Universiteit Leiden
The Netherlands

Computer Science

Autolayout of UML diagrams using metaheuristic algorithms and natural computing

Luca Stoffels (s2335220)

SUPERVISORS:
First supervisor: Dr. G.J. Ramackers
Second supervisor: Dr. A.V. Kononova

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.universiteitleiden.nl

Abstract

Automatic functions for laying out UML diagrams found in commercial tools have been limited so far. This is not ideal as visual layouts of these diagrams play a significant role in visualizing complex system requirements. In this work a layout algorithm is proposed that combines heuristic rules with artificial intelligence features stemming from natural computing. In particular, a bat based particle swarm algorithm is used along side a spring based network layout technique. These natural computing algorithms will optimize the parameters used by the layout generation algorithm, based on data gathered from test users and data generated by a computer. The result is a set of parameters for the layout algorithm that significantly improves the average score of the layouts generated compared to layouts generated with sets of random parameters. Only Class diagrams are covered in this work, however, the created software is designed with future expansion for other UML diagrams in mind. The future applications of a system like this could be a completely self-learning, self-data gathering system that learns how to layout a diagram for an individual or a group of people.

Acknowledgement

The research done for this paper would not have been possible without the dedicated support from my first supervisor Dr. G.J. Ramackers, and the Prose To Prototype project. This project is led by Dr. G.J. Ramackers and is a rapidly developing. Without the systems in this project that already existed I would not have been able to focus solely on this research. Dr. G.J. Ramackers had inspired me to start this research and has keep inspiring me throughout the whole process.
Two years ago I took a class led by Dr. A.V. Kononova about natural computing. This took my interest, with a particular focus on swarm optimization algorithms. This inspired me to make use of these natural computing techniques and I am delighted that Dr. A.V. Kononova is the second supervisor for this research project.
During a meeting with Dr. A.V. Kononova the PhD candidate Jacob de Nobel was part of the meeting and gave some insight full tips and questions.
At last I would like to thank Luuk van den Nouweland and Richard Middelkoop for testing a prototype made for this research project and giving feedback.

Contents

1 Introduction

At the start of a new software project, a significant amount of designing and documenting gets done by people before the actual implementation gets worked on and part of this is the making of UML diagrams. These diagrams require a human to make a layout and this is impractical as it leads to a lot of time getting spent on designing a practical layout for a UML diagram by hand. If this process could be sped up it would result in a more cost-efficient early stage of software development. There are basic auto-layout software technologies that could give a graph an auto-layout, but these are often limited and still require a human to make subtle and big changes to the graph, especially for UML diagrams. On top of that they tend to be rule-based and rules sometimes miss the mark when it comes to catering to the variability in human view on whether a graph looks good.

By collecting data about users' experience of layout a layout algorithm could be trained to reflect the needs of a group of people, but also individuals. By combining a rule-based layout with a system that deforms the layout based on a list of parameters, the parameters could be optimized to give the best results, whilst the rules still influence the layout. For this optimization multiple systems can be used. In this work an evolutionary-inspired algorithm is used to train a model that predicts a score that a human would give to a layout based on gathered scores from test users. This is then used to rank layouts generated with random parameters for the layout algorithm. The resulting data is used by a particle swarm optimization algorithm to find the best set of parameters for generating a layout. Using algorithms inspired used in natural computing for layout generation systems could lead to significant improvements of the state of the art for UML layout generation.
Therefore the question; could natural algorithms for multi-dimensional optimization problems be used to optimize an auto-layout algorithm for UML system design?

In section 2 related work and background knowledge is described. This sections contains background knowledge needed to know in order to properly understand the work in this research, as well as works by other authors explaining matters in more detail.
In section 3 the inner workings of the created prototype are explained, including a detailed overview of the layout algorithm. This section also describes the data gathering from users and data generation by the computer, along with the data processing to prepare it for the bats algorithm.
In section 4 the results of the prototype are shown, including statistics of the performance and layouts generated by the layout algorithm. At last in section 5 the results are discussed and future work is proposed

2 Related Works and Background

2.1 Unified Modeling Language

The Unified Modeling Language (UML) is a modeling language mainly used in computer engineering and software development. With this language diagrams can be made to easily display information about the architecture and requirements of a system. This is very useful during the design of the workings of any software project, as usually many components rely on the specific workings of other components within the project. In 1998 Andrew Lyons [Lyo98] created an extension for UML that was "specifically fine-tuned for the development of complex, event-driven, real-time systems, such as those found in telecommunications, aerospace, defense, and automatic control applications" ([Lyo98], p.6) . UML has 14 different types of diagrams which are all used to display different aspects of projects, ranging from user interaction to the most complicated inner workings of the system. Gianna Reggio et al. [GR15] did research into which of these diagrams are used the most and why. They found that the class diagram is used the most (at 100% of the time in any of their sources) as "it is indeed the fundamental diagram of the UML" ([GR15], p.7), closely followed by the activity diagram (at 98%) and the sequence diagram (at 97%). For this work usage of different UML diagrams is out of the scope, as all the diagrams have their own rules and ways of displaying information. Trying to make a layout algorithm with all these rules at once would be unavailing as there is no need to have the rules for these diagrams mixed up, given that they are displayed separately. Given that the class diagram is used the most we focus on exactly that diagram in this work.

2.2 Autolayout

Hauke Fuhrmann et al. [HFvH] proposed an automatic layout algorithm for UML diagrams. This algorithm focuses on recursively subdividing the elements into subgroups, after which a layout is applied to the nodes in the subgroups, and then subgroups themselves, moving up the hierarchy. In this research a similar approach is taken; however, a different method of applying the layout is used along with a different way of dividing the elements into subsets. Also, after applying the layout based on subgroups a spring based algorithm is used alter the layout even more. This algorithm used is based on a springs algorihtm proposed by Thomas Kamps et al. [KKR96]

2.3 Heuristic and Algorithms

Heuristic algorithms are a type of algorithms that trade off accuracy for speed, whilst focusing on approximating a solution, rather than finding the exact solution. Vincent Kenny et al. [VKS14] wrote a survey of well-known heuristic algorithms along with example problems. One of these problems is the traveling salesman problem which is as follows: "given a list of cities and the distances between each city, what is the shortest possible route that visits each city exactly once?" ([VKS14], p.3). The nearest Neighbour algorithm starts from any of the cities after which it moves to the closest city. After this, the distances from the current city to all not yet visited cities are compared and the closest city is picked to move to. This process repeats until all cities have been

visited. "This algorithm is heuristic in that it does not take into account the possibility of better steps being excluded due to the selection process" ([VKS14], p.3). This could and probably will result in a path that is not optimal; however, considering that there is no way to precisely compute the shortest path without brute-forcing the problem, an algorithm like this might prove itself useful nonetheless. Especially when using a large number of cities as for every city the time taken to brute force the solution multiplies by the total number of cities.
Laying out a UML diagram also does not have a perfect solution, however, with a heuristic based algorithm a good layout might be found anyway.

2.3.1 Meta-Heuristic Algorithms

Meta-heuristic algorithms are a category of algorithms that are not that dissimilar to heuristic algorithms, with the difference lying in the applicability. Heuristic algorithms are designed specifically to solve one problem, whereas meta-heuristic algorithms are designed so that they can be used to solve a variety of problems that stem from the same general problem concept. The "No Free Lunch theorem", as defined by Wolpert and Macready 1996 NoFreeLunchTheorum, states that there is no meta-heuristic algorithm that is best at solving every problem. This is due to the ways meta-heuristic algorithms are not focused on finding the exact solution, and instead make trade-offs in accuracy for speed. For different kinds of problems different sets of trade-offs are best, therefore there is no way that one meta-heuristic algorithm is best. This is relevant for this research as there are multiple types of UML diagrams and many possible structures within the networks for these types of diagrams. By combining different techniques, and allowing for change within settings for these techniques based on the type of UML diagram, hopefully the algorithm made for this research could be used in the future to layout types of UML diagrams not covered in this work.

2.4 Natural Computing

In the modern world it is easy to view computers and the algorithms that run on them as something unnatural. Many algorithms and computational systems are designed based on natural processes. According to Leandro Nunes "Natural computing is the computational version of the process of extracting ideas from nature to develop computational systems, or using natural materials (e.g., molecules) to perform computation" ([de 07], p.3). There are multiple branches of natural computing like computing algorithms to simulate natural phenomenona, computing using natural materials. In this work these forms of natural computing are not of interest. The one that is relevant here is the branch where the computing itself is inspired by nature. There are many complex problems in the world that are solved every day by organisms and natural occurring movements. Think of ants always finding the way home, and water flowing. Leandro Nunes [de 07] describes that this way of natural computing arose with two main goals in mind; " devise theoretical models, which can be implemented in computers, faithful enough to the natural mechanisms investigated so as to reproduce qualitatively and/or quantitatively some of their functioning" ([de 07], p.4) and to provide alternative methods of complex problem solving for problems that could not yet be solved with traditional methods.

