MASTER'S THESIS 2024

Force Directed Drawing Algorithms and Parameter Optimisation

Isak Nilsson

ISSN 1650-2884
LU-CS-EX: 2024-25

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2024-25

Force Directed Drawing Algorithms and Parameter Optimisation

Kraftbaserade Ritningsalgoritmer och Parameteroptimisering

Isak Nilsson

Force Directed Drawing Algorithms and Parameter Optimisation

Isak Nilsson
isak.nilsson.skola@gmail.com

June 6, 2024

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

Supervisors: Jens Oknelid, jens.oknelid@neo4j.com
Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se

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

Abstract

Through my thesis I have implemented and compared some different graph drawing algorithms in addition to some methods to speed up the slow parts of these algorithms. These algorithms were then used to test what to the best of my knowledge is a novel approach to select parameter values for graph drawing algorithms. For this, I use methods similar to those used in Machine Learning to select parameter values and measure the utility of any set of parameters by creating my own utility function. I created this function by looking at objective measures of drawing quality that are commonly known, such as the number of edge crossings, along with the time it took to draw a given graph. The resulting method for parameter optimisation could find significant increases in the speed of graph drawing for several of my implemented drawing algorithms without compromising drawing quality. Furthermore, the approach is not specific to any parameter set, and can with some modification be applied to any graph drawing algorithm dependent on some constants.

Keywords: graph, drawing, parameter, optimisation

Acknowledgements

Thanks go out to my colleagues at Neo4j, specifically my supervisor Jens Oknelid who provided me with consistent feedback and was always open to new ideas, as well as my supervisor Jonas Skeppstedt at Lund University who was easy to contact and provided good guidance for any concerns. I also got some good advice and suggestions from my friend Niklas Simandi and my brother Jonathan Nilsson.

Contents

Chapter 1
Introduction

1.1 Background

Graphs are mathematical structures consisting of nodes, also known as vertices, where some pairs of nodes are connected by edges. Graph drawing is a field combining mathematics and computer science to create aesthetically pleasing and informative images of mathematical graphs. Since the positions of nodes are not given in the structure a graph can be drawn in many ways and it is therefore up to the drawer to position them in a good way. A common approach for drawing a graph is using what is known within the graph drawing community as force-directed algorithms, where nodes are simulated as particles acting with forces on one another. Nodes are then moved in a loop of steps based on forces calculated at each step. For large graphs, these forces can become quite time-consuming to calculate and it is hence common to simplify the number of calculations with various approaches. Fortunately, this approach has quite a large overlap with simulating physical bodies, what is known as n-body simulation, and one can thus borrow algorithms from this field.

An interesting aspect of optimising these algorithms stems from the fact that one tries to optimise the time until stability has been reached, in combination with the quality of the graph drawn. This means that even if an approach drastically decreases the time to stabilise, for example massively increasing the electric charge of nodes (causing all disconnected nodes to spread out very far from each other), it might still reduce the quality of the drawing to such a level that it is unacceptable. One also needs to consider the difference between speeding up a single cycle of calculations in comparison to speeding up the total time to stabilise. Sometimes one can speed up the stabilisation by making quicker, less thorough, calculations, but other times the decrease in quality of calculations is so large that the time to stabilise actually increases even if each cycle is quicker.

This thesis was written together with Neo4j. I have therefore implemented my algorithms in Typescript in order to be able to use them in tandem with their existing drawing software Bloom.

1.2 Research questions

This thesis focused on how to optimise graph drawing. The main aim is summarised by the following research questions:

- How do different known algorithms for graph drawing compare to each other?
- Can one create a programmatic evaluation function of a graph drawing that will correspond to the beauty or quality evaluation of the user?
- Is it possible to use a programmatic evaluation function to optimise the parameters of a given algorithm for drawing a graph?
- Is it possible to create a generalised programmatic parameter selection to improve the speed and quality of drawing for any graph?

In chapters 3, 4, 5 and 6 I will go through the algorithms and methods used to answer the research questions. Later in chapter 7 I will present my answers to them, with some further discussion in chapter 8.

1.3 Related Work

1.3.1 Dot's algorithm

Graphviz is a very popular graph visualisation software. One approach they utilise is known as "dot's algorithm". This algorithm is used to draw graphs in the plane, but with the addition that it takes edge direction into consideration. Unlike the algorithms I have implemented, this one tries to aim edges such that they point in the same direction. In addition, this algorithm also allows for curved edges. Quickly summarised each iteration of this algorithm goes through 4 steps, the first ranking each node, the second using a heuristic to reduce edge crossings by setting vertex order based on rank, the third finding actual coordinates by constructing an auxiliary graph and the final fourth step drawing edges. The full paper describes the algorithm in detail [5]. Since Neo4j uses force-directed algorithms, I have focused especially at these, in order to find ways Neo4j's Bloom can be improved.

1.4 Spring-based

One of the first suggestions for force-directed layout algorithms was one proposed in the paper "An algorithm for drawing general undirected graphs" [7]. In this paper they propose a system where each node in a connected subgraph is connected to all other nodes in the graph by a spring with a desired length proportional to the "graph theoretic" distance between the nodes, meaning the number of steps in the shortest path between the nodes. In this way, both global separation and local connection over edges is maintained using a single method. The approach is rather simple, but a good starting point when just getting familiar with graph drawing, and shows some of the strength of force-directed algorithms, such as the symmetric nature of images that is often achieved. The algorithm for finding the graph theoretic distance

between all pairs of nodes is however $\mathbf{O}(n^2)$, where n is the number of nodes, and thus quite slow.

1.5 Electric- and Spring-based

Another common approach, which is also the one used by Neo4j, is one that utilises electric forces as well, keeping springs only between nodes that are connected by an edge and separating all nodes by simulating an electric repulsion between them. In this way, local separation is maintained by springs while global separation is maintained by the electric force. This algorithm is also $\mathbf{O}(n^2)$ for the electric calculations but some improvements can be made to speed up the algorithm, which we will look at in the next chapter. This algorithm also has the advantage that it works for graphs consisting of multiple disconnected subgraphs without any addition, while the fully spring based algorithm would need some kind of complement to separate the disconnected components. An example of utilizing this approach can be found in the paper "Graph Drawing by Force-directed Placement" [6].

1.6 Electric- and Constraint-based

Another algorithm I have looked at used a constraint-enforcing algorithm instead of forces to maintain the local separation for the edges. This approach was proposed in the paper "Towards Visualizing Big Data with Large-Scale Edge Constraint Graph Drawing" [4]. The algorithm works by setting some constraint on how far nodes that are connected with an edge can be apart, with some error tolerance. As long as the distance between the nodes is outside an error factor from the desired distance, the nodes are moved towards satisfying this constraint. One clear advantage of this algorithm is that the constraint satisfaction will never overshoot the target distance unlike a spring could. For pseudocode of my implementation of this see figure 3.4.

Chapter 2
Methodology

At Neo4j the graph drawing algorithm is used for a multitude of different graphs, and we would like to find an algorithm that performs well for all of them, both in terms of time and in terms of drawing quality. A drawing of high quality should show the structure of the graph in a simple way. Time becomes an issue for large graphs, and in these cases we would like to optimise our algorithm so that it runs quickly. At the same time it is quite important that the algorithm draws smaller graphs in an illustrative way, not for example with certain nodes being spread out too far from the rest. Since a lot of work has gone into implementing the current algorithm, we are looking primarily at algorithms that are similar to the electric- and spring-based algorithm currently used, but with different modifications.

The work was split into a few steps. Firstly we needed to get some different drawing approaches. Initially we looked for different papers detailing graph drawing algorithms, focusing on those that used force-directed approaches. After finding some promising papers, I implemented the algorithms from these. In addition I implemented some different techniques to optimise the drawings. During this process parameters were chosen by hand. When the algorithms were implemented, I started work on automatic parameter selection, in order to further improve the overall quality of the drawing algorithms, including the efficiency. After implementation and parameter selection was complete, algorithms were compared against each other in terms of time to draw and subjective drawing quality, to look for possible improvements of Neo4j's algorithm.

Chapter 3
Drawing Algorithms

In this chapter I will describe my implementations of the fully spring-based, the electric- and spring-based and the electric- and constraint-based algorithms mentioned in chapter 1. I will also describe my implementation of a new algorithm proposed by my supervisor at Neo4j where subcomponents of the graph are drawn separately.

Our algorithms for drawing graphs all have a common goal. We want to illustrate the structure of the graph, by keeping all nodes apart to some degree, while still keeping nodes that are connected close together. Often [4] [6], this is done by splitting the algorithm into two parts, one local separation, working to keep nodes connected with edges at a proper distance from each other, and one global separation, maintaining distance between all nodes.

Each algorithm is based on a loop, where nodes are moved in a cycle of iterations until a stabilisation condition is met. Force-directed algorithms, where some forces are calculated whereafter the nodes are pushed in accordance with the current speed they have. This is done using what is called Verlet integration [8], where speed is approximated based on the previous position. This method of approximating speed proved preferable over what is known as the Euler method, where speed is kept as a separate vector, computed based on previous speed and current acceleration. Running Bloom with both of these methods, we found cases where the Euler method would fail to stabilize, while Verlet integration allowed for a static state to arise. The advantage lies in the fact that numerical errors can not cause the speed and the positions to go out of sync [4]. The method to push nodes of a graph G(V,E), with vertex set V and edge set E, is described in figure 3.1 where $\boldsymbol{F_V}$ is the force acting on node v, $\boldsymbol{T}$ is a temperature factor in $\mathbf{[0, 1]}$ that causes stability as it is cooled down, and ξ is a damping factor on the speed, also in $\mathbf{[0, 1]}$. As the name suggests coolingFactor is a factor for reducing the temperature over time. G(V,E) will be used to symbolise the graph in other figures as well, with G(V) meaning the set of vertices and G(E) the set of edges.

