passed for that specific port, to which breakpoints are going to be calculated.

- `CollisionMap` class creates a two-dimensional boolean array from the provided `Port` and `Graph` object, by creating a local map of the area, from where free path is searched by the `PathCalculator`.

- `StandardPlacer` class is an implementation of `BreakpointPlacer`. It leads the connection out of the node area by placing two breakpoints to calculated positions. Then it creates a `CollisionMap` and uses the provided `PathCalculatorFactory` to generate a `PathCalculator` implementation, which will place the breakpoints between the two nodes, using the generated `CollisionMap`

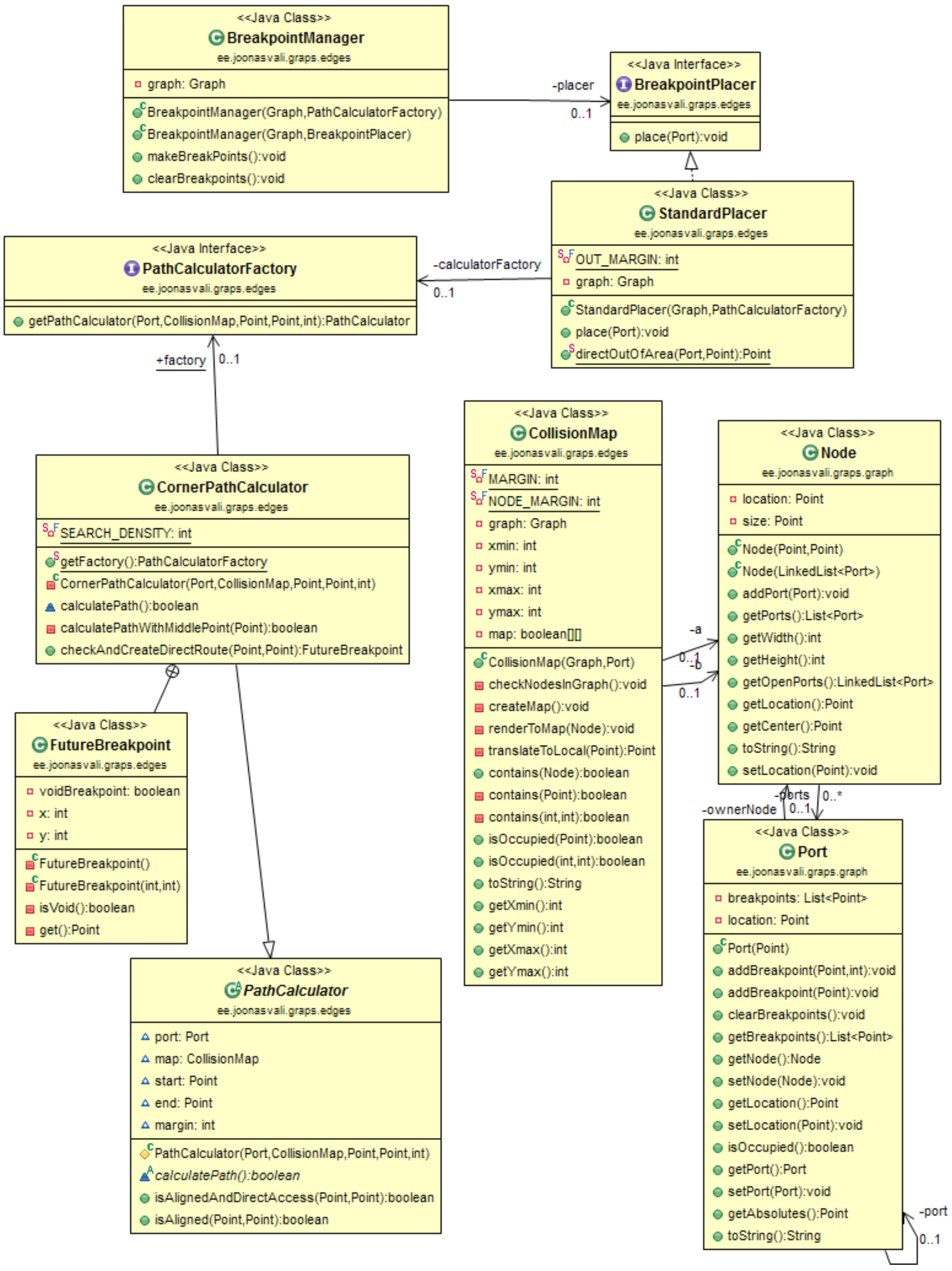


Figure 9: Class Diagram: Placing breakpoints to edges

- `PathCalculatorFactory` interface is used to generate specific `PathCalculator` objects, which are used for calculating breakpoint locations between two ports.
- `CornerPathCalculator` class is an implementation of `PathCalculator` which uses L-shaped search for free paths and places the breakpoint when free route is found. It is not guaranteed that the route is found even if it exists.
- `FutureBreakpoint` is a placeholder breakpoint used by the `CornerPathCalculator`. `FutureBreakpoint` is not immediately applied, but only when the algorithm has been fully executed and a path has been found. This helps to avoid situations when half of the path is found and other half could not be found, because of the dead end.

3.4 Pseudocode

The compact simplified pseudocode of the force based algorithm, excluding multithreading code and supporting methods and classes, could be represented as follows:

```
do while 'run'
  for each node1 in 'nodes'
    create new force
    for each node2 in 'nodes'
      Calculate force hook location for node1 as a
      Calculate force hook location for node2 as b
      define xdifference as a.x - b.x
      define ydifference as a.y - b.y
      define sqrdifference as xdifference * xdifference +
                              ydifference * ydifference
      define massmultiplier as node1.mass * node2.mass *
                               massconstant
      force.x add massmultiplier * coulombconstant *
                  (xdiff / sqrdistance)
      force.y add massmultiplier * coulombconstant *
                  (ydiff / sqrdistance)

    for each connected node
      Calculate force hook location for node1 as a
      Calculate force hook location for node2 as b
      define xdifference as a.x - b.x
      define ydifference as a.y - b.y
      force.x add xdiff * hookeconstant
      force.y add ydiff * hookeconstant

    if center force is enabled
      add attraction force between node and center

    node.velocity().x add netForce.x,
                      result reduced by dampingconstant
    node.velocity().y add netForce.y,
                      result reduced by dampingconstant

  for each node in nodes
    calculate and set new position based on forces and offset

  sleep predefined time before starting next iteration

  calculate offset for next iteration
  notify updatelisteners
```