2.4.1 Particle Swarm Optimization

Swarm intelligence is a term that refers to a system with seemingly unintelligent agents that rely on rules that make their individual capabilities simplistic, yet in a group show intelligent behavior. Examples of swarm intelligence algorithms are the artificial bee colony, and the ant colony optimization algorithm. The one that is of interest to this project is the particle swarm optimization (PSO) algorithm. Kennedy and Eberhart originally found the inspiration for this kind of algorithm in 1995 in trying to replicate human social behavior. Individuals would be capable of interacting with their neighbors and the environment around them, leading to behavior on a population level. "Although the original approach has also been inspired by particle systems and the collective behavior of some animal societies, the main focus of the algorithm is on its social adaptation of knowledge" ([de 07], p.11). García-Gonzalo et al. [GG12] reviewed the history of particle swarm optimizing techniques and Mahmud Iwan Solihin et al. [MISK11] used it to tune a PID controller.

2.5 Multi-Dimensional Problems

There are many types of optimization problems, with one of them being multi-dimensional optimization problems. These problems focus on optimizing multiple dimensions at once. This means that instead of trying to find the optimal value for one parameter, a set of values is found for multiple parameters. This results in a combination of values for parameters which with those values will produce the best result.

2.6 Bat-inspired Algorithm

CXin-She Yang [Yan10] proposed a bat-inspired metaheuristic particle swarm algorithm.

2.6.1 Bats In Nature

As this algorithm is inspired by microbats, a type of bat, it is essential to understand the bats' behavior. CXin-She Yang [Yan10] describes this behaviour very well. Microbats use echolocation to detect objects in the dark. The properties of their pulses used for echolocation vary depending on the species of bats and what they are doing at the time. These pulses have a constant frequency which for most species is between 25kHz and 100kHz. These pulses usually last between 10 to 20 ms for microbats. The rate of these pulses could go up to 200 pulses per second when they fly near their prey. "Amazingly, the emitted pulse could be as loud as 110 dB, and, fortunately, they are in the ultrasonic region" ([Yan10], p.3). With all this, the bats can detect the distance to, orientation of and moving speed of their target. "Such echolocation behaviour of microbats can be formulated in such a way that it can be associated with the objective function to be optimized, and this make it possible to formulate new optimization algorithms" ([Yan10], p.3).

2.6.2 Bat Algorithm Workings

Here is a simplified overview in pseudo-code of the bats algorithm.

$$f_i \leftarrow f_{min} + (f_{max} - f_{min})\beta, \tag{1}$$

$$v_i \leftarrow v_i + (x_i - x_*)f_i, \tag{2}$$

$$x_i \leftarrow x_i + v_i \tag{3}$$

$$\lambda = \frac{v}{q} \tag{4}$$

$$x_{new} \leftarrow x_{old} + \epsilon A \tag{5}$$

$$A_i \leftarrow \alpha A_i, \tag{6}$$

$$r_i \leftarrow r_i^0[1 - exp(-\gamma t)] \tag{7}$$

Algorithm 1 Bat algorithm

for $i = 1 \rightarrow M$ **do**
    Initialize population x_i (i= 1, 2, 3, ..., n) ▷ Initialize
    Initialize velocity v_i
    Define pulse frequency $q_i \epsilon [0, 1]$ *at* x_i
    Initialize pulse rate $r_i \epsilon [0, 1]$
    Initialize loudness $A_i \epsilon [1, 2]$
end for
while termination criteria are not met **do**
    $f_i \leftarrow f_{min} + (f_{max} - f_{min})\beta$ ▷ Generate new solution by updating the frequency Eq. (1)
    $v_i \leftarrow v_i + (x_i - x_*)f_i$ ▷ and the velocities Eq. (2)
    $x_{new_i} \leftarrow x_i + v_i$ ▷ and the solutions Eq. (3)
    if $rand > r_i$ **then**
        Select a solution from the best solutions ▷ Sort all solutions and grab a random one from the top 5
        Generate local solution around the selected best solution ▷ Eq. 5
    end if
    if $rand < A_i \ AND \ f(x_{new_i}) < f(x_i)$ **then**
        $x_i = x_{new_i}$ ▷ Accept new solution
        $r_i = r_i * 1.01$ ▷ Update pulse rate Eq. (7)
        $A_i = A_i * 0.99$ ▷ Update loudness Eq. (6)
    end if
    Rank the individuals and find the best current x_*
end while

2.7 Prose To Prototype Project

The prose to prototype (P2P) project is a project led by Dr. G.J. Ramackers. The focus of this project is to reduce the time spent on tasks in software development that do not add production value or knowledge. Tasks like listing requirements for parts of

the software when a general text describing the full process are already made. Dr. G.J. Ramackers et al. [GJR21] describes a way to pull these requirements straight from the text, with the use of artificial intelligence (AI). The work described in this paper is developed to be part of this Prose To Prototype Project.

2.7.1 Rendering

The prose to prototype project has a visualizer that renders UML diagrams, on which this research relies. This renders made by this visualizer have some shortcomings as it is still a work in progress, but these were not a big influence on the layout of the algorithm. The most important point for this research was that the lines in the layouts would attach to seemingly random parts of the nodes, which resulted in some connections being off center by a couple of pixels. This is not ideal, but it only has a minor effect on the layout an it is not the focus of this work, as the focus mainly lies on the placements of the nodes.

3 Methods

3.1 Layout Generation

The layout algorithm is a combination of various techniques. It starts with subdividing the network of nodes into smaller sets of nodes. This is done to reduce the complexity of generating a layout. By subdividing the network into subsets, we can first focus on generating a layout for the subsets, and then generate the final layout by moving the subsets around. After dividing the network of nodes, we pass the subsets on to the heuristic layout generator. This layout generator creates a simple layout taking into account the hierarchy of the nodes. This means that the parent classes will be above their sub-classes, which are next to each other. If two nodes are not connected in a way where one has to be above the other, then they are put next to each other. This results in a layout with layers of nodes with connections between and within the layers. After this, the springs algorithm [KKR96] will be applied to the layout, which pulls and pushes the nodes from and to each other.

3.1.1 Subset Division

The splitting of the network into subsets is done recursively and starts by defining a rootset. This naturally starts with all nodes in the network. After this, the rootset is made to split recursively. This is done in multiple ways as splitting into subgroups can be done based on multiple criteria.
The first one is connectivity. Generally, a UML diagram contains one network of nodes, but when splitting the sets a subset might contain nodes that cannot be reached from all other nodes in the set by only making use of connections within the set. In this case we split the set grouping the connected nodes together. This is done by taking a starting node from the set and along with all the reachable nodes putting it in a subset, and then repeating the process until all nodes are divided. Since we do not want to take unreachable nodes into account when splitting on other criteria, this is the first criteria any set is split on.
The second criterion is being part of a cycle. It is of great importance to realize the difference between cyclic parts and acyclic parts in a network. This is because we can layout an acyclic network by simply creating a tree-like layout. Whereas for a cyclic layout this is not always possible, especially when taking into account the rule that parent classes need to be above sub-classes. In order to split on whether nodes are part of a cycle or not, first all the cycles within the set (only making use of connections where both nodes are part of the set) are listed. This is done by walking every possible path in a breadth-first search, until either a dead end is reached, or a node that is already part of the path is reached. In that last case a cycle is detected. This cycle includes the reached node that was already part of the traversed path and all the nodes that follow it in this path. Then every node that is part of any of these cycles will be put in one subset, and the other nodes will be put in the other subset.
The third way of splitting a set is by splitting a set that contains only cycles. The cycles are listed as described previously, and duplicate cycles are removed. This is important as cycles will be listed at least twice due to the breadth-first search walking the cycle one way and then the other way. Removing duplicate cycles is important as we need to know how many cycles there are in the set. If the set contains multiple cycles the

smallest nodes that form the smallest cycle are put in one subset, and the rest of the nodes in the other subset.

After splitting a set the created subsets are given a tag that describes the type of the subset. These types are:

- **Initial**: the initial state of a set. If nothing else is specified this set is treated as the root set .
- **Initial but connected**: a fully connected set, but nothing is known about cycles.
- **Acyclic**: a set that does not contain any cycles.
- **Cyclic**: a set where every node is part of a cycle that consist of only nodes in the set.
- **Rest of a cyclic set**: the rest after splitting the smallest cycle from a cyclic set. This set might contain more cycles and might be fully connected.
- **Rest of a cyclic set and connected**: the rest after splitting the smallest cycle from a cyclic set. This set might contain more cycles and is fully connected.
- **Rest of a cyclic set and cyclic**: the rest after splitting the smallest cycle from a cyclic set. This set contains more cycles and is fully connected.
- **Cycle fragment**: the rest after splitting the smallest cycle from a cyclic set. This set is fully connected, but does not contain any cycles.
- **Dead end**: contains a dead end that does not contain any cycles.

For each of these set types holds that one of the previously mentioned splitting criteria is used to split. Figure 1 shows a flowchart of set types. The rootset starts as the ”Initial” type, after which it gets split recursively and eventually all the sets that contain nodes will be of a type that is marked green in figure 1.