Listing 3.1: Pseudocode describing the method used in all my algorithms for pushing nodes based off forces.

```
for v in G(V) do
  v_{t+1} = (v_t + T * (ξ * (v_t - v_{t-1}) + F_v * dt^2)
end for
T = T * coolingFactor
```

My implementation for the fully spring based algorithm can be seen in listing 3.2, where k is the spring constant and L is the desired distance between two neighboring nodes in the graph. BFS is here a standard breadth-first search to find the shortest graph distance to all connected nodes from the input node. $dist_v(u)$ is the graph theoretic distance between nodes v and u.

Listing 3.2: Pseudocode describing the spring-based algorithm.

```
for v in G(V) do
  dist_v = BFS(v)
end for
for e = (v1,v2) in G(E) do
  if v1 \neq v2 then
    d = v1 - v2
    k = k_0 * d
    F = k * (L * dist_v(v2) - d) * d / d
    F_v1 = F_v1 + F
  end if
end for
pushNodes()
```

Pseudocode describing the electric and spring-based algorithm is shown in listing3.3, where k is the fspring constant, r the spring's rest length, c_k the electric force constant and q the charge of particles.

Listing 3.3: Pseudocode describing the electric and spring-based algorithm.

```
while notStable() do
  for e in G(E) do
    v1 = e(1)
    if v1 ≠ v2 then
      d = v1 - v2
      F = k * (r - d) * d / ||d||
      F_v1 = F_v1 + F
    end if
  end for
  for v1 in G(V) do
    for v2 in G(V) do
      if v1 ≠ v2 then
        d = v1 - v2
        F = c_K * q^2 / ||d||^2 * d / ||d||
        F_v1 = F_v1 + F
      end if
    end for
  end for
  pushNodes()
end while
```

Pseudocode describing the electric and constraint-based algorithm is displayed in listing 3.4, where r is the desired distance between nodes and e the error tolerance. Γ is the constraint-precision, defining how hard we should enforce the constraints in each loop. c_k the and q are the same as in Listing 3.3.

Listing 3.4: Pseudocode describing the electric and constraint-based algorithm

```
while notStable() do
  for v in G(V) do
    for u in G(V) do
      if v ≠ u then
        d = v - u
        F = c_K * q² / ||d||² * d / ||d||
        F_v = F_v + F
      end if
    end for
  end for
  pushNodes()
  for i = 0; i < Γ; i++ do
    for e = (v1,v2) in G(E) do
      if v1 ≠ v2 then
        d = v1 - v2
        if ||d|| > r or ||d|| < e * r then
          D = 1 - r / d
          if ||d|| < r * e then
            D = 1 - r * e / d
          end if
          v1 = v1 + 1/2 * D * T * d
          v2 = v2 - 1/2 * D * T * d
        end if
      end if
    end for
  end for
end while
```

3.1 Sublayout-based

The final algorithm I have implemented uses the fact that disconnected components can be seen as a graph of their own. Drawing graphs using this concept was suggested to me by my supervisor at Neo4j. One could then draw each disconnected component independently, create a "supernode" to represent the subgraph and try to lay out the supernodes in a pleasing pattern. The total node positions can then be acquired by summing up the position of the supernode and the position of the node inside the sublayout. For pseudocode describing my algorithm for this, see listing 3.5. I have decided to try to connect the supernodes with edges in a grid pattern as in figure 3.1, where the desired length of an edge is determined by the radius of the two layouts it connects. I chose for the width of the grid to be $\lceil \sqrt{numSuperNodes} \rceil$ so that we would get a perfect square when we have a square number of nodes. I also set the charge of each supernode as the sum of the number of nodes inside. Unlike the other algorithms where I bootstrap (pick a good initial value for) centres for connected nodes randomly I have chosen to also bootstrap the starting positions in the grid pattern described in figure 3.1. Since this layout method allows us to ignore other subcomponents when calculating a sublayout, it could reduce the total number of calculations for the electric forces.

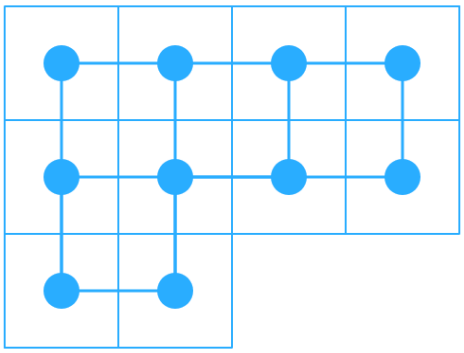


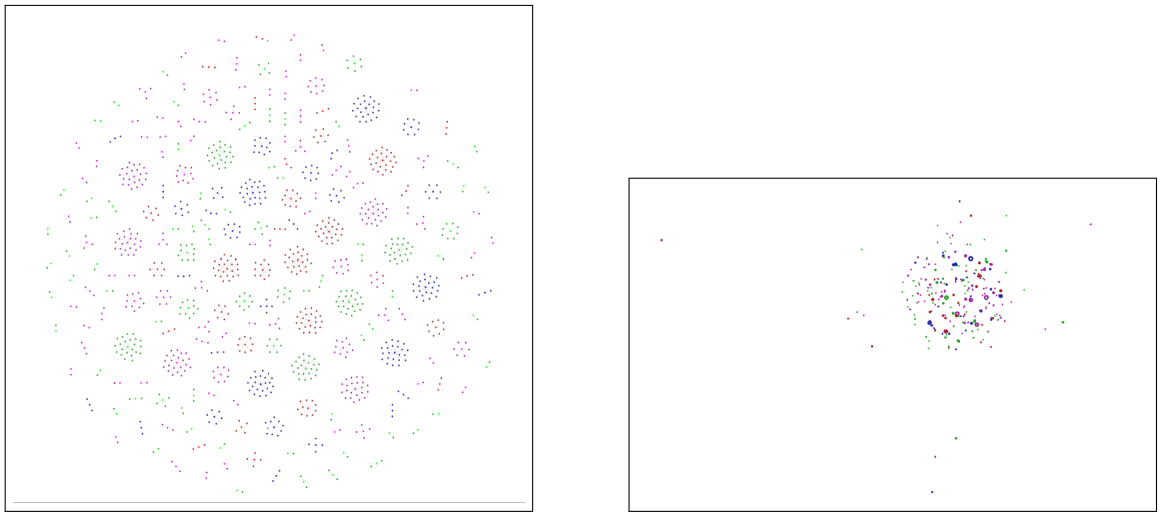
Figure 3.1: A figure illustrating how supernodes are placed and connected in a grid when setting up the sublayout-based algorithm.

Listing 3.5: Pseudocode describing the sublayout-based algorithm.

```
Gs = [G_1, G_2, ...] = BFS()
constraints = [0,0,...]
superNodes = initSuperNodes()
while notStable() do
  for (i = 0; i < Gs.length; i++) do
    G_i = Gs[i]
    G_i.layout()
    constraints[i] = largestCoord(G_i)
    for v in G_i do
      G(v)
    end for
  end for
  for e in G(E) do
    v1 = e(1)
    if v1 ≠ v2 then
      d = v1 - v2
      F = k * (r - ||d||) * d / ||d||
      F_v1 = F_v1 + F
    end if
  end for
  pushNodes()
end while
```

3.2 Common algorithm improvements

Certain techniques can be used across multiple algorithms. Through the work I experimented and found some common things that can be implemented across the different drawing methods that can have a large impact on the speed or quality of the final drawing. Most significant of these is the use of a ceiling on forces. The use of a ceiling helps prevent very large single pushes when nodes are close together, that could otherwise push nodes out in a way that



(a) (b)

Figure 3.2: A comparison of drawing a graph consisting of many small subcomponents with and without a ceiling on the forces acting on nodes. Both drawing algorithms are in other parts identical.

drastically increases the drawing area. An easy way to do this would be to simply let the force become some maximum value if it becomes larger than the ceiling. Another approach I tried was to use a logarithm of the force after the unaltered force becomes larger than the logarithm of it. By altering the base of the logarithm, one can set when this occurs, and create a sort of "soft ceiling" for the force, such that it can grow indefinitely still, but far slower after it hits the soft ceiling. When comparing this logarithm-based ceiling with just a set maximum, I found no large differences, but using either one of the two proved very important for drawing quality. A comparison between using no ceiling and using the log-based ceiling can be seen in figure 3.2. We can note that the graph in figure 3.2b becomes much harder to analyse in total since certain nodes get pushed needlessly far away from the rest. In order to not have to zoom out too much the drawing without a ceiling had to be cropped to only a part of it, but there were certain nodes that were around twice as far from the centre as the furthest visible.

Another important thing was the starting positions of nodes. For most papers I have read these are initialised randomly. In graphs with large disconnected subcomponents, I have however found that randomly placing each node in a grid before starting the physics simulation will often lead to a drawing where disconnected subgraphs are tangled up in each other. My solution for this is to bootstrap initial positions by finding each connected subcomponent using a BFS at the beginning of the drawing and randomly generating a centre point in a large grid and then generating all nodes of the subcomponent in a smaller grid around this centre. Since we will only need to visit each node once this search is a quick $\mathbf{O(n)}$ operation that should not slow down drawing. In figure 3.3 you can see a comparison between the stable drawings when using this bootstrapping and not for a graph consisting of some rather large subcomponents. In the figure on the left where bootstrapping was used, the three subgraphs are clearly separated, while the other drawing is too entangled to show the number of disconnected components.