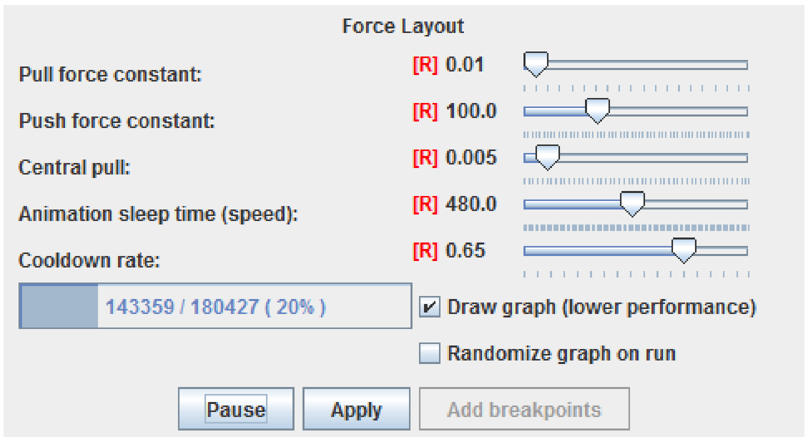
3.5 Integration with CoCoViLa

A dialog was added to CoCoViLa (Figure 10) that enabled to configure some of the constants used by the `ForceLayout`. The dialog allows to start the layout algorithm and keep control the constants during runtime, which are redefined before the next iteration if user decides to apply those. A special hook was added to ForceLayout algorithm which forces the main iterating thread to execute tasks before the next iteration. CoCoViLa uses that hook to apply new configuration settings between the iterations safely.

After every iteration the `ForceLayout` notifies its listeners about the total velocity of the nodes. CoCoViLa adds a listener and uses the provided data to redraw the graph at the right time, if drawing in real time is enabled, and shows the progress of the layout algorithm on the corresponding progress bar.

The main difference between the two hooks between the iterations is that the first hook, `Runnable`, gets executed once and is removed after, while the second hook, `UpdateListener`, is executed every iteration with the new data about the total velocity of the nodes.

To translate the CoCoViLa graph to the graph usable by the `ForceLayout` an adapter was needed, which takes CoCoViLa `ObjectList` and `ConnectionList` as the arguments and composes the corresponding graph, which the layout can use. After every iteration, if the graph drawing is enabled, the graph uses previously composed maps to place the CoCoViLa objects according to the placed graph. If graph drawing is switched off, the graph is only translated back after the algorithm is stopped, saving computation time.


Force Layout

Pull force constant: [R] 0.01

Push force constant: [R] 100.0

Central pull: [R] 0.005

Animation sleep time (speed): [R] 480.0

Cooldown rate: [R] 0.65

143359 / 180427 (20%)

[✓] Draw graph (lower performance)

[ ] Randomize graph on run

Figure 10: Layout control dialog in CoCoViLa

The breakpoints are translated similarly, but due to the absence of iterative properties, the breakpoints get placed, translated and drawn in CoCoViLa sequentially and as a single command to the user.

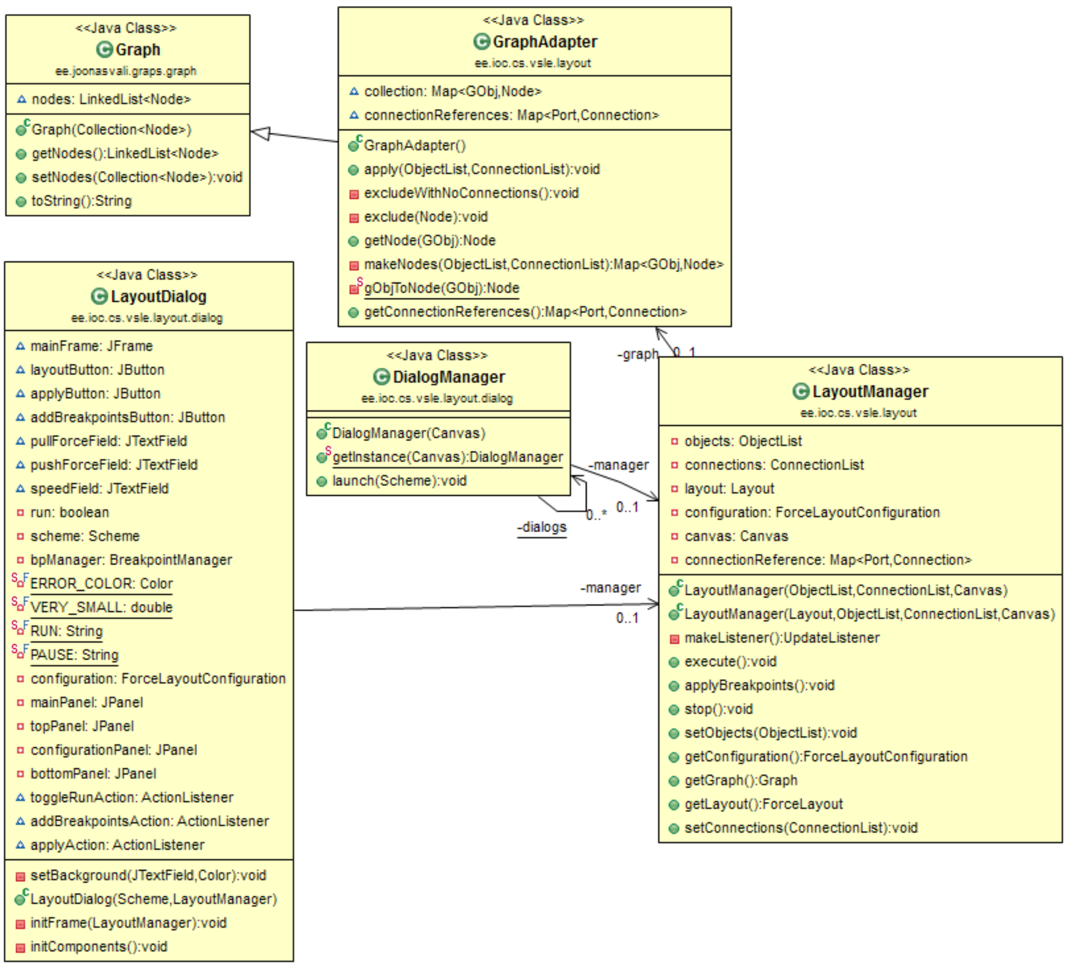
3.5.1 CoCoViLa integration architecture

The class diagram in Figure 11 describes the algorithm's integration for CoCoViLa.

- `LayoutManager` class is the entry point from CoCoViLa to the `Layout` implementation. It takes CoCoViLa graphs and is able to apply any provided layout to those graphs. It also manages the breakpoint placement and translates them as necessary.
- `GraphAdapter` class is used for converting CoCoViLa `Node` and `Connection` objects to a `Graph`, that is usable by the `ForceLayout`.
- `DialogManager` class manages the relations between CoCoViLa canvas and `LayoutManager`.
- `LayoutDialog` is a dialog which is used to control the layout algorithm in CoCoViLa. It connects to the `ForceLayoutConfiguration`.