Figure 1: Flowchart of set types. The green set types are final types

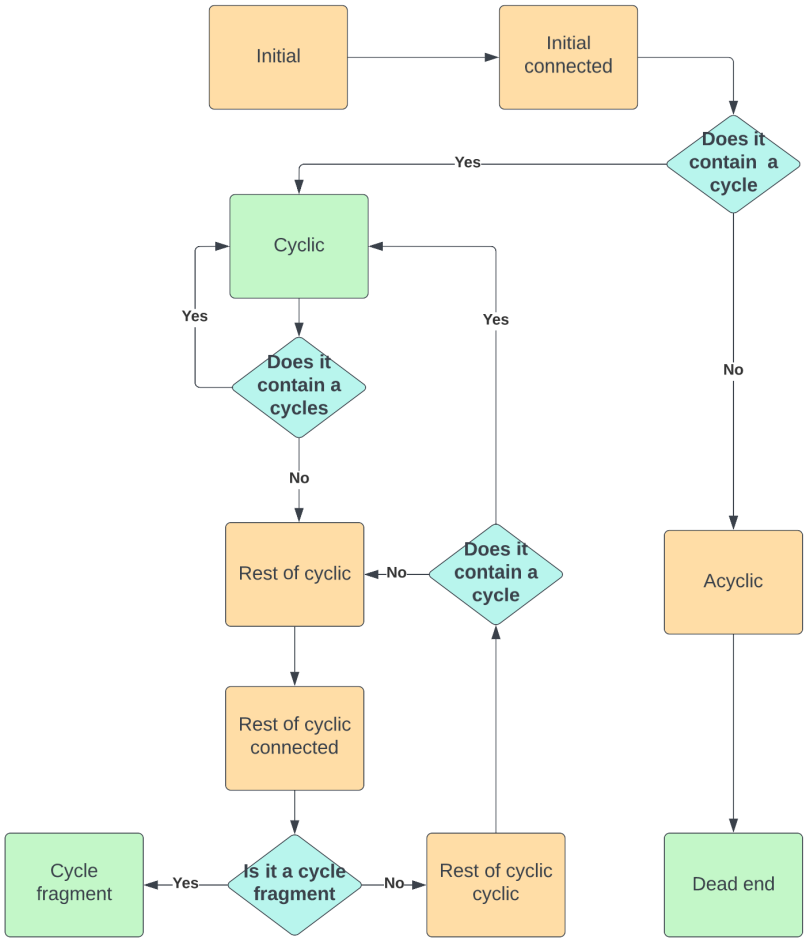
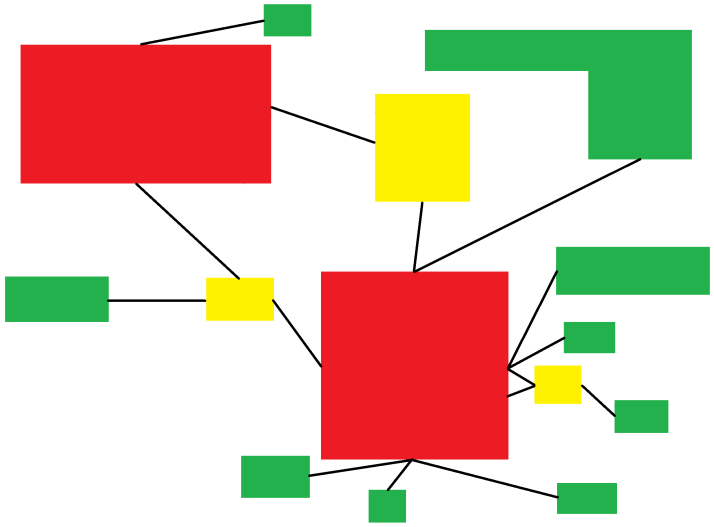


Figure 2 shows a network that is divided into subsets, with every colored blocks representing a set than contains one or more nodes. The red blocks are sets that contain one cycle, the yellow blocks contain cycle fragments and the green blocks contain dead ends.

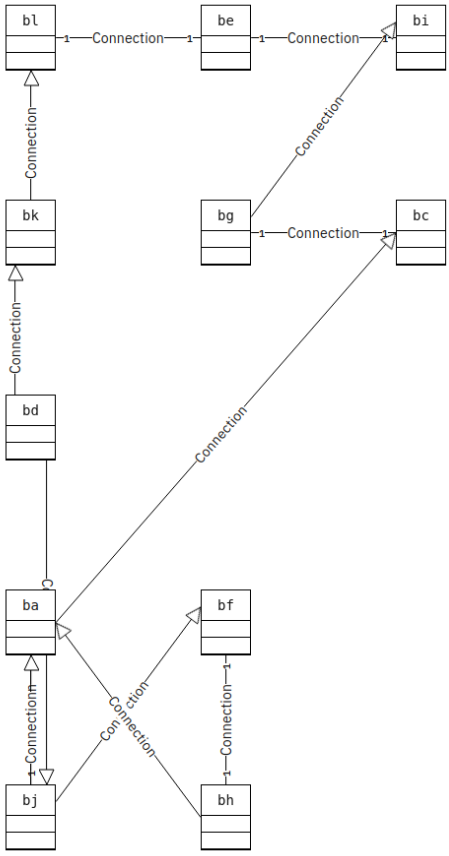
Figure 2: Network divided into subsets. Red is a cycle, yellow is a cycle fragment, and green is a dead end



3.1.2 Heuristic Layout

After the network is divided into subsets each subset gets a layout. This layout is based on two simple rules: (1) if two nodes have a parent-child relationship, then the parent needs to be above the child, and (2) if no parent-child relationship exists between two nodes then they are put next to each other.
A starting node is chosen and is put in an empty layer. Then all the nodes connected to the starting node are put either in the same layer or a layer above or below depending on which node is the parent. By adding all nodes of the set whilst taking into account the relationships between nodes, we eventually end up with a set of layers that represent a hierarchy of nodes. When a node is added to the layers it is put in a queue. For every node in this queue the connected nodes that are not yet in the layers will be added, after which the node is removed from the queue. For a set of nodes without cycles this results in a set of layers where nodes can be moved horizontally without passing each other to make a tree structure without overlapping lines. For a set of nodes that does contain a cycle this is not the case. Also in that case there might even be connections between layers that have one or more layers in between.
After the layouts for all the subsets have been generated, the relations between sets are determined. If between two sets there is one connection of the parent-child type, the sets containing the parent node is marked as the parent set of the set containing the child node. Once all the relations between sets have been determined, the exact same process happens as when the nodes within a set are laid out, but this time with sets instead of nodes.

Figure 3: The layout after applying the heuristic layout



As seen in figure 3, after the heuristic layout every node is aligned properly and parent-child relations are satisfied where possible. However it is not without issues. Node *bd* is connected to node *bj*, even though they are not in layers that have direct contact, and on top of that *bj* should have been above *bd*. This might not be clearly visible at first as it looks like *bd* has a connection with *ba*, but actually the word "connection" is poorly placed with the first letter looking like the tip of an arrow.

3.1.3 Springs

The last step after the heuristic layout is generated is the springs algorithm. Thomas Kamps et al. [KKR96] proposed a spring-based layout algorithm, however, this had the constraints of setting every edge to the same length. Here that is not the case. Instead, this algorithm will move every node once per iteration to and from other nodes based on a spring-like structure.
For each node a total movement m_t is calculated by calculating movements m_i to or from all the other nodes and adding those movements together. These small movements are calculated by taking the target distance $targetDistance$ and subtracting the actual distance $actualDistance$ and then multiplying it with a force $force$.

$$m_i \leftarrow (targetDistance - actualDistance) * force \tag{8}$$

$$m_t \leftarrow sum_{i=0}^{nrNodes} m_i \tag{9}$$

There is a separate target distance and force for connected nodes and unconnected nodes. This is to allow greater control of the layout generation.
When determining m_i based on a connected node, if both nodes share a similar x or y coordinate, the corresponding axis is marked as aligned. If $align_{enable}$ is set to true this means that the marked axis is set to 0 in m_t to ensure that the nodes stay aligned if possible. How close the x or y coordinates need to be in order to be marked as aligned is determined by $align_{distance}$.
The second to last step is the random chance $rand_{chance}$ to add a random vector with magnitude $rand_{magnitude}$ to the total movement. This random movement ignores alignment to prevent nodes from never moving on an axis.

$$\textbf{if } randomizer < rand_{chance} \textbf{ then} \quad m_t \leftarrow m_t + [random(-1,1), random(-1,1)] * rand_{magnitude} \tag{10}$$

At last, there is a check for overlapping connections. If $checkOverlap$ is set to true the total number of overlapping connections will be calculated. If after applying m_t to the node there are more overlapping connections, the move will be undone.