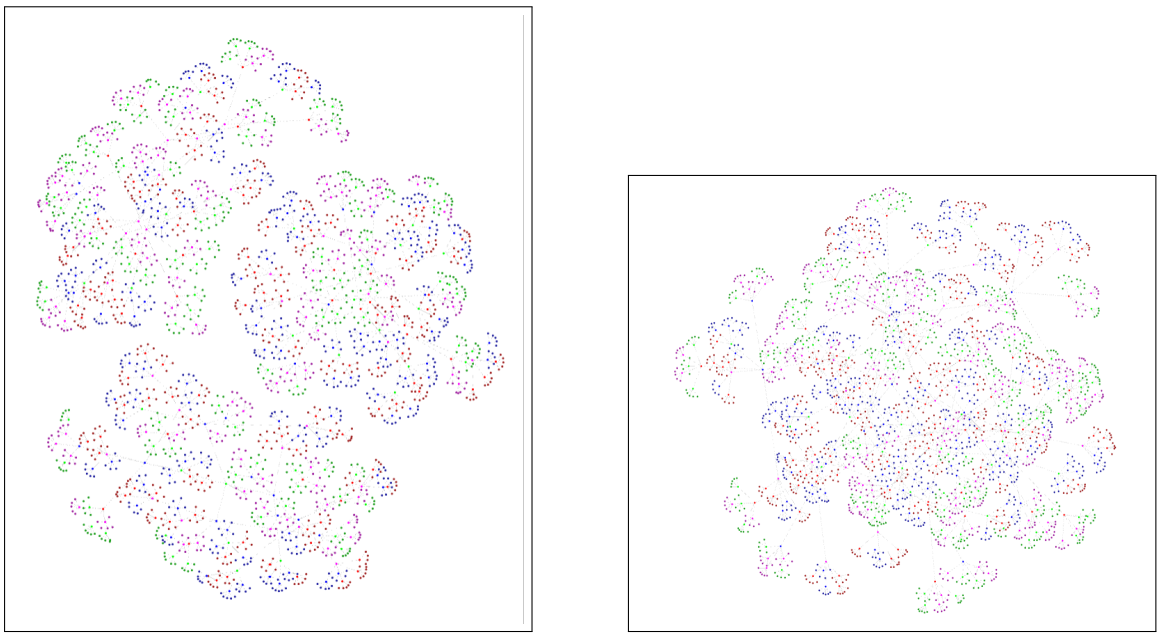


Figure 3.3: A comparison of drawing a graph consisting of three rather large disconnected subgraphs with and without bootstrapping. Both drawing algorithms are in other parts identical.

Chapter 4
Techniques for improving running time

Graphs stored in databases such as neo4j are often very large, which can lead to slow performance when drawing. Fortunately, it is often possible to speed up the slow part of drawing algorithms. My implementations of some of these methods will be discussed in more detail here.

4.1 Grid-based

A rather simple approach for speeding up the electric calculations is to split the drawing space into a grid and then store nodes in a data structure like a matrix. If the grid is recreated each cycle, which can be done in **O(n)** time, then one can choose the grid size to fit all nodes each iteration. Another approach, which in practice proved a bit quicker, is to create an initial grid, moving nodes inside this, and only recreate the grid with a larger size if some node is moved outside. By storing nodes in cells corresponding to their location in the grid one can determine which nodes to compare to based on their location in the matrix. When calculating the electric force acting on any node within a cell one can then cut down the calculations to the ones from nodes within a certain proximity. My implementation of this algorithm can be seen in figure 4.1. The method gridNeighs() that returns neighbors of v I takes neighbors within a certain circular radius. For the speed of a single calculation, it would obviously be better to have a closer proximity - however for the correctness of the computation the opposite would be true. Since the force fades based on distance it also makes sense to choose which cells should be included based on distance, creating a circular neighbourhood of cells around the one we are currently calculating for. Note that the neighbourhood selects cells, not individual nodes so that we only need to check the distance to a cell rather than the distance to each node inside a cell - see figure 4.1. Since multiple, or even all nodes could be inside a single cell, there is no guarantee for speed-up in terms of time complexity, but in practice, this algorithm is often several times faster.

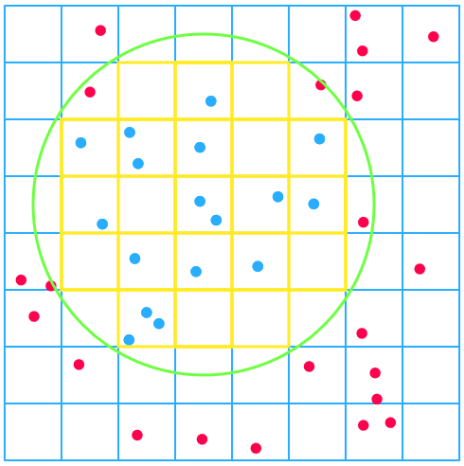


Figure 4.1: An illustration of which cells and nodes are selected when using my circular neighbourhood algorithm, with selected cells in yellow and selected nodes in blue.

```
for v in G(V) do
  G_v = gridNeighs(v)
  for u in G_v do
    if u ≠ v then
      d = u - v
      F = c_K * q^2 / ||d||^2 * d / ||d||
      F_v = F_v + F
    end if
  end for
end for
```

4.2 Quadtree-based

A more advanced approach involves using a data structure known as a quadtree to split up the drawing space. This data structure starts off as a 4-cell grid, but each cell is then split into a quadtree of its own if the number of nodes inside the cell exceeds some limit. One example is what is known as Barnes-Hut simulation where one splits the quadtree as long as there is more than 1 node in each cell [1]. An example of this can be seen in figure 4.2. This process creates a data structure that has a high amount of detail where it is necessary and low where it is not. This in turn allows one to only do detailed calculations for the nodes that are closest, and thus most relevant for electric repulsion, while computing an approximated force for nodes in cells further away, similar to how the grid works. Only dividing cells with many nodes can also allow us to save memory by not allocating a very large but sparse array for locations which the grid-based approach could do. Pseudocode for electric calculations

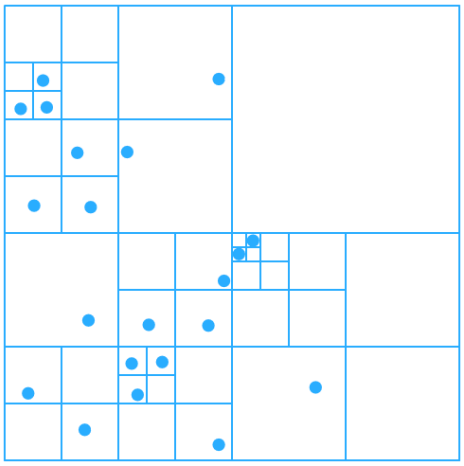


Figure 4.2: An image of the cells of a quadtree split into subtrees in accordance with the Barnes-Hut algorithm. Note that any cell that is not subdivided only contains one node, or none at all.

using this method can be seen in listings 4.2 and 4.3 where 4.2 starts the calculation where 4.3 recursively calculates forces. Since we recreate the quadtree every cycle we can also compute the center of mass and number of nodes in each subtree at this time and store it for force calculations. Theta is a parameter that determines when we start approximating. As can be seen in the expression, a high theta means that we will start approximate calculations for bigger cells at a shorter distance. Similar to the grid-based approach we can make no broad guarantee that the speed of drawing is quicker than $\mathbf{O}(n^2)$, however, when the points are well spread the algorithm will be of speed $\mathbf{O}(n \log n)$ [1]. One could also try having the tree split into fewer pieces by limiting the maximum depth from the root, or the amount of nodes required to be inside a cell for it to continue splitting. I have added these options as parameters which we will look at in chapter 6.

Listing 4.2: Pseudocode describing the loop that starts a recursive calculation of forces from the root of the quadtree for each node.

```
quad = mainQuad
for v in G(V) do
  quad.calcForce(v)
end for
```

Listing 4.3: Pseudocode describing the recursive quadtree-based algorithm.

```
c = quad.centerOfMass
d = c - v
if quad.width / d < θ then
    n = subQuad.numNodes
    F = n * c_K *  q^2 / ||d||^2 * d / ||d||
else
    for subQuad in quad do
        if subQuad.isDivided then
            subQuad.calcForce(v)
        else
            for u in subQuad do
                if u ≠ v then
                    d = u - v
                    F = c_K * q^2 / ||d||^2 * d / ||d||
                    F_v = F_v + F
                end if
            end for
        end if
    end for
end if
G_v = gridNeighs(v)
```

Chapter 5
Quality of a graph drawing

5.1 Measures of graph beauty

The quality of a given drawing can of course be subjectively measured by a person. Often I have experienced that there is quite a bit of overlap between different people for what they find to be an appealing graph. This is however frustrating for any approach where we want to analyse the quality of graph drawing in a predictable manner. It also does not work at all for an automatic process where the quality of a drawing is needed, for example in the parameter selection described in the following chapter.

Fortunately, there are some commonly known measures of graph beauty [2]. Many of these are easy and quite quick to calculate for most graphs, such as the "density" of the drawing (meaning the amount of empty space between nodes), how short the longest edge is, the number of edge crossings and the variance of edge lengths. It is also agreed that symmetrical drawings are more pleasing, although this is a harder metric to calculate. In many applications, Bloom being one example, it is also very important that the model stabilises quickly, which is impacted by the parameter values.

5.2 Implementation

Implementing algorithms to calculate most of these metrics is rather straightforward. Some that would bear closer inspection are the density of the drawing as well as the number of edge crossings. To measure the density of the graph I looked at the area of the smallest bounding box of the graph and divided it by the number of nodes to get the "area used per node". This is not the only way one could measure the density of a graph but one that is simple to compute and understand. For calculating the number of edge crossings I have simply computed the lines through all nodes with edges between them and then for each pair combination looked for crossings in the segments between the nodes. This is also not too hard, but ends up in

$$score = -A \cdot \frac{nCrossings}{nNodes} + |sparseness - 700000| - B \cdot \frac{time}{refTime} \tag{5.1}$$

Figure 5.1: The utility function for graph drawing quality, where A and B are constants that can be set to adjust the importance of the different beauty metrics.

an $\mathbf{O}(n^2)$ algorithm which for large graphs can be quite slow. Some improvements exist for this, for example, the Bentley-Ottman algorithm [3] which can be quite quick when there are few line crossings, however since the calculation of this measurement is not necessary for the actual running of the models, I have focused on other parts of my programs.

In the end, I settled for a utility function that weighs together the number of crossings, the time to draw compared to the time it took for my reference parameters and the sparseness of the graph, measured as the number of nodes relative to the area of the smallest bounding box of the drawing. To somewhat normalise the score, I looked at the number of crossings relative to the number of nodes and the time with these parameters relative to the time needed with reference parameters. For sparseness, the score was already quite similar for "good drawings" of different sizes. The utility function I used to compare fitness using different parameters can be seen in figure 5.1. Sparseness distance from 700000 was chosen because many good drawings with handpicked parameters had sparseness around this value.