3.5.2 Source code

The source code for Layout Engine and CoCoViLa integration is available at:
`https://bitbucket.org/jvali/graphs-engine`
`https://github.com/JoonasVali/CoCoViLa/tree/layout`



<<Java Class>>
Graph
ee.joonasvali.graps.graph
nodes: LinkedList<Node>
Graph(Collection<Node>)
getNodes():LinkedList<Node>
setNodes(Collection<Node>):void
toString():String
<<Java Class>>
GraphAdapter
ee.ioc.cs.vsle.layout
collection: Map<GObj,Node>
connectionReferences: Map<Port,Connection>
GraphAdapter()
apply(ObjectList,ConnectionList):void
excludeWithNoConnections():void
exclude(Node):void
getNode(GObj):Node
makeNodes(ObjectList,ConnectionList):Map<GObj,Node>
gObjToNode(GObj):Node
getConnectionReferences():Map<Port,Connection>
<<Java Class>>
LayoutDialog
ee.ioc.cs.vsle.layout.dialog
mainFrame: JFrame
layoutButton: JButton
applyButton: JButton
addBreakpointsButton: JButton
pullForceField: JTextField
pushForceField: JTextField
speedField: JTextField
run: boolean
scheme: Scheme
bpManager: BreakpointManager
ERROR_COLOR: Color
VERY_SMALL: double
RUN: String
PAUSE: String
configuration: ForceLayoutConfiguration
mainPanel: JPanel
topPanel: JPanel
configurationPanel: JPanel
bottomPanel: JPanel
toggleRunAction: ActionListener
addBreakpointsAction: ActionListener
applyAction: ActionListener
setBackground(JTextField,Color):void
LayoutDialog(Scheme,LayoutManager)
initFrame(LayoutManager):void
initComponents():void
<<Java Class>>
DialogManager
ee.ioc.cs.vsle.layout.dialog
DialogManager(Canvas)
getInstance(Canvas):DialogManager
launch(Scheme):void
-graph
0..1
-manager
0..1
0..*
-dialogs
-manager
0..1
<<Java Class>>
LayoutManager
ee.ioc.cs.vsle.layout
objects: ObjectList
connections: ConnectionList
layout: Layout
configuration: ForceLayoutConfiguration
canvas: Canvas
connectionReference: Map<Port,Connection>
LayoutManager(ObjectList,ConnectionList,Canvas)
LayoutManager(Layout,ObjectList,ConnectionList,Canvas)
makeListener():UpdateListener
execute():void
applyBreakpoints():void
stop():void
setObjects(ObjectList):void
getConfiguration():ForceLayoutConfiguration
getGraph():Graph
getLayout():ForceLayout
setConnections(ConnectionList):void

Figure 11: Class Diagram: CoCoViLa Integration

4 Related work

4.1 The Spring Embedder Method - Eades algorithm

The Spring-Embedder method is the earliest viable method for drawing general graphs. The algorithm is often described as a physical system of steel rings connected by electrical springs. The original Eades algorithm did not follow the Hooke's law, but invented his own logarithmic force formula for the edges between the nodes, because the Hooke's law resulted in too strong force between the more distant nodes [13]. $F_1 = c_1 * log(d/c_2)$, where d is the length of the edge, and c_1, c_2 are constants. Every vertice has a repulsive force towards any other vertice. $F_2 = c_3/d^2$. The method proved to be successful up to 50 nodes [15].

4.2 Fruchterman and Reingold algorithm

The Fruchterman-Reingold Algorithm [13] fits for visualising large undirected graphs. It is based on Eades, which in turn evolved from VLSI technique called a force-based placement.

The authors describe the algorithm based on two principles:

1. nodes connected by an edge should be drawn to each other

2. nodes should not be drawn *too* close to each other

Although, additional factors are present, inspired by atom nuclei, being the repulsive force and temperature, making the system lose energy by cooling down.

Three steps are ran for every iteration: calculate attractive forces for every connected pair, calculate the forces of repulsive forces, and, finally, limit the displacement by the temperature.

4.3 Kamanda and Kawai method

Kamanda and Kawai method [15] is based on Hooke's law. It can also be considered as a system of springs. The output quality is similar to the Spring Embedder method, but the advantage of the method is that it takes into account the weights of the nodes.

4.4 Multi-Scale Algorithm - Hadany and Harel algorithm

Hadany and Harel [15] focused on drawing larger graphs. It simplifies the structure of the graphs by introducing *coarse graphs.* The energy minimisation is localised to small neighbourhoods, resulting in relatively fast solution, compared to algorithms simulating full physical systems.

4.5 Harel and Koren

Harel and Koren's [15] *extremely fast* algorithm is motivated by Hadany and Harel algorithm and built around Kamanda-Kawai method. It was designed as a fast algorithm for undirected straight edged graphs. The authors state the algorithm could significally improve the speed of any force-directed method. The algorithm uses approximation, where closely positioned nodes are collapsed into a single vertex. It produces fewer edge crossings than Kamanda-Kawai algorithm and supports drawing graphs containing over 15000 nodes.

4.6 Davidson and Harel algorithm

Davidson and Harel [13] used system energy reduction method. They adapted a method from VLSI, *simulated annealing*, which is computationally costly. The method restates the graph placing problem as an optimization problem by turning it into energy minimization problem.

Simulated annealing requires energy function. Davidson and Harel combined it from vertex distribution, distance from edge and edge-crossings.

4.7 The method of Walshaw

Walshaw's multilevel force-directed algorithm [2] is an extension of Fruchterman and Reingold algorithm, which can deal with huge graphs, over 100 000 nodes, in relatively short time [15].

It places nodes initially randomly, and uses multilevel approach. The idea is to cluster the nodes and simulate the subset of graphs by one level at the time, then by higher hierarchical level, it can consider the previous level graphs and apply the same logic on a level down. The layout isn't applied concurrently on different levels, but rather the graph is refined at each level and the result is extended to the level down.

4.8 Quigley and Eades method

The algorithm is an improvement to the original Spring-embedder algorithm, it features clustering to a quad-tree structure. The forces of distant nodes are approximated and clustered into a single force significally reducing the total amount of forces computed [15].

4.9 Comparison to the implemented force-based algorithm

The implemented force-based algorithm base, like all force-based algorithms, is similar to original Spring-Embedder method, having edges as springs pulling the nodes closer and otherwise pushing the surrounding nodes away. It is remotely influenced by Kamanda-Kawai method, including weights for nodes. The algorithm is also using similar approach to the method of Walshaw, by optionally making the node initial placement random. The algorithm is not multileveled, which could be implemented in the future, to make significant performance improvements, but unlikely to have better visual results.

5 Analysis

5.1 Analysis methods

In this section the produced layouts and breakpoint placements are discussed. The algorithm is applied on several CoCoVila packages, which use different sets of nodes to see how the algorithm manages to place breakpoints in unforeseen situations. Due to the fact that the layout fitness is completely subjective and dependent on the observer, our conclusions are also subjective and the results can not be absolutely or precisely evaluated.