3.1.4 Parameters

In total there are 13 parameters that are given to the layout algorithm, which influence the layout generation. Below is the list of these parameters, along with what they represent in the algorithm:

- $HeuristicDistance$ The distance between nodes after generating heuristic layout
- $force_{connected}$ The force used to influence movement magnitude based on connected nodes
- $targetDistance_{connected}$ The target distance used to influence movement based on connected nodes
- $force_{unconnected}$ The force used to influence movement magnitude based on unconnected nodes
- $targetDistance_{unconnected}$ The target distance used to influence movement based on unconnected nodes
- $align_{distance}$ The distance in which a node is considered aligned with other nodes
- $align_{enable}$ If enabled, cancel movement on axis if a node is aligned on that axis
- $checkOverlap_{interval}$ The accuracy for checking if connections overlap
- $checkOverlap_{distance}$ How close two connections have to be to count as overlapping
- $checkOverlap_{enable}$ If enabled, cancel a move if it leads to more connections overlapping
- $rand_{chance}$ The change of the random step getting added

- $rand_{magnitude}$ The maximum magnitude of the random step
- $nrIterations$ The number of iterations done by the springs algorithm

3.2 Gathering and Tuning

In order to find the best values for the parameters used in the layout algorithm, user data is necessary. This is because we need to define a way to score a set of values for the parameters, and we cannot do that without gathering data about what people think of the generated layouts. Therefore a number of test users were asked to score the generated layouts of simple diagrams on a scale of 1 to 10. This scale was chosen as it is detailed enough to pack information, but not so detailed that users might become inconsistent when scoring the layouts. When a user would generate a layout, first a diagram was chosen at random out of 15 diagrams. These diagrams varied in size and complexity, with some being very simplistic and some containing hard to layout features like cycles. After that, a layout was generated with random values for the parameters that influence the layout algorithm. Then the score of the user was saved along with the parameter values for the generation algorithm.

In order to let a PSO work properly, a lot of data is needed. Given that the layout generation has 13 parameters, it is very time-consuming to manually score enough diagrams to gather a lot of data. In order to solve this problem computer-generated scores are used alongside the user scores. These computer scores fill in the gaps where there is no user data. However, if computer-generated scores are used alongside user scores a similarity between the two is important. Otherwise, if we would have computer scores that are very different from the user scores, we would still have the problem of not having enough user data. Therefore we need to create a function that will try to score the layouts as close as possible to the user score. In order to do this a dataset was generated consisting of generation parameters (just like for the user test) and statics of the layout generated with these parameters. These statics are:

- **Average connected node distance**: The average distance between all pairs of nodes that have a connection between them
- **Average unconnected node distance**: The average distance between all pairs of nodes that do not have a connection between them
- **Overlapping-fraction**: The number of overlapping connections divided by the number of connections
- **Alignedness**: The average offset between nodes on the axis with the smallest difference
- **Correct relational position fraction**: The amount of child nodes below the parent node divided by the total number of parent-child pairs

All statistics were normalized with 0 equaling the smallest value found in all layouts and 1 equaling the biggest value found in the layouts. This means that we can define a set of weights for these statistics to generate a score.

$$computer - generatedscore \leftarrow sum_{i=0}^{4} statistics[i] * weights[i] \tag{11}$$

A system like this can be trained to approximate the score given by the user. However, overlapping-fraction and alignedness are statistics that for high values indicate bad layouts. This is a problem since training weights with the possibility of negative weights increases the complexity of the training and with that decreases the accuracy. Therefore these statistics are inverted so that high values imply good layouts and we can use only positive weights.
In order to compare the generated score with a user score, a user score is calculated based on the 5 closest user scores. This is done by taking the generation parameters of the layout the computer tries to score, and finding the closest 5 parameter sets with user scores, and then weighting the scores based on the distance.
An interesting note is that there is no statistic for overlapping nodes. This is due to there already being a statistic that describes the average distance between connected node pairs. These two statistics would have a big correlation and therefore only one is used.

3.2.1 Optimizing Weight

An optimization algorithm was used to find the best weights to generate the computer score. The algorithm starts with a set of weights that are all set to 0. Then for each set of weights the algorithm repeatedly increases or decreases a random weight by a random amount, whilst checking the *totalError* every time a change is made to the weights. This *totalError* is the sum of the difference between the score generated based on the weights and the user score to the power of 3

$$totalError \leftarrow sum_{i=0}^{nrUserScores} |userScore - computerScore|^3 \tag{12}$$

If after increasing or decreasing a weight the totalError is more than before, the weights are reverted to the previous values. After many iterations a set of weights could get close to approximating the user scores. However, if we do this for just one set of weights, the weights might get stuck in a local optimum and never reach the best set of values. This could be solved by running the algorithm over and over again until every possible set of weights has been tried, but that would not be better than brute forcing it, which is too expensive. Therefore the algorithm meets somewhere in the middle. It runs 100 times, after which it saves best the 50. Then it runs again a 100 times, but instead of starting with weights all at 0, it starts with a random set of weights from the previous top 50. By repeatedly doing this the algorithm will be quicker than brute forcing it and runs less risk of getting stuck in a local optimum.

3.2.2 Normalizing Data

The user scores are not equally distributed across all possible scores. This means that if the weights would be trained on the user scores without preparing the data our weights might be set to always predict the most common score. This way the average error per prediction would be low, but it would not really work as a way to synthesize user scores. To avoid this the data is prepared to have an equal distribution of scores, by leaving out some of the user scores of which there are too many. This process is also known as undersampling. Unfortunately for the highest scores there simply is not

enough data to achieve a perfectly uniform distribution without leaving the majority of the data out. Due to this the averages of the scores are 4.5 and 4.2 for all the data and under-sampled data respectively.

Figure 4: User score distribution

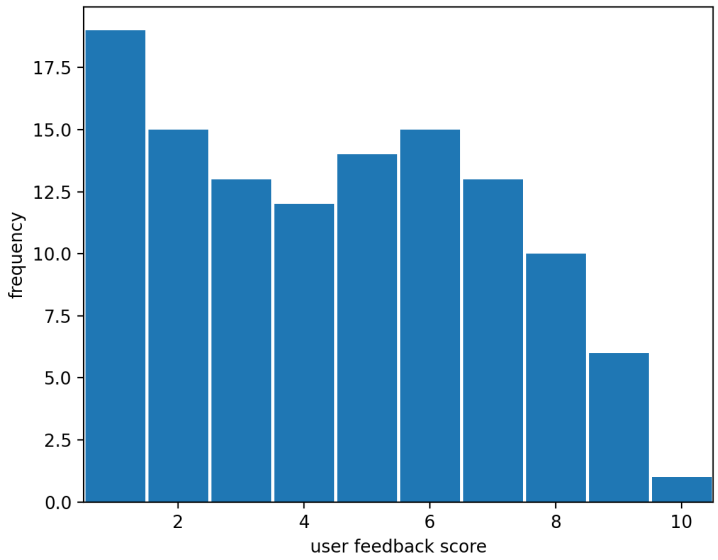
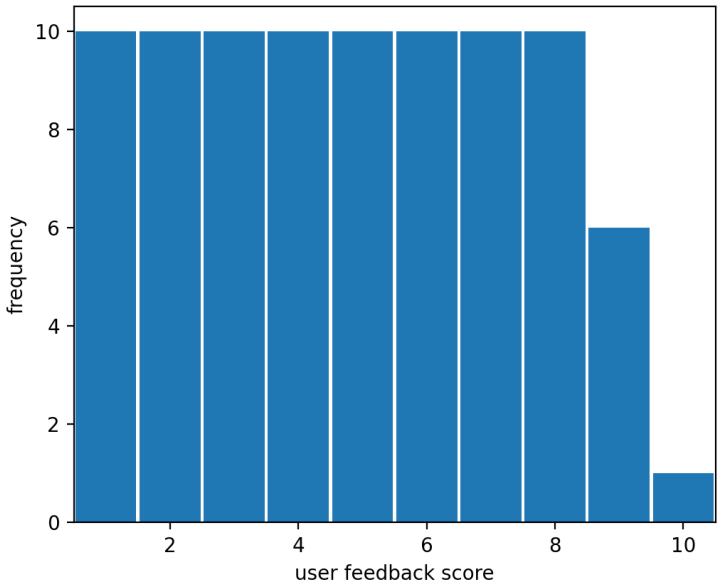


Figure 5: User score distribution after undersampling



3.3 Objective Function

A PSO needs an objective function to give every particle in the algorithm a score based on its location. As stated previously the user score is calculated by combining the user score of the closest 5 parameter sets that have a user score, giving each score a weight based on the distance to the corresponding parameter set. In order to calculate a computer-generated score, the same principle applies, but instead of combining the

computer-generated score, the computer-generated statistics are combined and from that, the score is generated. Given that two of the thirteen parameters of the generation algorithm are binary, a sigmoid function is used to set these values to 1 or 0 when calculating the scores.
Two objective functions were used for this research:

$$objectiveScore \leftarrow calculateUserScore(parameterSet) \tag{13}$$

$$objectiveScore \leftarrow calculateUserScore(parameterSet) + calculateComputerScore(parameterSet) \tag{14}$$

Given that the computer score is supposed to be an approximation of the user score, we could halve the result of the second objective function (equation 14) to compare it to the result of the first objective function (equation 13).