Chapter 6
Parameter selection

For all drawing algorithms I have seen, there are multiple parameters that do not have a given "best value". Nevertheless, the choice of values affects how well the algorithm performs, and thus picking the "right" values is something to consider. This is a problem that is also common in the field of machine learning, where the values to be tuned are often called hyperparameters. Similarly to how we before could borrow methods from n-body simulation, we can now borrow algorithms from this field to optimise our parameters. To evaluate a given parameter choice we can use the utility function described above. Since we want parameter sets that work for a multitude of graphs we should take care to pick a set that performs well on all of them.

For my parameter optimisation I will look at the constraint-based algorithm and the spring- and electric-based algorithm, using grid-based electric calculations. In addition, I will look at a constraint-based algorithm that utilises a quadtree-based approach for electric calculations as well as a sub-layout algorithm where all inner layouts are calculated using this same algorithm, optionally with different parameters.

For these four different setups, I have decided to look at the parameters seen in table 6.1, where the values shown are the default values that I handpicked.

6.1 Random Choice

A common and simple approach in machine learning is to simply pick multiple sets of parameters randomly and see how well they perform. Since we can very quickly pick random parameter values we get an approach with high speed that allows us to explore a large parameter space. The downside would be that there is no guarantee that a given random set is any good at all. In order for the sets to be likely to be somewhat useful it is a good idea to have a known subspace of values that could work and only sample from this. In my case I did some manual testing of different values and looked when the model took too long to stabilise, using 1.5x the time of my initial handpicked parameters as a ceiling for what was

Table 6.1: Default parameters for different algorithms.

Parameter	Value
charge	0.5
constraintPrecision	3
cool	0.98
damp	0.15
dt	0.1
e	0.8
logBase	1.00005
r	200

(a) Default parameters for the constraint- and grid-based algorithm.

Parameter	Value
charge	0.5
springConst	60
cool	0.98
damp	0.15
dt	0.1
logBase	1.00005

(b) Default parameters for the spring- and grid-based algorithm.

Parameter	Value
charge	0.5
constraintPrecision	3
cool	0.98
damp	0.15
dt	0.1
e	0.8
logBase	1.00005
r	200
theta	0.5
maxDepth	20
CellMaxSize	1

(c) Default parameters for the constraint- and quadtree-based algorithm.

Parameter	Value
charge	0.5
constraintPrecision	3
cool	0.98
damp	0.15
dt	0.1
e	0.8
logBase	1.00005
r	200
theta	0.5
maxDepth	20
CellMaxSize	1

(d) Default parameters for both the superlayout and the sublayouts in the sublayout-based algorithm. In our parameter optimization these are optimized separately.

considered too long. Once we have a subspace of values for each parameter we can sample randomly from each of these to create a new set of parameters. These can then be used and evaluated by our utility function to see if they're worth using instead of our current ones. For an example in two dimensions, see figure 6.1 where we get a sense of how a large space can be sparsely explored using only a few samples.

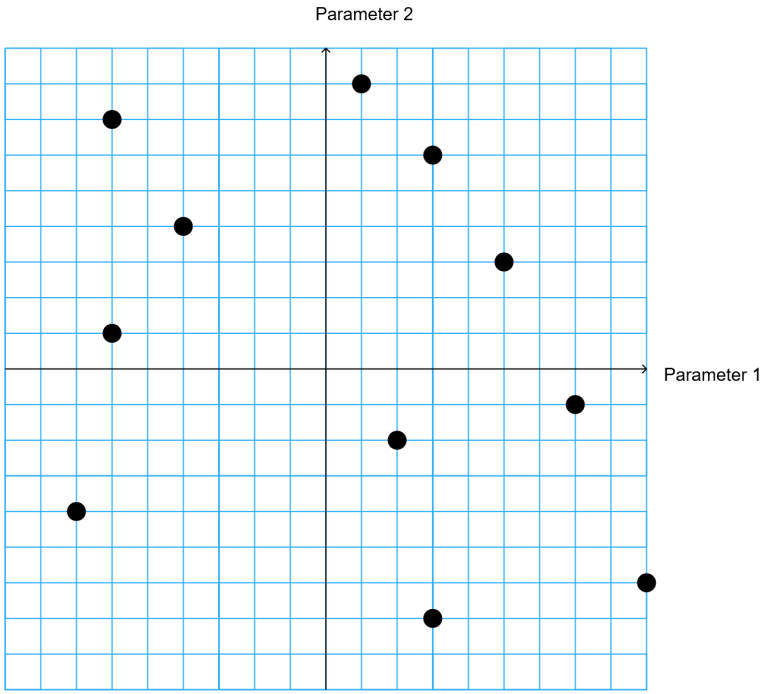


Figure 6.1: Example of random search over two parameters, where each black dot marks the place where one parameter set is sampled.

6.2 Grid Search

Grid search is an approach where we look at all combinations of parameters taking steps of a given length for each one. In two dimensions this can be illustrated as in figure 6.2. In opposition to random search, we quite thoroughly look through a space with many samples, necessarily reducing the size of the search space to compute within a reasonable time. Even if we have a rather large step size over our subspace of values such that we look at a few values in each parameter, the number of operations still quickly grows very large when we use a lot of parameters. For graphs that are not very small, waiting for a single run to stabilise can take a long time. For example, having around 1000 nodes, waiting for a single run to stabilise my original parameter values can take time of the order 1 second (see for example figure 7.4) for my grid-based algorithms. This means that if we have say 7 values for each parameter and 10 parameters we could end up waiting around $7^{10} = 282475249$ seconds - almost exactly 9 years.

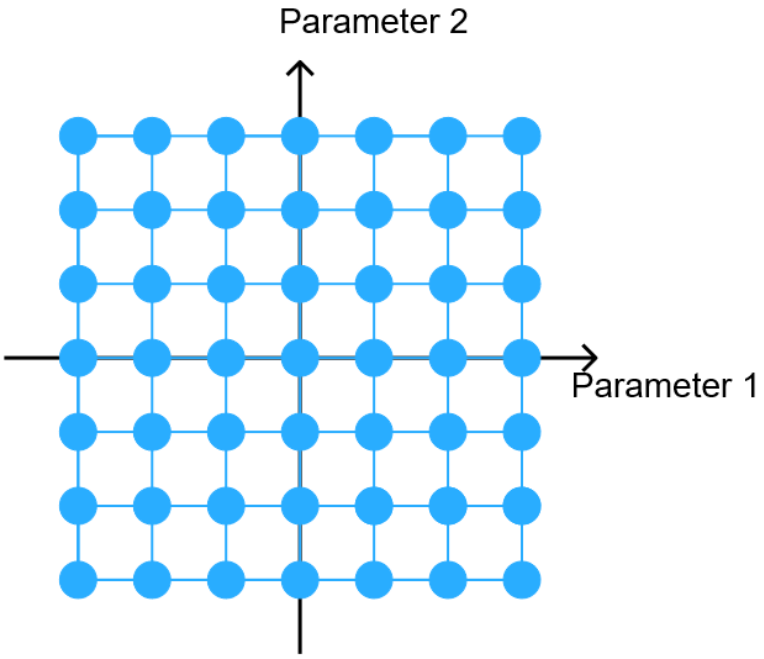


Figure 6.2: Grid search over two parameters, where each blue dot marks the place where one parameter set is sampled.

6.3 Per-parameter search

A much quicker option is to not look exhaustively over the grid, but rather to iterate along the dimension of a single parameter optimising this first, then proceeding to the next dimension. If there is some dependency between the different parameters for the outcome of the utility function this might cause us to miss some good combinations, but we have the big advantage of turning the exponent in the expression in the previous chapter into a factor, making the whole process incredibly quick. Though this is rather simple, I empirically found this approach to be quite useful. In two dimensions this can be displayed as seen in figure 6.3.

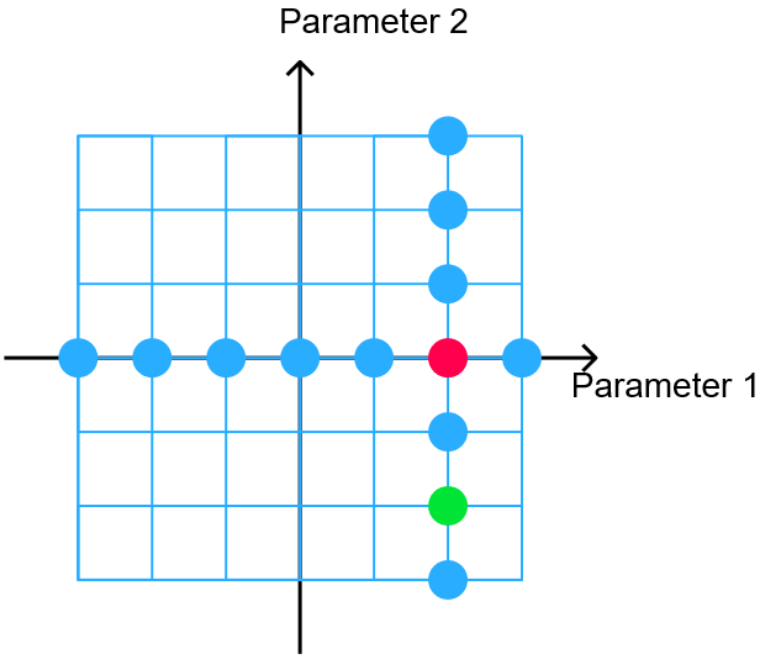


Figure 6.3: Per-parameter search over two parameters, where each blue dot resembles the place where one parameter set is sampled. The red dot marks the optima when looking at parameter 1. The green marks where parameter 2 is optimal for this value of parameter 1.