The evaluation of the force layout algorithm performance is complicated for two reasons. First, the properties we have added to the graph compared to the graphs used by many authors who have researched the same problem previously, make the algorithm complex and there are many possible optimizations that can be introduced on algorithmic level or by using some sort of caching for repetitive calculations. The following optimizations were added to the implementation: concurrency and caching of reusable components. Such optimizations are not final and more can be added later, which might make the algorithm perform faster.

Second, the algorithm does not have a "ready" condition, and user action is required to stop the algorithm, due to its iterative properties. The performance evaluation will require a precise moment when the algorithm has finished its job. Very often the algorithm can be terminated before the whole system has stabilised and still have a decent result. The forces are biggest in the first iteration and smallest in the last iteration, thus waiting for the algorithm to "end" would be almost worthless waiting for smaller and smaller changes to happen. Despite that, the end condition for the algorithm could be calculated by measuring average node velocity during one iteration and comparing it to few next iterations to see if progress is big enough to keep the algorithm going or to conclude that the algorithm has finished. In other words, the latter approach requires a subjective view on how small change is small enough to be considered final. Also there might be a certain kind of stagnation or even increasing velocity at certain stages at the beginning of the placing operation, particularly when the graph is in very unstable condition from the force-based layout's point of view and an "explosion" occurs, when the algorithm is launched.

5.2 Performance

For performance testing following computer specification was used:

```
OS: Windows 7 64-bit
Processor: Intel Core i5 2500 Quad Core 4 Threads 3.30 Ghz
RAM: 4x2GB DDR3 1333 DIMM
```

For testing we measure elapsed time in different equilibrium states of the graph: 50%, 75%, 90%, 95% and 97% of maximum recorded. The percentage is calculated from maximum velocity compared to the current velocity of nodes in the graph subtracted from total of 100%, and for that reason is subjective to the initial placement of the graph. The greater the total velocity of the initial placement, the easier it is to reach the goals. In tests the nodes in graphs were initialised in random positions in dynamically calculated area. Also the drawing and translating of the calculated graph to CoCoViLa per iteration is turned off during the performance measuring.

```
Pull constant: 0.01
Push constant: 100.0
Cooldown rate: 0.65
```

All times are shown in milliseconds.

---	888 nodes	1120 nodes	2160 nodes
---	1508 edges	3519 edges	5959 edges
50%	1 986	265	12 407
75%	4 050	5 240	19 931
90%	18 032	20 326	43 470
95%	45 418	77 168	130 718
97%	70 642	138 771	492 395

It is important to notice that the performance results gathered are highly subjective to the graph structure and its initial node positions. Also the completion percentage, at when the times are measured are dependent on the maximum velocity of the nodes in graphs, which is, in turn, dependent on the initial positioning. The results should not be interpreted as absolute, but rather as relative to each other, showing the total computation time growing exponentially as the number of nodes grows. The performance results can be seen as a line chart in Figure 12.

While equilibrium is a sign that a good layout has been found, there is no guarantee, that 95% placed graph is any better than the 50% placed graph. The good layout in reality might not necessarily need such symmetry and mathematical equilibrium to look good and very often does not.

5.3 Visual analysis

5.3.1 Scheme with many ports

When layout is applied to a scheme with many ports and connections on nodes, we get results as seen on Figure 13, page 33. Due to the factor that the ports are located in orderly fashion and are aligned, the resulting breakpoints are often placed on the same line, creating a kind of minimised

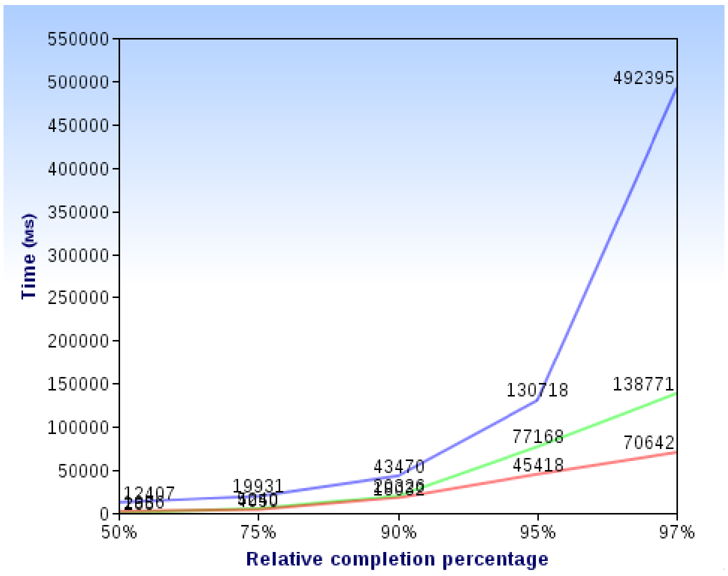


Figure 12: Performance Testing (888, 1120 and 2160 nodes)

layout with connection "highways", where a lot of connections are placed on top of each other, if they share at least one node.

The result might look visually good, but the shortcoming is that just by looking, it might be hard to tell which port connects to which port. Once clicked on, CoCoViLa highlights the connection, enabling to see the actual connections. CoCoViLa allows to connect one port to several other ports, so "reserving" paths for only one connection really does not seem to be an option in this case and it looks like the algorithm solves this problem satisfactorily.

Similar large graph with 1120 nodes can be seen on Figure 15, page 34. It shows some signs of grouping. From the closer view on Figure 14, page 33, we can see that the graph has small density of unplaced edges, which generally does not make the graph less comprehensible, if there are not too many of those.

5.3.2 Scheme with few ports and many connections

In other case, we have a large graph with lots of connections and a few ports on nodes (Figure 16, page 35). The resulting layout is somewhat confusing, but not less confusing than the result of placing the nodes manually. The main reason this particular graph looks confusing when placed is that it is very far from being planar and there are very many connections per port. As a result the neighbouring nodes are often torn apart across the canvas.

The same graph, when examined from parts not in the center, where most connections go through, has some meaningful and visually pleasant

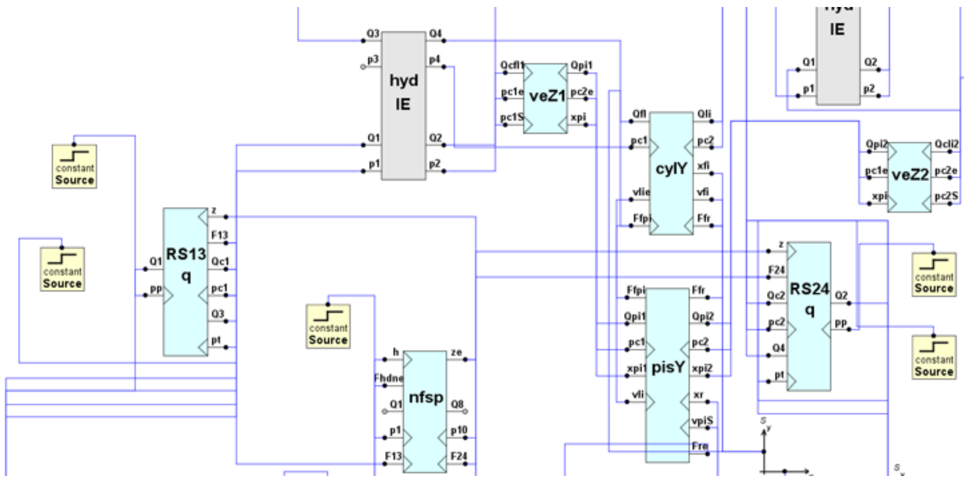


Figure 13: Close view of the placed graph with breakpoints applied (Truncated)