3.4 Parameter Optimization

In order to get the best results from the bat algorithm, the hyperparameters need to have the right values. In order to find these first I manually searched for the general factor of values. Then once those were found, the algorithm ran repeatedly with randomized hyperparameters around the initial found values. Per set of random parameters the algorithm ran 10 experiments so that the the average result could be taken, instead of the result of just one run. This resulted in a list of hyperparameters along with the average result of 10 experiments. The best performing hyperparameters were used for the final experiment.

3.5 Alignment Tool

After generating a layout for a diagram, the nodes are usually not aligned, even when $alignment_{enable}$ is set to true. This is due to the springs algorithm having to chance to ignore $alignment_{enable}$, to avoid being unable to move nodes that are aligned on both axis. Therefore a simple tool is used to align the nodes afterwards. This works by checking the difference in x and y coordinated between every pair of nodes. If the difference in coordinate in less that 40 pixels, one of the nodes is moved to match the coordinate. Iterating over every pair of nodes is done by iterating over a list of nodes, and for each node iterating over the following nodes in the list. By moving the nodes of the second iteration loop to match the nodes of the first iteration loop, it is made sure that pairs only get matched once.
This tool is completely separate from the rest of the layout algorithm and none of the parameter optimization or user score takes this tool into account. It is purely made to show the potential of implementing a similar tool into the main algorithm.

4 Results

4.1 computer-generated Scores

The performance of the computer-generated scores can be seen in figures 6, 7 and 8. As seen in figure 8 the computer-generated scores are often within 1 point of the user score, however from figures 6 and 7 it is clear that even though the user scores in the dataset are distributed equally, the predictions still tend to follow a similar trend regardless of the user score. The fact that the computer-generated score is 5 most of the time, means that we cannot compare the scores of different objective functions by halving the one that takes into account both user score and computer-generated score. Instead, we could simply subtract 5 from the result.

Figure 6: Computer-generated score against calculated user score

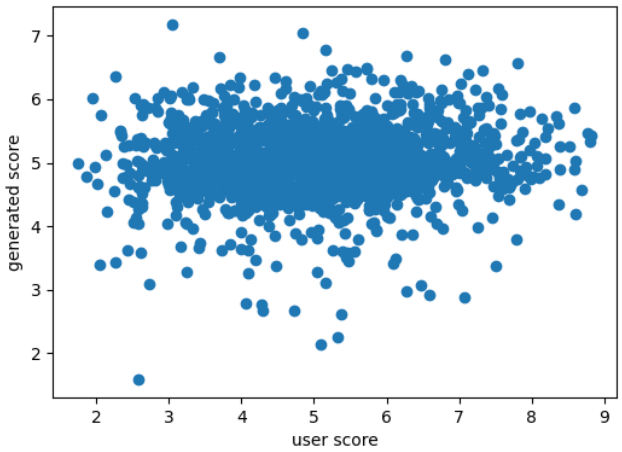


Figure 7: Computer-generated score distribution

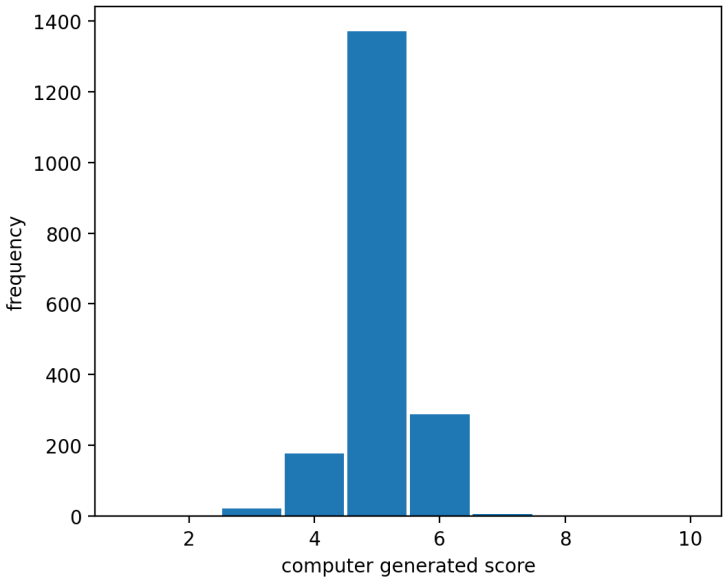
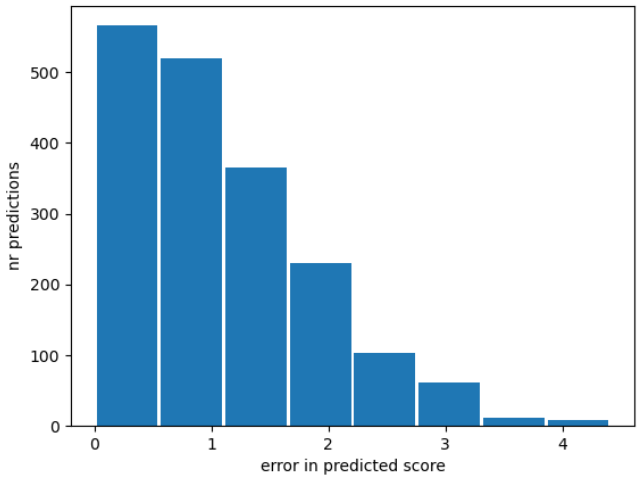


Figure 8: Computer-generated score against calculated user score



4.2 Bats algorithm

Two experiments were performed in an identical way, except for the objective function. The bat algorithm is initialized with the optimized hyperparameters and is run 10 times, with a 1000 bats and 15 iterations are done. When a new solution close to one of the best solutions is generated, instead of taking one of the best 5 bats, one of the best 50 bats is chosen to give a new solution to prevent getting stuck in a local optimum. The results of these experiments are shown in the table below. As stated previously, the scores of the objective functions cannot be compared by halving the score of the objective function that uses both user score and computer-generated score. Instead, we subtract 5 from the scores that use computer-generated data. From the table below it is clear that objective function based only on the user score leads to a result of just over 9, whereas the other objective function leads to a result of around 15. If 5 would be subtracted from the second results one could argue that that result is better than the first result since 10 is greater than 5. However, there is still a chance that in this case the computer-generated a score of 7 in which case the total result of around 15 is not better than 9. Because of this it is very hard to compare these results. However, what is clear is that in both cases the bats found a solution that has a better score than the average score of diagrams when laid out with random parameters for the layout algorithm, which was a 4.5.

Run	User score	User score and computer score
1	9.123	14.836
2	9.098	14.542
3	9.082	14.613
4	9.056	14.735
5	9.093	14.800
6	9.125	14.995
7	9.090	14.945
8	9.107	14.829
9	9.086	15.109
10	9.035	14.769

The parameter values corresponding to the best found value for each objective function can be found in the table below.

Parameter	User score	User score and computer score
$HeuristicDistance$	153	59
$force_{connected}$	47	54
$targetDistance_{connected}$	350	198
$force_{unconnected}$	75	53
$targetDistance_{unconnected}$	441	258
$align_{distance}$	261	360
$align_{enable}$	1 (True)	1 (True)
$checkOverlap_{interval}$	35	26
$checkOverlap_{distance}$	200	159
$checkOverlap_{enable}$	0 (False)	0 (False)
$rand_{chance}$	1	6
$rand_{magnitude}$	22	87
$nrIterations$	17	21

For some parameters the difference between the two values found is quite large. There could be multiple reasons for this. One could be that instead of one clear best set of parameters, there would be a numerous parameter sets that compete for the best set across the entire 13-dimensional space. Another possibility is that due to a lack of large enough data the algorithms have not found a correlation between the results and every parameter. The parameters $checkOverlap_{enable}$ and $align_{enable}$ have the same value and other parameters like $nrIterations$ and $force_{connected}$ have similar values. These parameters are most likely to have the biggest influence on the layout features that the users found important.

4.3 Final Scores

After running the experiment with the bat algorithm for both objective functions, the best result and with that a set of parameters for the layout algorithm was found for each objective function. Both sets of parameters were used to generate 25 layouts for users to rank. The results of these rankings could be found in figures 9 and 10 and the average score for these layouts is 6.1 for the parameters found only making use of user feedback, and 6.4 for the parameters found by also using computer-generated data. Whilst the result of parameters found using the computer-generated data is slightly higher than the result of only using user feedback to find the parameters, given that only 25 layouts were scored per parameter set these results could be interpreted as similar in performance. Nonetheless the average score and the distribution of scores for both sets of parameters are significantly better than the original scores gathered using random parameter sets.