6.4 Hybrid

An alternative approach is to try to create a hybrid of the random search and one of the iterative searches. This will allow us the advantage of the speed of random choices, but also allow us to optimise each random choice in some local neighbourhood. With the grid search, this will allow us to pick a smaller but denser grid around our given choice. Still, we will have to keep the number of values for each parameter very small in the grid if we want it to be quick. I therefore chose a hybrid approach using both quick options, first retrieving some sets of random parameters that proved reasonably quick and then optimising each using the per-parameter search described above. For a small example in two dimensions, see figure 6.4 which displays how this hybrid approach could find local optima (green) around some good initial random values (black). In this figure, we would have generated random parameter sets until we have three "good" sets, meaning three sets that perform similarly or better than our

hand-picked parameters. We then optimise around these with a per-parameter search.

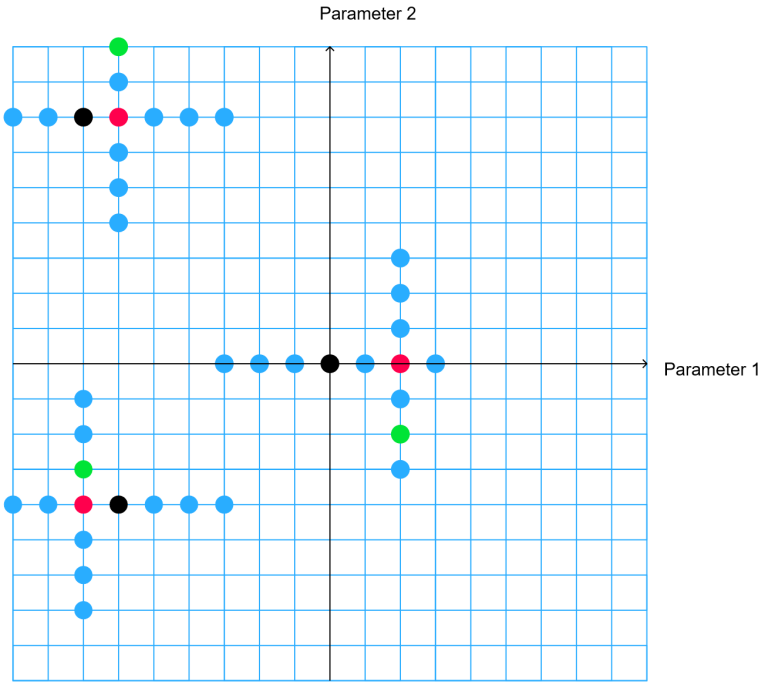
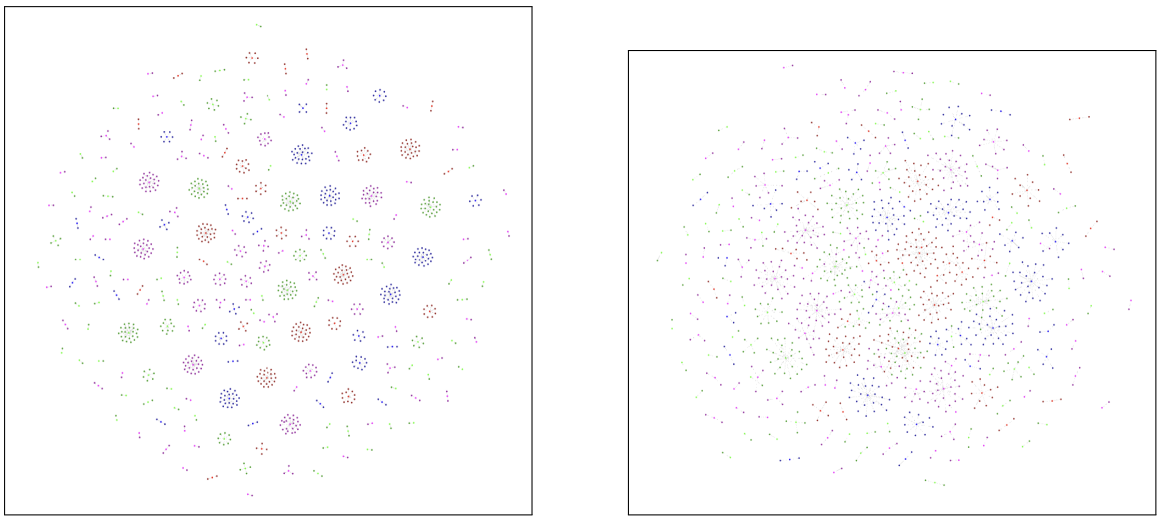


Figure 6.4: Hybrid search, first creating random parameter sets and then optimising these locally.

Chapter 7
Analysis

To analyse the performance of my parameter selection I have created a separate program for nodeJS that will run only the layout part of the algorithms without drawing the graph on the screen in each step. In this software, I have defined graphs with the characteristics described in table 7.1, and then compared the different layout algorithms, as well as optimised their parameters. Before running the optimisation however, I needed to choose the constants for the utility function in figure 5.1.

To pick the constants I looked at the values of our graph beauty metrics for graph drawings that I thought were good, and thought about how problems like having crossings or having a too dense or too sparse drawing should be weighed against each other. After picking constants I then tried to optimise parameters to maximise the utility function, and looking at the quality of drawings made with the optimised parameters I tried to evaluate if I had made a good choice of weights. If the weighting was off I adjusted constants. For example, when the number of crossings in the new drawings was unacceptably high, I increased the weight of this measurement. In this way, one can adjust their utility function to their personal taste for what a good drawing looks like. For a comparison of how the utility function can affect the drawings made with optimised parameters, see figure 7.1. As can be seen in the figure, removing the weight in our utility function on edge crossings can result in a drawing where many of these crossings occur, as is to be expected. After some trial and error, I ended up using the constants seen in figure 7.2



(a) **(b)**

Figure 7.1: A comparison of how parameter selection can differ depending on the weightings of beauty metrics in the utility function. On the right we have removed the weight on crossings.

$$score = -10^{11} \cdot \frac{nCrossings}{nNodes} + |sparseness - 700000| - 10^{8} \cdot \frac{time}{refTime} \tag{7.1}$$

Figure 7.2: The utility function for graph drawing quality, with hand-picked constants for weighting the different quality metrics.

Table 7.1: A summary of the graphs used for comparison and to optimise parameters.

Number of nodes	Number of edges	Description
1205	980	Many sets of small clusters of nodes of different sizes. Each cluster consists of a center node with some surrounding nodes.
3560	3550	Ten sets of superclusters of random size, where a supercluster is a cluster like above except each node around the center is also a supercluster.
3906	3905	Three large hyperclusters.
55	35	A few small clusters and some single nodes.
843	800	A mixture of clusters, hyperclusters and some chains of nodes.

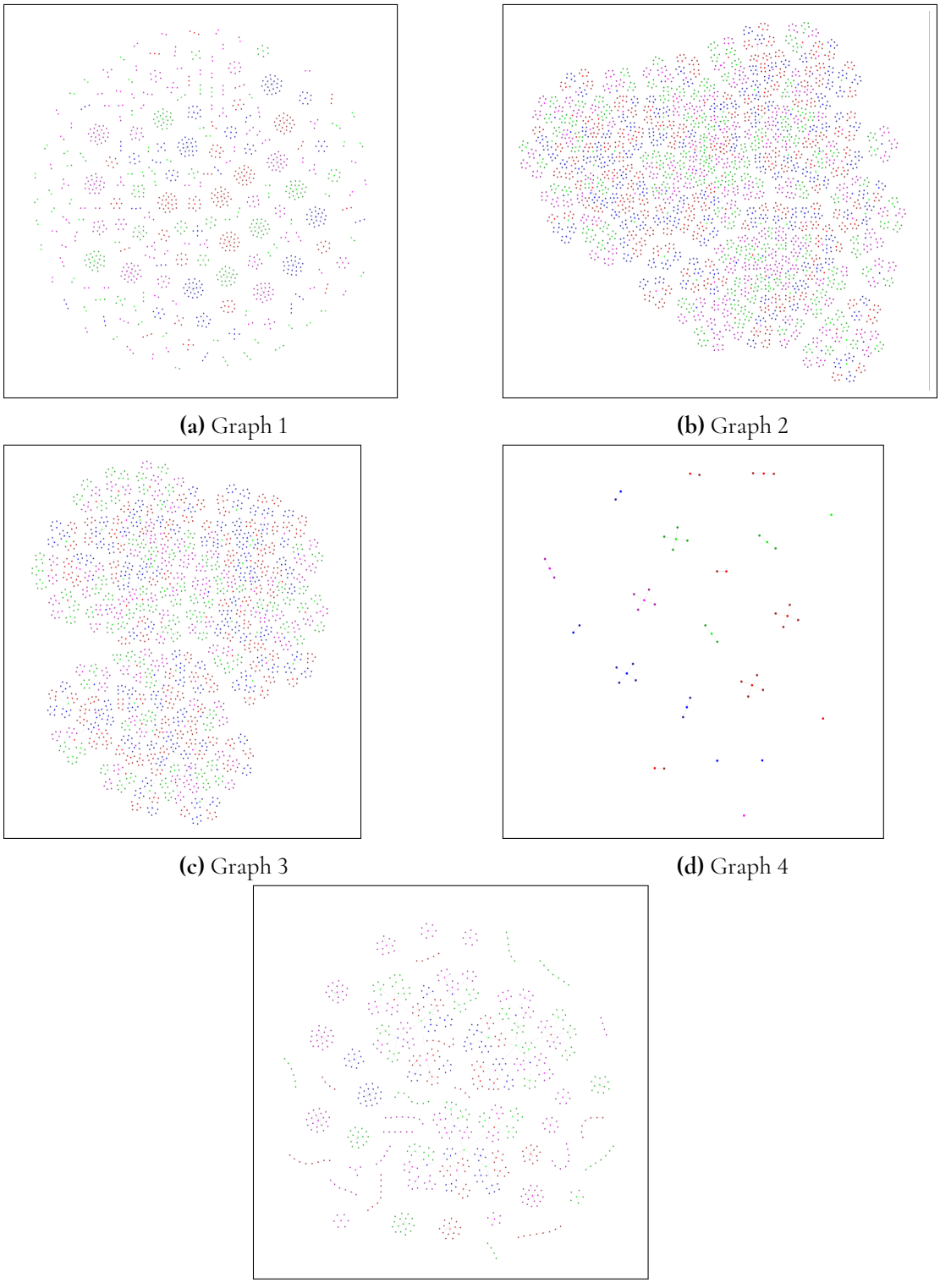
Using the initial parameter values that I picked by hand - values one could describe as "good enough" - I got the drawings seen in figure 7.3 using the constraint-based layout. The default parameter values can be seen in table 6.1.