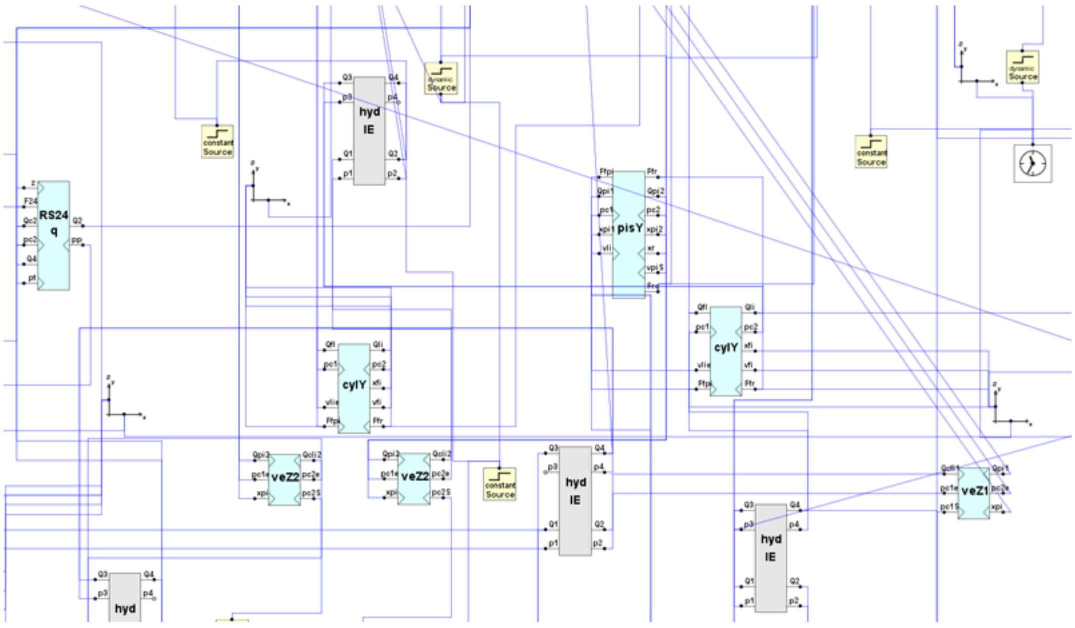


Figure 14: Close view of a large graph with 1120 nodes

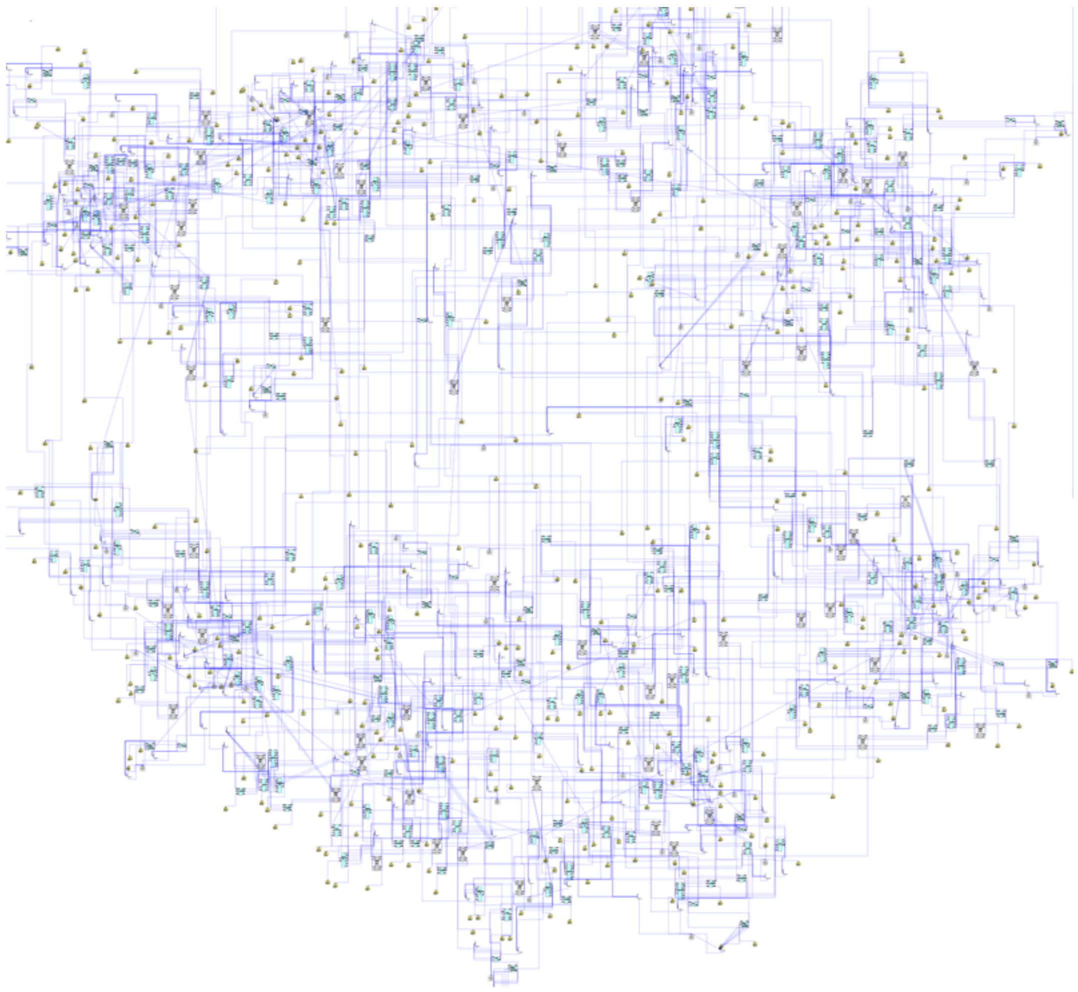


Figure 15: Distant view of a large graph with 1120 nodes

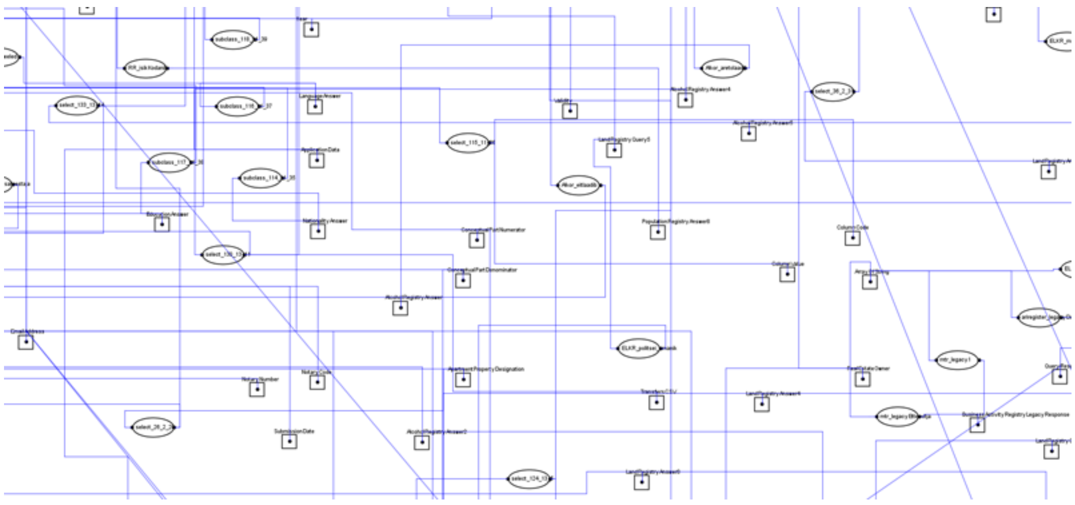


Figure 16: Close view of the placed graph with breakpoints applied (Truncated)

group of nodes placed on the side areas (Figure 17 , page 36).

5.3.3 Satellite groups

From Figure 18 we can see that large graphs create a problem with loosely connected groups. The nodes located in the main part of the graph generate a lot of force towards the small groups and single strings fail to pull those