Figure 9: Score distribution of found parameters by only using user feedback

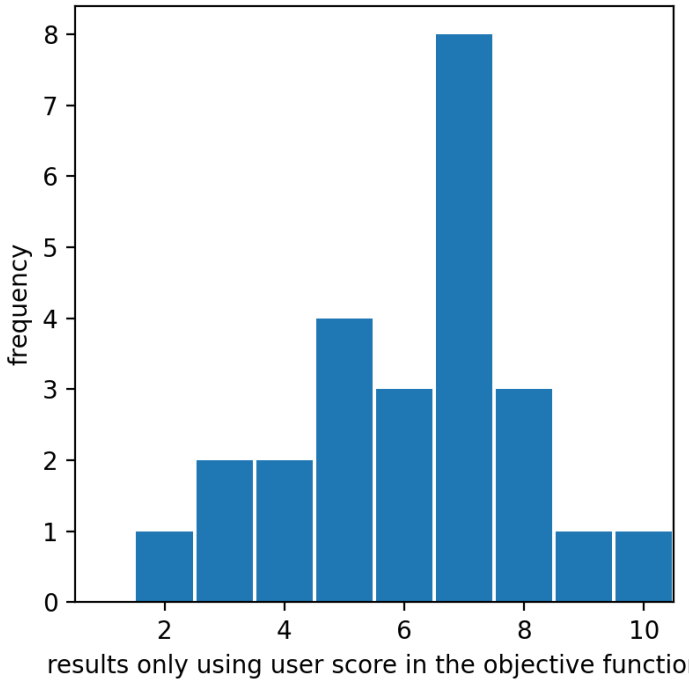
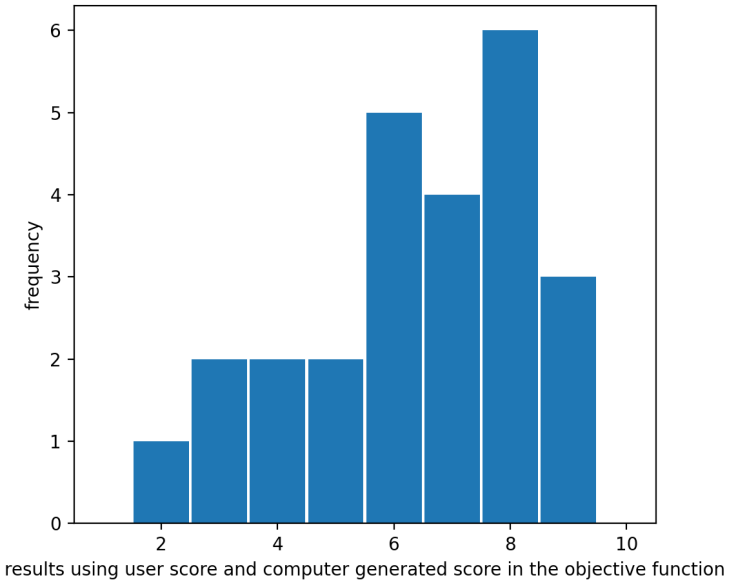


Figure 10: Score distribution of found parameters by using user feedback and computer-generated feedback



4.4 Generated Layouts

The Prose To Prototype project that this layout algorithm will be implemented in has no working auto-layout algorithm yet. Therefore the baseline layouts will be the layouts generated with this algorithm with random parameters. Given their random nature, some layouts will not be received well, but others might be.
Below are screenshots of generated layouts of four are shown. Per diagram four layouts

have been generated with different parameter sets. The first one with random parameters, the second one with the best found parameters based on solely user feedback, the third one with the best found parameters based on user feedback and computer generated feedback, and the fourth one with the best found parameters based on solely user feedback and the final alignment tool applied.
It is interesting to see how the layout based on random parameters differ drastically between some diagrams, whereas the differences in layout between the layouts generated with set parameters are consistent with each other.
Another observation is that the final layout with the alignment tool applied has a varying effect per diagram. This leads to the conclusion that whilst the working of the alignment tool does not directly depend on the parameters of the layout algorithm, indirectly it does since the generated layout has an influence on the effectiveness of the alignment tool.

4.5 Diagram 1

Figure 11: Diagram 1 laid out with random parameters

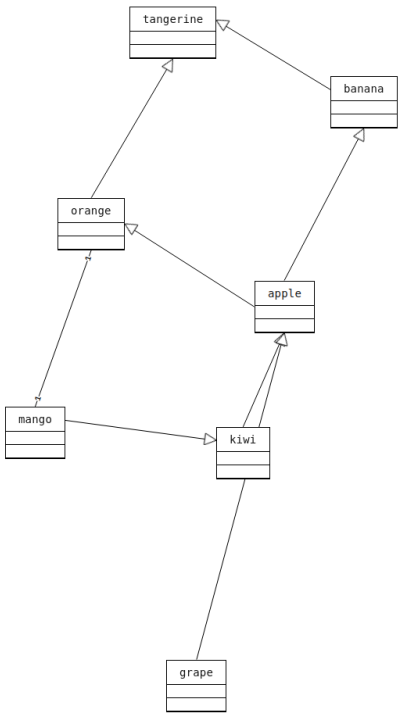


Figure 12: Diagram 1 laid out with parameters based on user feedback

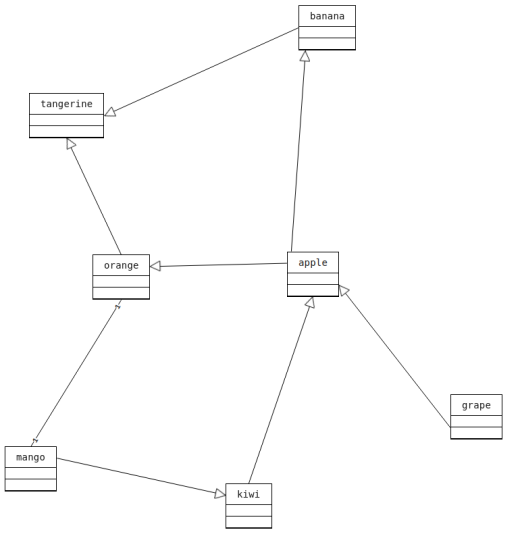


Figure 13: Diagram 1 laid out with parameters based on user feedback and computer feedback

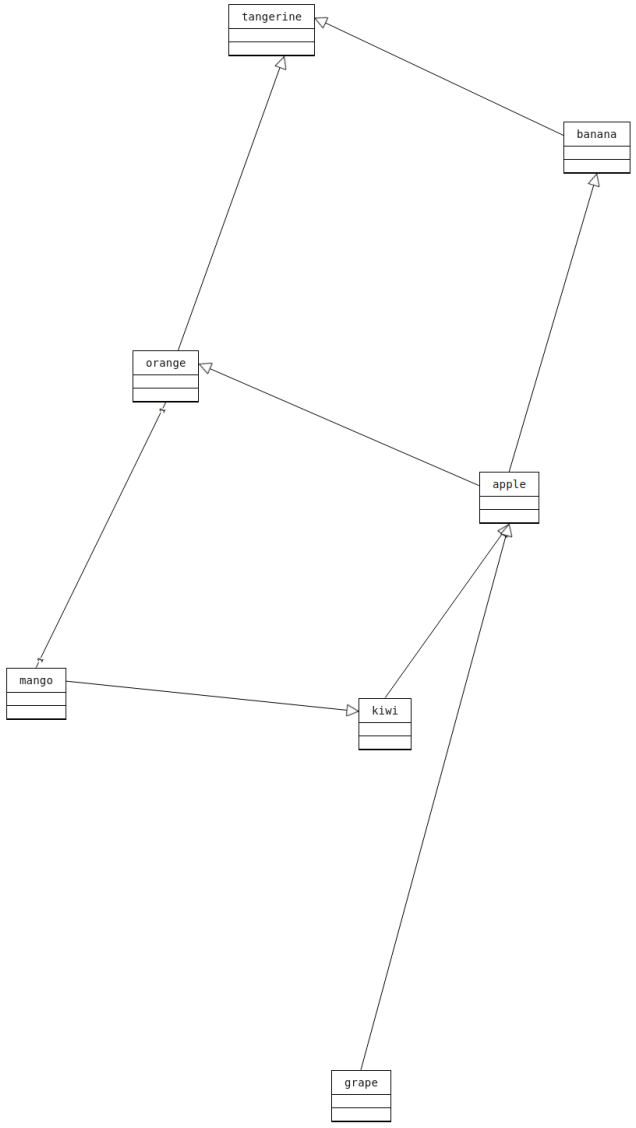
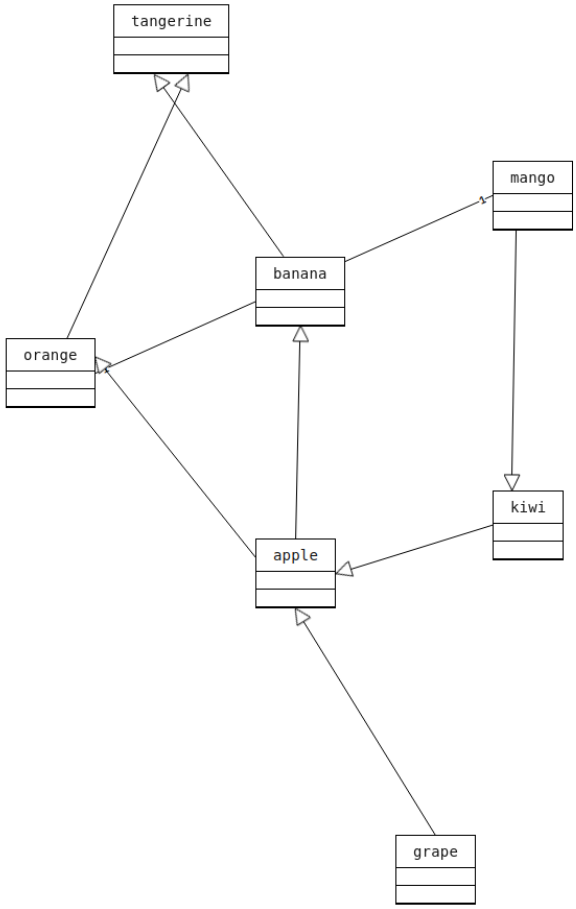