However, for some algorithms, it is possible that I stumbled upon values that work extra well for this algorithm while on other algorithms the hand picked values were inferior to what further optimisation could give. For a fair comparison we should therefore try to optimise our parameters for all algorithms individually.

I started off with optimising my parameters for a single graph using a single algorithm as a proof of concept. For graph 1 in table 7.1 a comparison of an optimized versus an unoptimized drawing can be seen in figure 7.4 along with the time it took for the drawing to stabilize. The algorithm used was the constraint- and grid-based one. To get a more fair estimate for the time, I ran each drawing three times and took the average time. As can be seen in the figure, the drawing time could be improved quite a bit without any significant loss of image quality.

One problem with this though is that we might pick parameters that are overly specific to one graph. This was indeed what happened. For example, drawing graph 2 with the parameters optimised for graph 1 we got the drawings seen in figure 7.5. Clearly, the quality of the drawing has been ruined by a significant overlap of the disconnected subcomponents.

For different graphs, we get different optimal sets. This is a problem that my supervisor at Neo4j expressed that they also have, for example that values that work for small graphs are less efficient for large ones. One approach to solve this could be to let the parameter values



(a) Graph 1

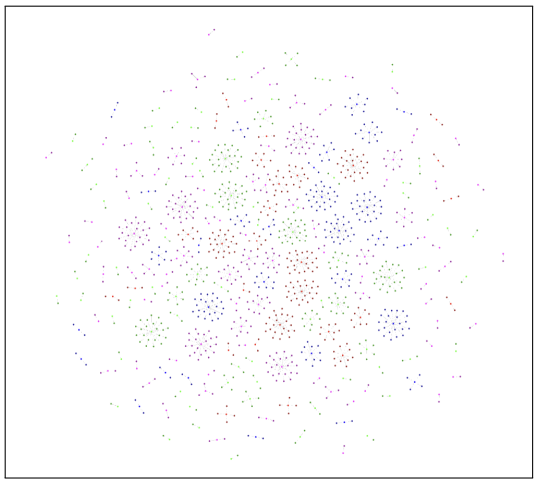
(b) Graph 2

(c) Graph 3

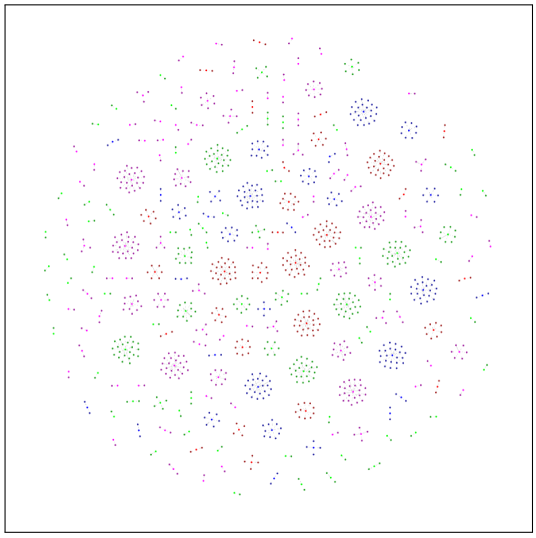
(d) Graph 4

(e) Graph 5

Figure 7.3: Drawings of the five testing graphs described in table 7.1, using the constraint-based algorithm with grid-based electric calculations, with handpicked parameters.

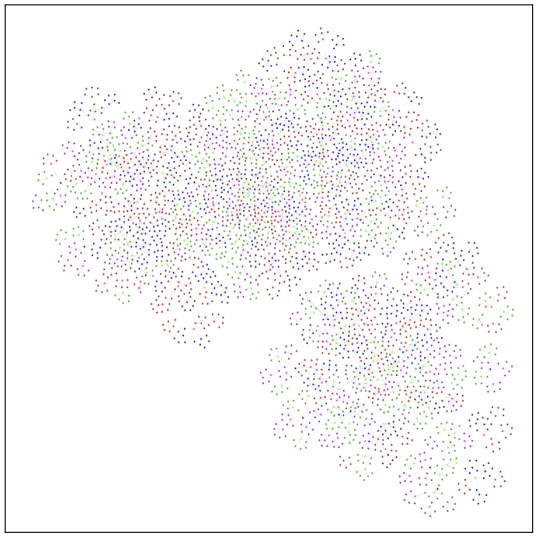


(a) Graph drawing with programmatically optimized parameters. Drawn in an average of 0.50 seconds.

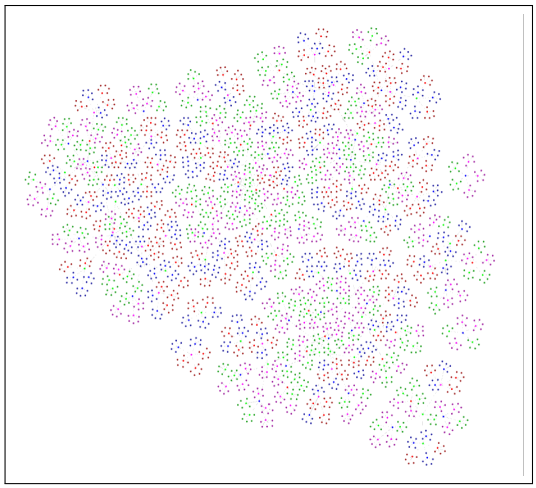


(b) Graph drawing with default parameters. Drawn in an average of 1.43 seconds.

Figure 7.4: A comparison of images and times when optimizing parameters for a single graph.



(a) Drawing of graph 2 using parameters programmatically optimized for graph 1.



(b) Graph drawing with default parameters for graph 2.

Figure 7.5: A comparison of drawings, showing what can happen when parameters are optimised for a different graph.

$$totalScore = \frac{score_1}{|refScore_1|} + \frac{score_2}{|refScore_2|} + ... \tag{7.2}$$

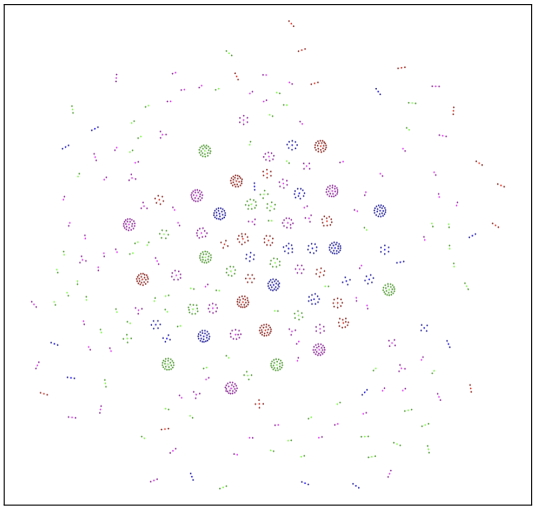
Figure 7.6: The total score, or utility for an algorithm with a given parameter set when looking at multiple graphs, where $score_i$ is the utility score received when drawing graph i.

be a function of the graph structure, most simply as a function of the number of nodes and number of relations. Another simpler approach would be to try to use the found sets to try to find some set of parameters that works quite well for all graphs. An easy way is to see which set of the optimal sets for each respective graph creates the best results for all given graphs by weighting together the evaluation scores for multiple graphs. To get a more fair comparison between multiple graphs I normalised scores by the ones achieved with default parameters, using an equation as seen in figure 7.6.

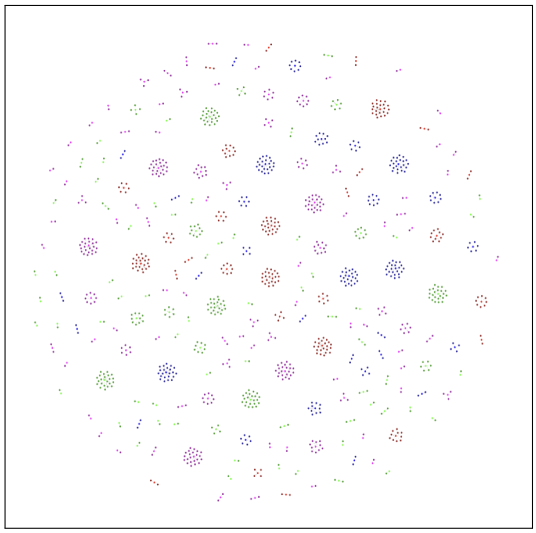
Using this approach I could find new optima for the constraint- and quadtree-based approach that was on average 1.08 times faster, for the constraint- and grid-based approach we got results 1.11 times faster on average, and after a long search the sublayout-based approach could get a big improvement that was about 5.37 times faster. Speed improvements were often larger for big graphs, with the smaller graphs 4 and 5 sometimes even becoming a bit slower. Since these were very fast to draw in the first place I would not see this as a problem however. I suspect that this difference in speedups is due to the fact that I used a timeout for drawing any given graph, and that any drawing slower than this would get a huge penalty basically ensuring that no parameter set where any drawing timed out would be viable. Since speeding up the small graphs could cause the already slow large graphs to slow down further, parameters sets causing this would not be selected.

As mentioned earlier, speed is of course not the only important factor when drawing a graph. Since the evaluation function 7.2 weighs in the number of crossings and the density of the drawing too though, the hope is that the quality of the drawing should also improve, or at least stay as good. To evaluate whether this worked to also improve my subjective perception of the graph I looked at the drawings for each graph by eye. Most times I found the drawing as good, see for example figures 7.7, 7.8 and 7.9. In one case however, I found the optimised drawing to be significantly worse, although not to an unacceptable degree. This can be seen in figure 7.10, where the resulting drawing is too dense in some parts and too spread out in others.