groups back to the main group, pushing the graph too wide and forming satellite groups far from center. The issue can be countered by using the central node, mentioned in section 3.1.2.

Figure 19, page 37, shows an example of a graph with central node. With rest of the parameters the same, it looks more compact, with no satellite groups. A sample close up from the same graph can be seen on Figure 20, page 38

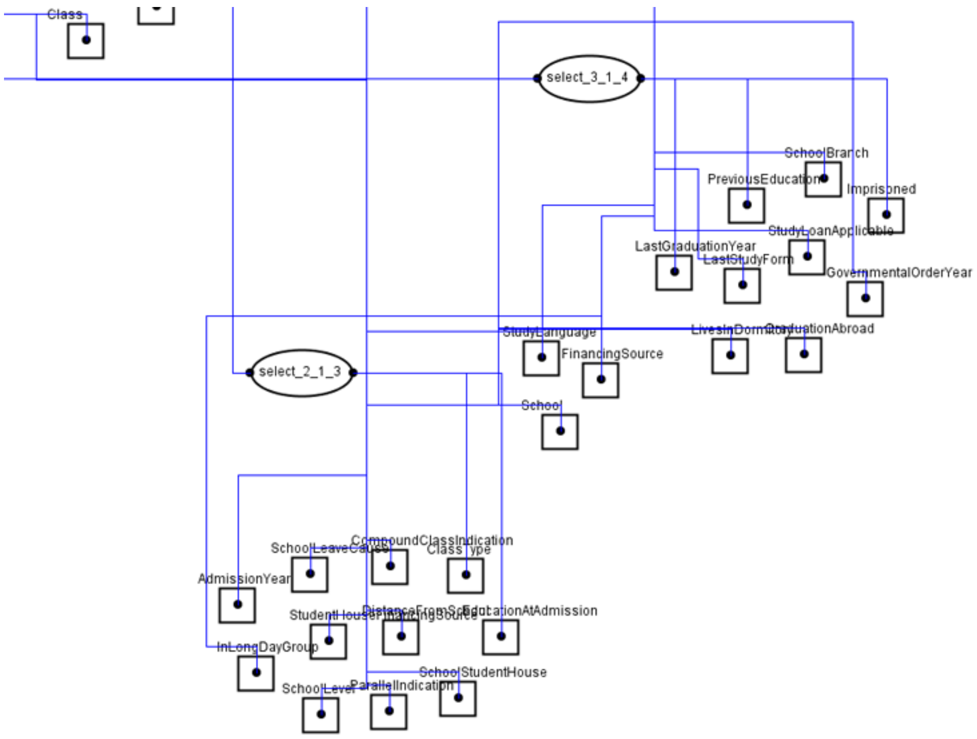


Figure 17: Close view of the placed graph with breakpoints applied (Truncated)

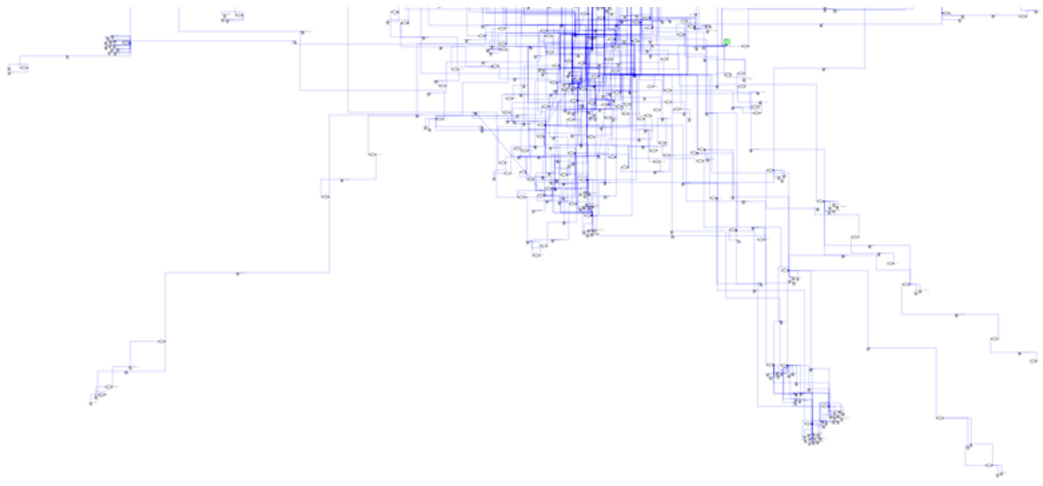


Figure 18: Distant view of satellite groups (Truncated)