Figure 14: Diagram 1 laid out with parameters based on user feedback and alignment algorithm applied afterwards



4.6 Diagram 2

Figure 15: Diagram 2 laid out with random parameters

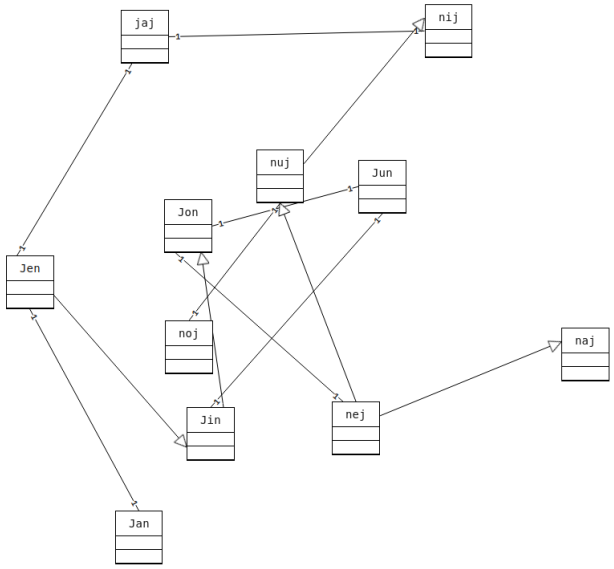


Figure 16: Diagram 2 laid out with parameters based on user feedback

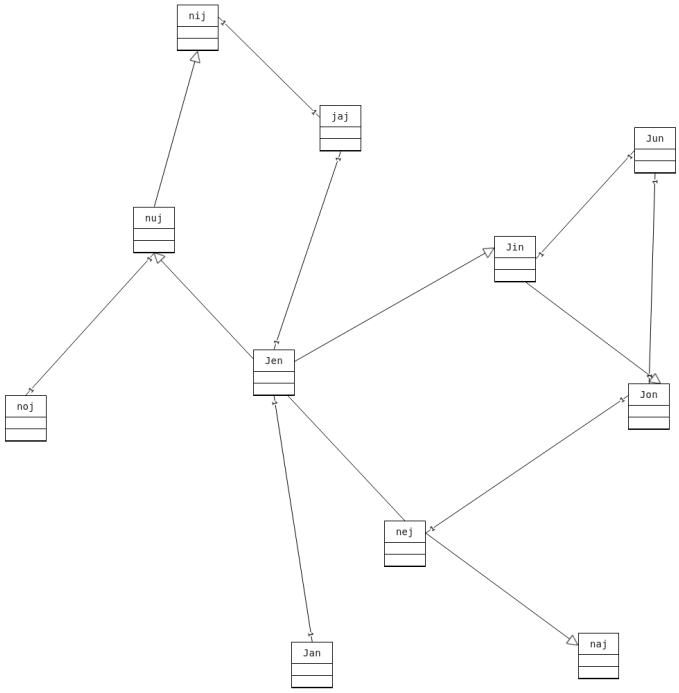


Figure 17: Diagram 2 laid out with parameters based on user feedback and computer feedback

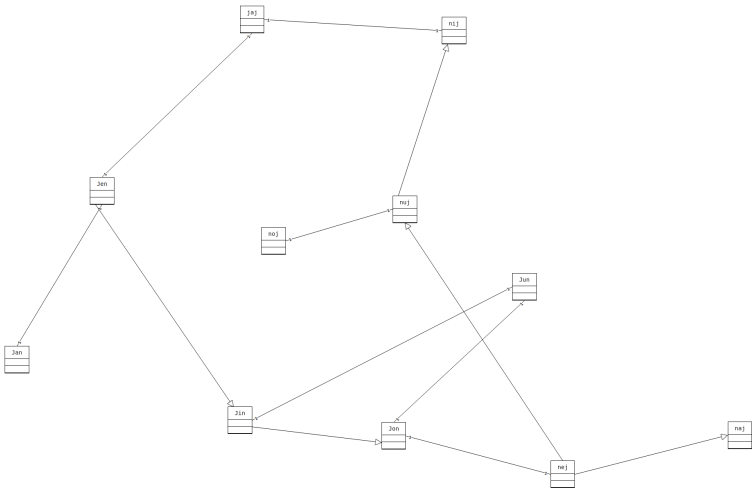
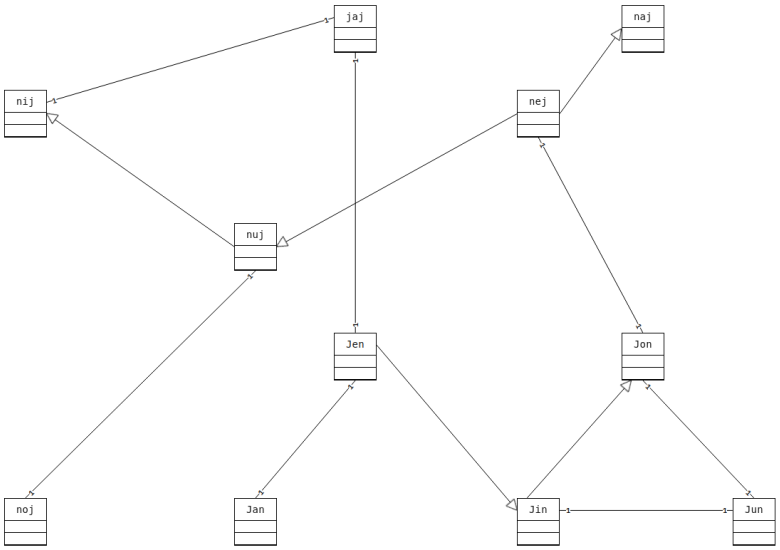


Figure 18: Diagram 2 laid out with parameters based on user feedback and alignment algorithm applied afterwards



4.7 Diagram 3

Figure 19: Diagram 3 laid out with random parameters

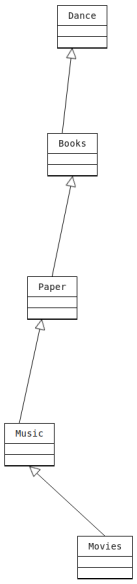


Figure 20: Diagram 3 laid out with parameters based on user feedback

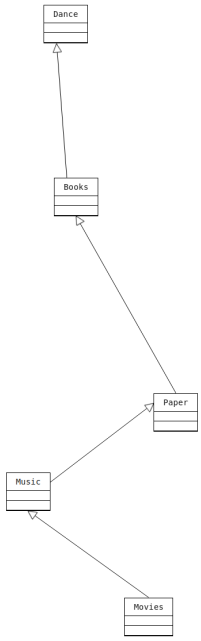
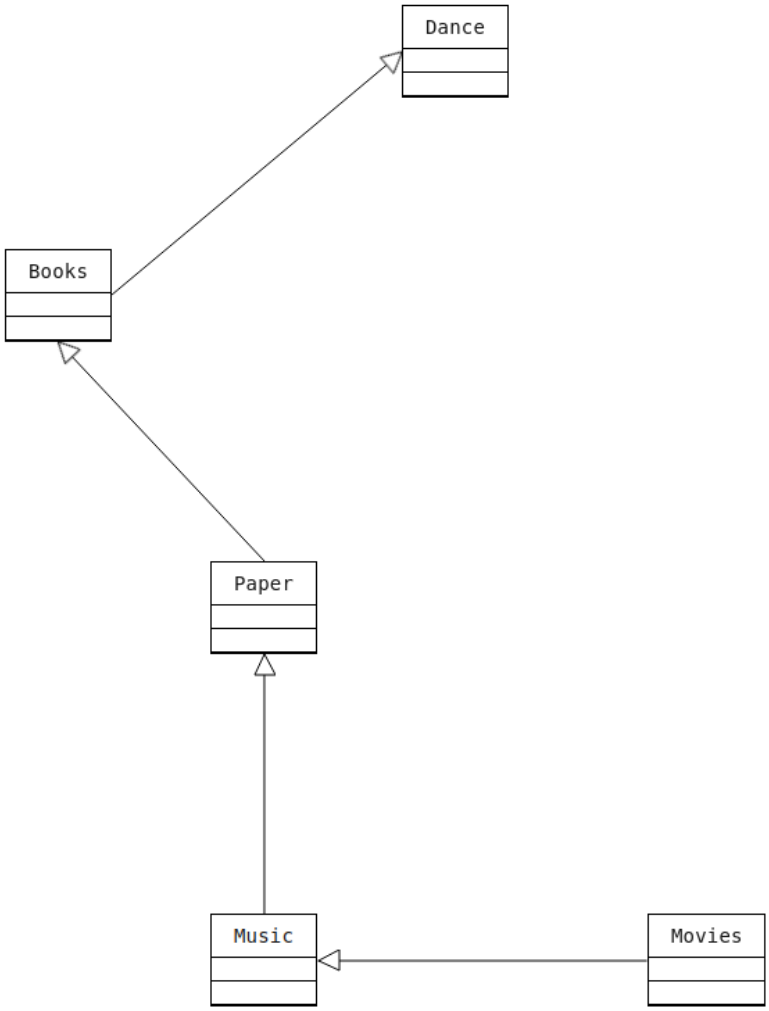