I also wanted to analyse the performance of our different versions of algorithms. First, we can compare the spring-based approach to the constraint-based when using the optimised parameter sets. For both algorithms, we will use a grid for electric calculations. The constraint-based approach turned out faster for all drawings, on average 3.23 times faster, where drawings are of similar quality, sometimes slightly better in my opinion when using constraints. An example comparing drawings can be seen in figure 7.11. Other drawings were even more similar between the algorithms. Secondly, we can compare the quadtree-based optimisation for electric calculations to the grid-based one. In this comparison, I will use constraint-based calculations for maintaining edge distance. In general, I found the quality of drawings to be quite similar, while the quadtree-based approach was a bit slower. A comparison showing the similarity of drawings can be seen in figure 7.12. As seen there are small

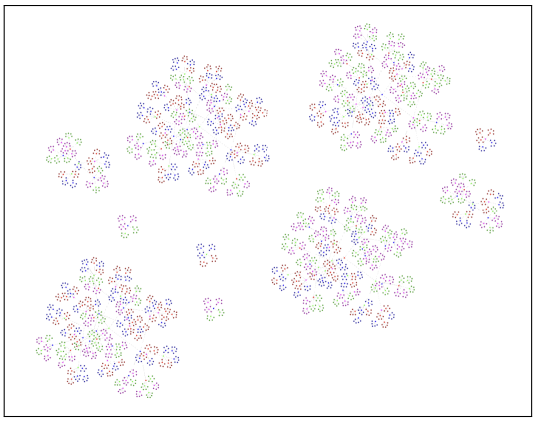


(a) Drawing with optimised parameters. Average time was 3.60 seconds.

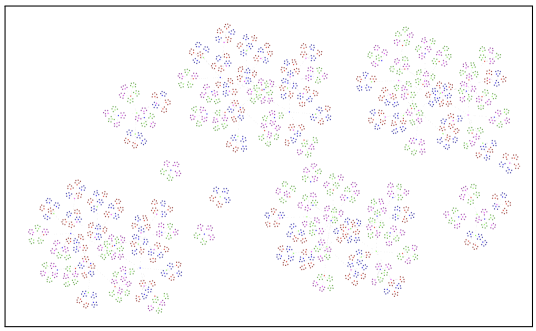


(b) Drawing with default parameters. Average time was 5.39 seconds.

Figure 7.7: A comparison of drawings graph 1 in table 7.1, using the constraint- and quadtree-based approach.

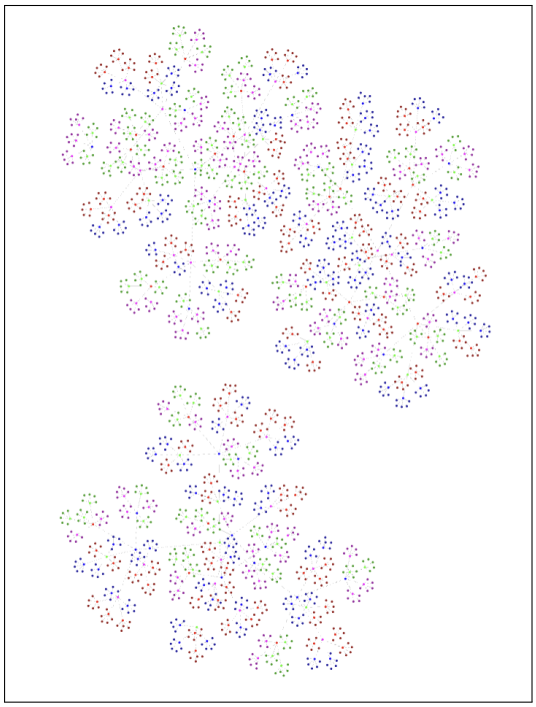


(a) Drawing with optimised parameters. Average time was 9.44 seconds.

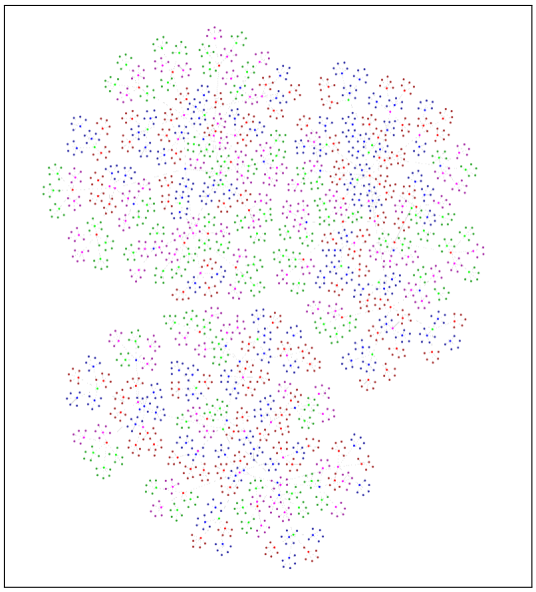


(b) Drawing with default parameters. Average time was 50.61 seconds.

Figure 7.8: A comparison of drawing graph 3 with the sublayout-based approach where sublayouts were drawn with the constraint- and quadtree-based algorithm.

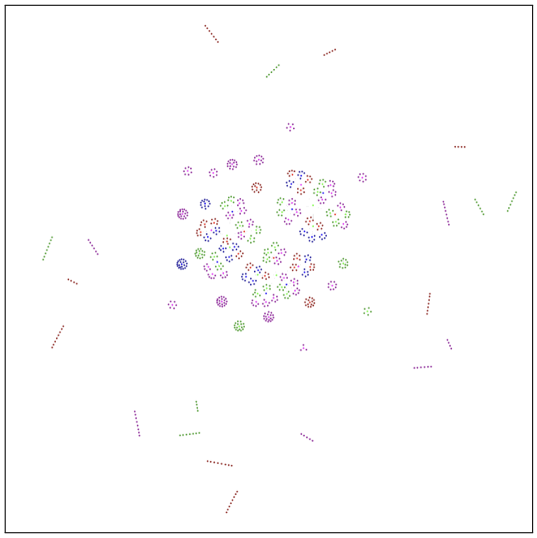


(a) Drawing with optimised parameters. Average time was 2.97 seconds.

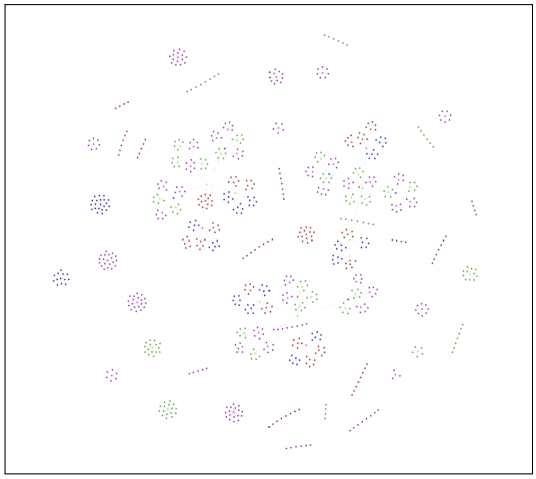


(b) Drawing with default parameters. Average time was 4.09 seconds.

Figure 7.9: A comparison of drawing graph 3 with the constraint- and grid-based approach.

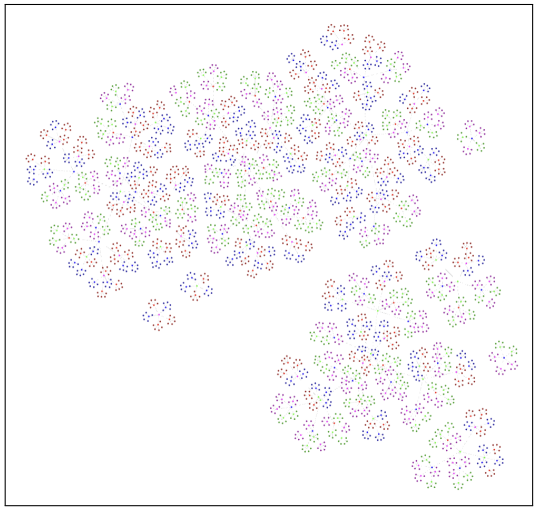


(a) Drawing with optimised parameters. Average time was 3.00 seconds.

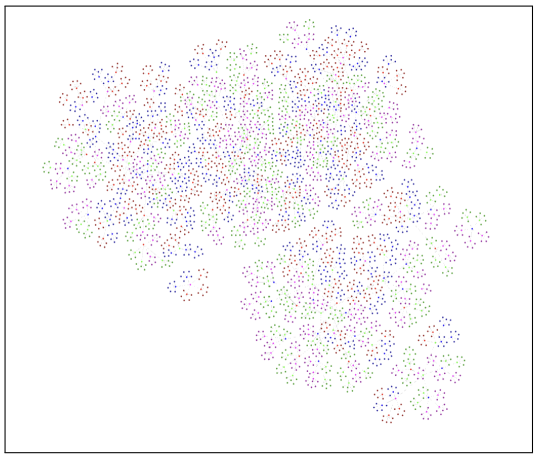


(b) Drawing with default parameters. Average time was 2.28 seconds.

Figure 7.10: A comparison of drawing graph 5 with the constraint- and quadtree-based approach.