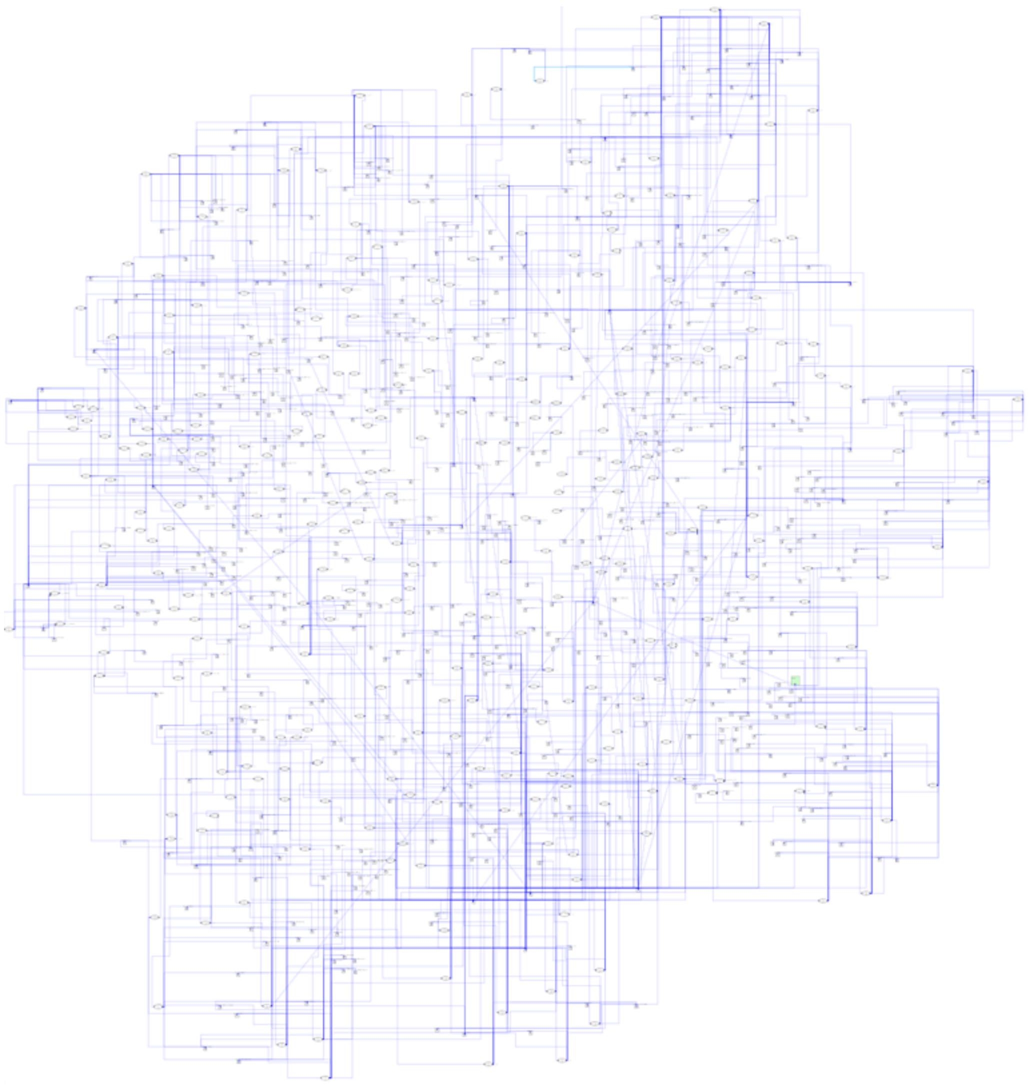


Figure 19: Distant view of a large graph with center node enabled

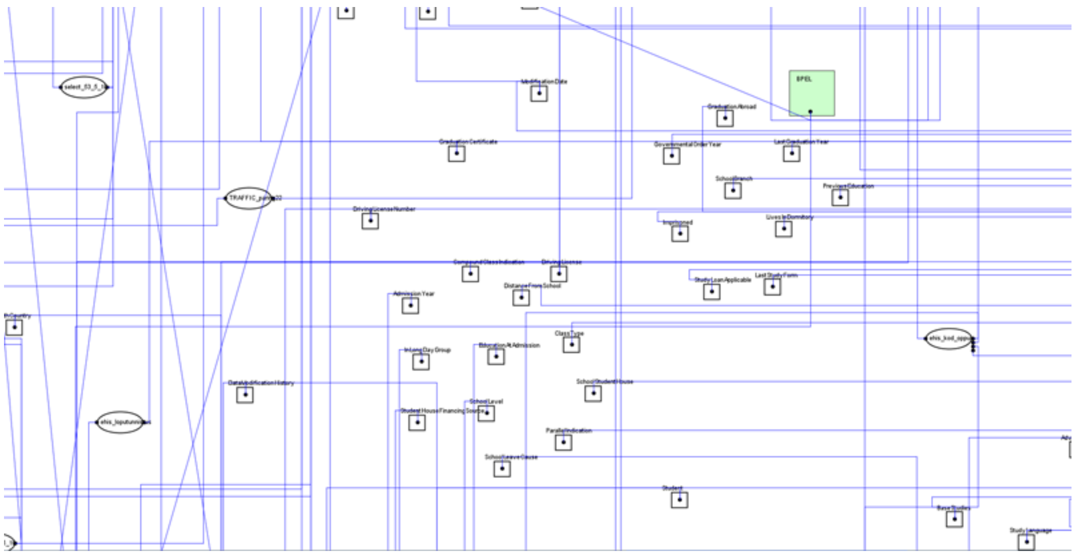


Figure 20: Close view of a large graph with center node enabled

6 Conclusions

The force based layout algorithm produces a convenient layout style for unpredictably composed graphs. The layout is able to adapt to any new situation, programmer never might have foreseen. The layout works best for small and medium sized graphs. For large graphs some user help is needed to find the equilibrium of the forces in the graph.

On large-scaled graphs the algorithm works well, if the graph is nearly or completely planar. Non-planar graphs may sometimes force the neighbouring nodes far away from each other, creating problems with placing edges later, which tend to look crowded if this situation is present throughout the graph.

The layout algorithm can be used with different domains, optionally with a little configuration assistance from the user. In case there are many edges per ports, the algorithm manages to overlap some of the edges, and making the graph look less crowded to the observer. This comes with a cost with the graph being more confusing if single edges are intended to be visually distinct.

Graphs with a lot of nodes tend to divide into several groups if it is possible. Grouping in the layout happens when two or more groups of nodes are connected by relatively few edges to each other. The forces between the two groups then force the nodes to float apart, while the single or few edges try to pull those together.

Optional central node helps to keep unconnected parts of graphs and groups together and forces the satellite groups back to the main graph. The central node might make the graph look more crowded, which can be balanced by introducing bigger coulomb force.

In most cases the simple L-shaped pattern for placing breakpoints to edges is sufficient, but in case the graph is more crowded, the algorithm fails more often to find a way. User may counter this, by configuring the algorithm so that there would be left more room between the nodes.

Due to the fact that the every node is required to iterate over every other node, when calculating forces, the complexity of a single iteration for the force-based algorithm can be roughly evaluated to $O(n^2)$. The performance analysis done confirms that multiplying the number of nodes is exponentially reflected in the time of achieving equilibrium in the graph.

For some domains, the force-based layout and the accompanying breakpoint placement fits better than for the others. The domains that have very few ports for nodes with many connections tend to look more confusing than the ones with many ports and many connections.

In conclusion, the implemented force-based layout in CoCoViLa can be used with a limited success on large graphs, and some configuration is needed. The produced layout is comprehensible in most cases, and usually configuring the layout to have smaller pulling forces between edges, constituting better results in cases where too many edges are otherwise left without breakpoints and nodes are too close to each other.