Figure 21: Diagram 3 laid out with parameters based on user feedback and computer feedback

Figure 22: Diagram 3 laid out with parameters based on user feedback and alignment algorithm applied afterwards



4.8

Diagram 4

Figure 23: Diagram 4 laid out with random parameters

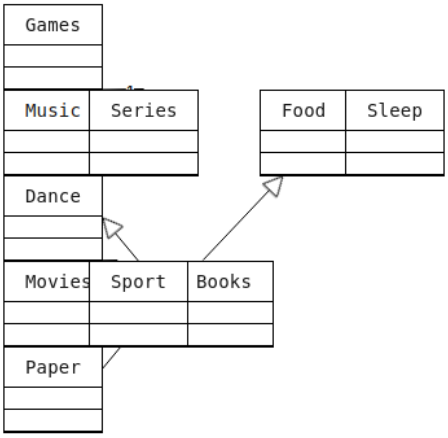


Figure 24: Diagram 4 laid out with parameters based on user feedback

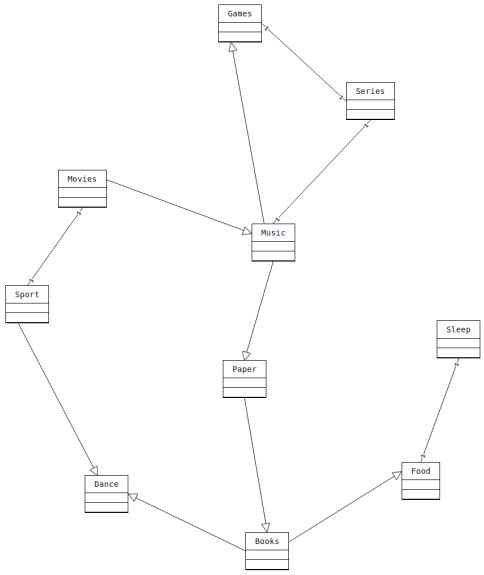


Figure 25: Diagram 4 laid out with parameters based on user feedback and computer feedback

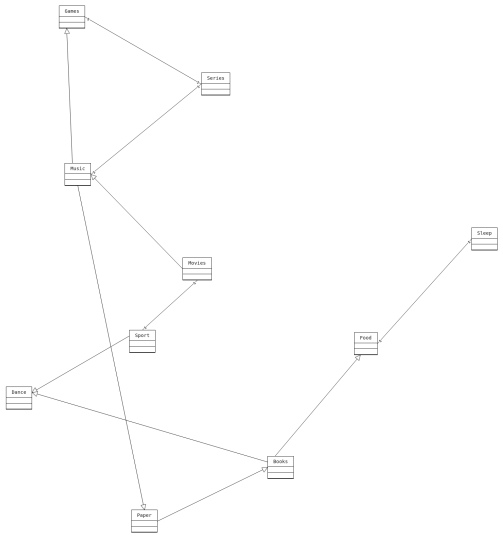
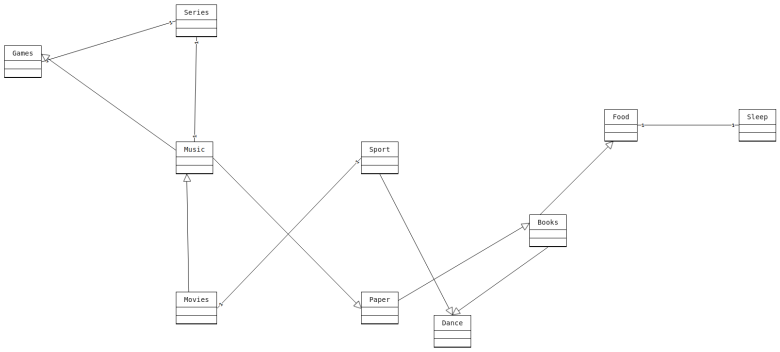


Figure 26: Diagram 4 laid out with parameters based on user feedback and alignment algorithm applied afterwards



5 Conclusion and Future Works

The results show an improvement over the starting point, which was using random parameters for the layout generation algorithm.
This shows that the selected natural algorithm can be used to increase the performance of a layout algorithm for UML system design. However, increasing the performance does not equal optimisation and it is clear that there is still room for improvement.
Due to a limited data set the algorithms most likely did not have enough data to properly train themselves. Eventhough a lack of data is not preferred, being able to significantly improve the performance of a self learning algorithm even without sufficient data shows that the potential of the algorithm. In the end the layouts generated with the parameters found by the bats algorithm improved the average score of the layouts by almost 2 points out of 10.

Even though the algorithm performs better with the achieved parameters than before,

there is still much to improve. Especially the computer-generated score system. If this would be improved the bats algorithm would not only have much more data to use, but it would be taking in an extra type of data as the computer-generated score is based on the calculated statistics of the layout. This could be good to take into account. Ideas to improve this system would be to gather more user scores to train on, or to pick a different optimizing algorithm for the weights. Another possible improvement would be to calculate more statistics of a layout and with that increase the number of weights that need to be trained.
The springs algorithm might also be improved as currently it only checks overlapping connections, not nodes, and it does not take into account that some nodes need to be positioned above other nodes.
The alignment tool could be optimized by changing the distance at which a nodes is considered aligned. It was set to 40 for this research, but perhaps future work could indicate that another set distance, or even a dynamic distance would be better.
This project was done with UML class diagrams in mind, but there are many more UML diagrams that could benefit from a similar layout function. Specifically the activity diagram comes to mind. Adding these other UML diagrams would require the addition of more types of nodes and connection to the code, along with heuristic rules describing the relation and with that the placements of nodes that share a connection. By training the system to perform better on other UML diagrams interesting results could follow.
The last point of future work is automation. Automated data gathering from test users when using this layout algorithm would result in a dataset that grows over time. This could then be used to automatically train the weights for the computer-generated scores, build a new dataset with these computer-generated scores and then rerun the bats algorithm to find a new set of parameters with an improved performance. A system like this could be used to rapidly grow a dataset when used by many people, but could also be used to gather data for individuals so that the algorithm would generate a layout based on only the individuals' preferences. This could ultimately be used in large-scale development teams where every team member has the same diagram but with a different layout so they all can easily view the graphical layout of the specification in their personal preferred format.

References

[de 07] Leandro Nunes de Castro. Fundamentals of natural computing: an overview. *Physics of Life Reviews*, 4(1):1–36, 2007.

[GG12] J. L. García-Gonzalo, E.; Fernández-Martínez. A brief historical review of particle swarm optimization (pso). 2012.

[GJR21] Martijn B.J. Schouten Michel R.V. Chaudron Guus J. Ramackers, Pepijn P. Griffioen. From prose to prototype: Synthesising executable uml models from natural language. 2021.

[GR15] Filippo Ricca Diego Clerissi Gianna Reggio, Maurizio Leotta. What are the used uml diagrams? a preliminary survey. 2015.

[HFvH] Michael Matzen Hauke Fuhrmann, Miro Sponemann and Reinhard von Hanxleden. Automatic layout and structure-based editing of uml diagrams.

[KKR96] Thomas Kamps, Joerg Kleinz, and John Read. Constraint-based spring-model algorithm for graph layout. pages 349–360, 1996.

[Lyo98] Andrew Lyons. Uml for real-time overview. 1998.

[MISK11] Lee Fook Tack Mahmud Iwan Solihin and Moey Leap Kean. Tuning of pid controller using particle swarm optimization (pso). 2011.

[VKS14] Matthew Nathal Vincent Kenny and Spencer Saldana. Heuristic algorithms. 2014.

[Yan10] CXin-She Yang. New metaheuristic bat-inspired algorithm. *Nature Inspired Coop-erative Strategies for Optimization*, NISCO 2010, 2010.