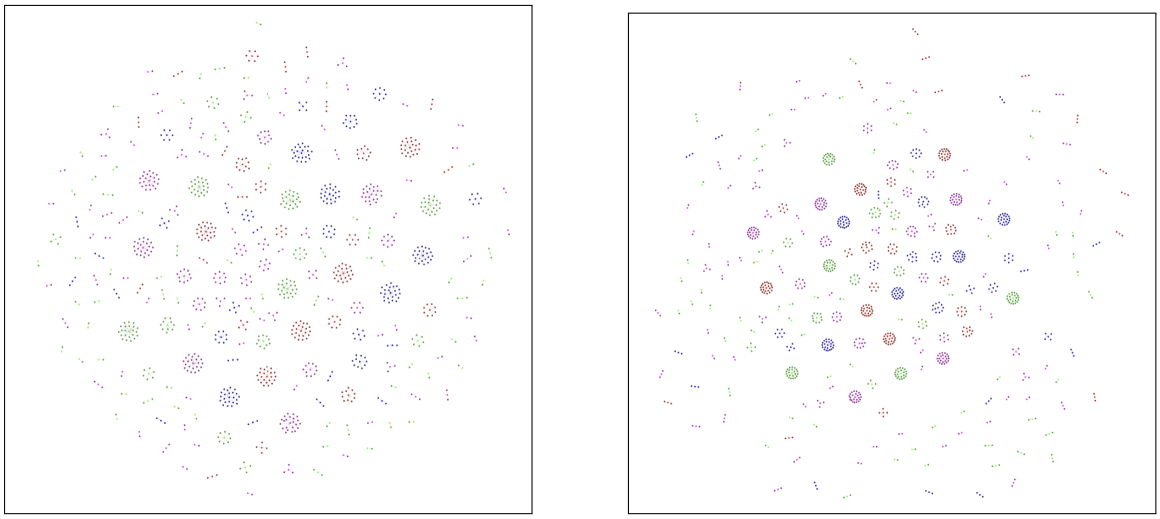


(a) For the constraint-based approach the graph was drawn in an average of 3.93 seconds.



(b) For the spring-based approach the graph was drawn in an average of 9.94 seconds.

Figure 7.11: A comparison of drawings when using the constraint-based approach in comparison to the spring-based one. The graph drawn is graph 2 in table 7.1.



(a) For the grid-based approach the graph was drawn in an average of 1.57 seconds.

(b) For the quadtree-based approach the graph was drawn in an average of 3.60 seconds.

Figure 7.12: A comparison of drawings when using the grid-based approach to speed up electric calculations in comparison to the quadtree-based one. The graph drawn is graph 1 in table 7.1.

differences but overall quite large similarities. This was the case for the other drawings too.

I suspected that the difference in speed was due to the implementation of the algorithms, for example the recursive structure of the quadtree. To motivate that this was the case I looked at how long the different algorithms took when using 1-to-1 electric comparisons and not doing any simplification, which for the quadtree-based approach would correspond to theta=0. For the grid, I simply did a regular loop over the list of nodes. For the same calculations on graph 1 in table 7.1, I then got stability from the regular electric calculations in 10.8 seconds compared to the 30.0 seconds it took for the quadtree-based calculation, using the same parameters. It seems that the recursive structure of the quadtree causes a significant increase in time to access our values. The times to stability for all comparisons above were taken as averages of three runs.

Chapter 8
Conclusions and Future Work

Firstly, lets consider the different drawing algorithms. Using constraints for edges rather than springs was quicker for all my drawings as mentioned before. I also found it easier to optimise parameters for the constraint-based layouts, although this could be because I spent more time picking good default parameters for this method. Subjectively I also found the visual quality higher for the constraint-based algorithm when looking at the resulting graphs, which leads me to believe that this is a preferable method overall.

For the electric calculations, we saw that using a grid was often quicker than using a quadtree. This means that in some cases, unless one manages to significantly improve the implementation of the quadtree-based algorithm, it might be better to use the grid-based approach. One should note however that to reach this speed I had a rather fine grid, and that this causes the risk of running out of memory, or losing out on quality, if the graph grows beyond the grid size too many times. In fact, when I tried drawing the same area with a grid only a few times finer I did actually run out of memory.

Another thing that I wanted to find out was the beauty evaluation function and its use for assessing parameter fitness.

Looking at the time and images in figure 7.4 it is clear that the use of an evaluation function to measure the quality of a drawing can be a good way to measure which parameters to use if we want to optimize drawing of a single graph. Since we weigh the different beauty metrics with constants, setting these in different ways can be used to prioritise different parts of the drawing, as can be seen in figure 7.1. In this way selecting constants for the utility function becomes a quick way to optimise all given parameters at once, and as seen in the analysis this can both improve running time and often my subjective appraisal of drawing quality too, see for example 7.9.

Even for the generalised parameter search, the algorithm often found better parameters for all given graphs, with one notable exception. This signifies that the approach would also be valuable in a more general setting, with the caveat that one should take care to avoid optimising too much to certain graphs by testing for many graphs, and modifying the evaluation function seen in 7.2 if necessary.

A further development of this function could be to not just make the evaluation a multilinear function of the metrics but to also allow different kinds of expressions. For example, the time difference for two very quick drawings might not be too important but grows more important the longer the time is to begin with. One could then make the time expression a power of the length of time, or even exponential with the length of time.

It is worth noting that the results seen in figures 7.7, 7.8, 7.9 and 7.10 are highly dependent on the utility function that I selected. In a product where the aim is to create drawings that are of high quality in the eyes of some general consumer, one would probably wish to use different weights. One idea for future work would be to create drawings with different algorithms, measure multiple metrics and then perform some kind of user study to try to capture how viewers perceive the general quality of multiple graph drawings. This could then allow one to try to find how the different values of the metrics correlate with graph beauty.

It might be the case that a single set of parameters used for all graphs is not the best option and that letting the parameters be a function of the graph structure would be a better option. This would also be an avenue for future work. Another variant of this would be to retrieve multiple sets of parameters optimised for graphs of different structures and then allow the user to pick which parameter set to use in some kind of setting.

Another thing that could improve the algorithm is adding different beauty metrics in order to better encompass what a good drawing is, for example some measure of symmetry. As mentioned in the analysis of 7.10 there can be a problem where some areas of the drawing are too dense and others too sparse. To take this into account one could add a new measurement that looks at variance in density over the drawing.

In summary, we can answer the research questions like this:

- How do different known algorithms for graph drawing compare to each other?

  In our findings, using soft constraints to maintain proper distance over edges proved the superior drawing method, producing quicker results with similar subjective experience of the drawing quality. For global separation of vertices using electric forces, simplifying calculations using a grid or a quadtree could greatly speed up drawing compared to complete calculations. The grid based approach somtimes outperformed the quadtree based one, with the caveat that one can need much more memory for a fine grid.

- Can one create a programmatic evaluation function of a graph drawing that will correspond to the beauty or quality evaluation of the user?

  For this question we found that at least in my case it was quite possible to adjust the mentioned evaluation function to correspond well to my evaluation of the beauty of a drawing when comparing two scores.

- Is it possible to use a programmatic evaluation function to optimise the parameters of a given algorithm for drawing a graph?

  For a single graph we could see a large improvement in drawing time using the chosen evaluation function, without significant difference in drawing quality when looking at such things as number of edge crossings or density of the graph. Therefor it does seem very possible to programmaticaly optimise a graph drawing, with our implementation being one example.

- Is it possible to create a generalised programmatic parameter selection to improve the speed and quality of drawing for any graph?

  When looking at drawing times following the generalised programmatic evaluation we could see a large increase in drawing speed. Though this improvement did not occur for all graphs, it does seem possible to improve how the algorithm performs on average in this way.

it seems that a programmatic evaluation of drawing quality can be useful for both optimising parameters for a single graph, and to find a more general parameter set. The generally good drawing quality resulting from optimising the evaluation function also motivates my belief that one can quite well encapsulate the perceived quality of a drawing by looking at objective drawing quality metrics.

References

[1] Josh Barnes and Piet Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[2] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[3] Jon Bentley and Thomas Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979.

[4] Ariyawat Chonbodeechalermroong and Rattikorn Hewett. Towards visualizing big data with large-scale edge constraint graph drawing. *Big Data Research*, 10:21–32, 2017.

[5] Stephen North Emden Gansner, Eleftherios Koutsofios and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

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

[7] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[8] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.

INSTITUTION OF COMPUTER SCIENCE | LUNDS UNIVERSITY | PRESENTED 2024-04-25

MASTER THESIS Force Directed Drawing Algorithms and Parameter Optimisation
STUDENT Isak Nilsson
SUPERVISOR Jonas Skeppstedt (LTH)
EXAMINER Flavius Gruian (LTH)

Fantastic Graphs and How to Draw Them

POPULAR SCIENCE SUMMARY **Isak Nilsson**

These days it seems like many websites want access to any data they can. But data is not only a powerful force for big social media companies. Anyone can choose to store some information in a database and analyse it for insights into their own life, their company or even the world. Graph databases are a popular way to do this, and an important step in this analysis can be to look at a drawing of your graph.

Mathematical graphs consisting of nodes and edges are becoming an increasingly used way of storing data. To illustrate these graphs, graph database companies like Neo4j commonly use algorithms simulating systems of particles affecting each other with physical forces. For all these algorithms some values often called parameters are used, most times with hand-picked values representing for example the electric charge of a particle. Optimally picking these values can be a cause of both boredom and frustration as the possible combinations are endless. That is where my work comes in. I start off with an algorithm that has a certain parameter set for drawing graphs. Starting with this initial set I then look at some objective quality metrics of a drawing made with these parameters, and programmatically optimise the parameters in a way that manages to speed up the drawing process from hand-picked parameters while maintaining drawing quality.

The beauty of this method is that it is not specific to any given algorithm for drawing graphs since it only needs the final drawing and how long it took to create in order to evaluate a parameter set. This means that one can adapt the method to work for any algorithm that uses some parameter set and improve the efficiency of the algorithm without the need for testing by hand. I implemented this method for multiple algorithms and found significant improvements, especially in drawing speed.

The process can be divided into a few steps. The first step can be considered a broad search for new initial values. In a broad surrounding of the initial parameters, I select random parameter sets and review their graph drawing performance in the same way as for the initial parameters. Then, when a sufficient amount of random sets are found to be performing close enough to the original parameters, I go into the second step. In this step I optimise each random set locally, stepping through a small neighbourhood of each random parameter set and picking values that increase the quality of the drawing in accordance with my chosen objective quality metrics. The final step is then to check that these parameters are not overly specific to one graph. To do this I looked at how well the best parameter sets for one graph performed on a few other graphs and removed any sets where performance was sub-par for some graph, finally selecting the one that performed best overall. In this way, we end up with a new optimised parameter set that in general outperforms handpicked parameters.