6.1 Future work

The engine has been designed to be easily extendable and in the future it is possible to add additional layouts and edge positioning algorithms. In CoCoViLa, a valuable extension could be the possibility to equip visual language packages with custom layout managers and layout configurations specific to domains of packages. This would save unexperienced users from manual effort to choose and setup the layout.

7 Summary

Graph drawing helps humans to understand the data presented in the graph form better to detect anomalies and patterns.

CoCoViLa allows users to create and manipulate schemes in the visual manner and good placement is required for the graphs to look meaningful and reflect the data present at visual observation.

The goal of the work was to research a problem of automatic layout on large-scale graphs exceeding 1000 nodes, investigate and implement and to implement a convenient layout algorithm for the purpose to use it in the CoCoViLa environment.

In Section 2, *Force-based algorithms*, force-based layouts were described with their application areas. Force-based algorithms treat nodes as spring connected objects, hold together by forces remotely or directly based on Hooke's Law. The purpose of springs is to position the nodes of a graph in two-dimensional or three-dimensional space so that all the edges are of more or less equal length and there are as few crossing edges as possible. The entire graph is then simulated as if it were a physical system. force-directed algorithms are in general considered to have a $O(n^3)$ running time. In case of a planar graph or a nearly planar graph, a force-based algorithm will provide a high quality layout, further improved by appropriate edge placing algorithm. There are no or very few overlapping edges, depending on the locations of gates on the nodes.

In Section 3, *Implementation*, the details of the implemented algorithm and related work done previously on this area were described. The implementation was written kept in mind the algorithms could be used on many different domains, and thus any special information about the graph is not revealed to the layout. The information revealed is the size of the node, the location of the ports on specific node, and the connections between the nodes without any specific limitations or predefined information.

To solve a problem of uniformly distributed force around a node, which might not necessarily be square shaped, force vectors were redirected to the edge of the node, as seen on Figure 4, page 15.

An invisible central node was introduced, which connects to every other node, to keep unconnected parts of the graphs together and the graph compact, described more in detail on page 15.

To connect two unaligned nodes with non-diagonal edges, there is a need for a breakpoint to connect the different parts of the edge. Breakpoints act as invisible nodes connecting two edges for a turn.

A map of local elements was created between two selected nodes, to compute the locations of breakpoints more efficiently. The map can be seen on Figure 7, page 18

An simple yet effective algorithm was developed to place the breakpoints, by detecting L-shaped vacant routes on the computed local map. The algorithm is based on trial and error method and the calculated breakpoints are often a result of combination of two L-shaped paths. The algorithm can be improved further to calculate unlimited amount of combinations for the breakpoints iteratively, always succeeding, if path exists.

To integrate the developed force-based layout to CoCoViLa, several code hooks were introduced for injecting custom code to the graph placing process, described more in detail on page 24. A dialog was designed to control the layout configuration constants and the layout running process, as seen on Figure 10, page 25.

Related work details in this area were described in *Related work*, page 27.

In Section 5, *Analysis*, relative performance test results of the layout algorithm were published, which were done by measuring the time of graph achieving certain percentage of equilibrium, which was detected by comparing maximum velocity of the nodes to the current velocity. The results indicated that the time which was needed to achieve a specified level of equilibrium in the graph, grew exponentially when number of nodes was nearly doubled. It was also noted, that equilibrium does not necessarily equal to the visually good looking layout.

Due to the fact that the every node is required to iterate over every other node, when calculating forces, the complexity of a single iteration for the force-based algorithm can be roughly evaluated to $O(n^2)$. The performance analysis done confirms that multiplying the number of nodes is exponentially reflected in the time of achieving equilibrium in the graph.

A visual examination of the graphs showed that certain problems arise placing large-scaled graphs and often calibration is needed by the user, to produce the visually good looking results. Some domains were not placed as successfully as others, because the nodes had large amount of edges per port, and as large amount of neighbouring nodes, which create difficult looking layouts, if the node is not actually placed anywhere near the neighbouring node.

It was concluded that on large-scaled graphs the algorithm works well, if the graph is nearly or completely planar. Non-planar graphs may sometimes force the neighbouring nodes far away from each other, creating problems with placing edges later, which tend to look crowded if this situation is present throughout the graph.

The implemented force-based layout in CoCoViLa can be used with a limited success on large graphs, and some configuration is needed. The produced layout is comprehensible in most cases, and usually configuring the layout to have smaller pulling forces between edges, gives better results in cases where too many edges are otherwise left without breakpoints and nodes are too close to each other.

References

[1] http://www.cs.ioc.ee/cocovila/, "CoCoViLa — Model-Based Software Development Platform." www.cs.ioc.ee/cocovila/. [Online; accessed 13-January-2013].

[2] C. Walshaw, "A multilevel algorithm for force-directed graph-drawing.," *Journal of Graph Algorithms and Applications*, vol. 7, no. 3, pp. 253–285, 2003.

[3] T. Dwyer, B. Lee, D. Fisher, K. I. Quinn, P. Isenberg, G. Robertson, and C. North, "A comparison of user-generated and automatic graph layouts," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 15, no. 6, pp. 961–968, 2009.

[4] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Computational Geometry*, vol. 4, no. 5, pp. 235–282, 1994.

[5] D. Easley and J. Kleinberg, "Networks, crowds, and markets: Reasoning about a highly connected world," *Cambridge Univ Pr*, 2010.

[6] Wikipedia, "Graphs." http://en.wikipedia.org/wiki/Graph_%28mathematics%29. [Online; accessed 27-January-2013].

[7] Wikipedia, "Planar graph.." http://en.wikipedia.org/wiki/Planar_graph. [Online; accessed 03-February-2013].

[8] F. Greco, *Traveling Salesman Problem.* InTech, 2008.

[9] S. G. Kobourov, "Spring embedders and force directed graph drawing algorithms," *CoRR*, vol. abs/1201.3011, 2012.

[10] Wikipedia, "Force-based algorithms (graph drawing) — wikipedia, the free encyclopedia." http://en.wikipedia.org/w/index.php?title=Force-based_algorithms_(graph_drawing)&oldid=520004468, 2012. [Online; accessed 17-November-2012].

[11] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph drawing: algorithms for the visualization of graphs.* Prentice Hall PTR, 1998.

[12] Y. Hu, "Efficient, high-quality force-directed graph drawing," *Mathematica Journal*, vol. 10, no. 1, pp. 37–71, 2005.

[13] 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.

[14] yWorks, “Organic layout style - yworks, automatic graph layout.” `http://docs.yworks.com/yfiles/doc/developers-guide/smart_organic_layouter.html`, 2011. [Online; accessed 17-November-2012].

[15] D. Harel and Y. Koren, “A fast multi-scale method for drawing large graphs,” in *Graph drawing*, pp. 183–196, Springer, 2001.