Umeå University

Efficient graph embeddings with community detection

APRIL 16, 2021

Felix Djuphammar

PUBLISHED BY FELIX DJUPHAMMAR

April 2021

Submitted as a master's thesis in Engineering Physics to the Department of Physics, Umeå University

Supervised by Martin Rosvall, Integrated Science Lab, Department of Physics, Umeå University
Supervised by Andrea Lancichinetti, Integrated Science Lab, Department of Physics, Umeå University
Examiner was Ludvig Lizana, Integrated Science Lab, Department of Physics, Umeå University

Abstract

Networks are useful when modeling interactions in real-world systems based on relational data. Since networks often contain thousands or millions of nodes and links, analyzing and exploring them requires powerful visualizations. Presenting the network nodes in a map-like fashion provides a large-scale overview of the data while also providing specific details. A suite of algorithms can compute an appropriate layout of all nodes for the visualization.

However, these algorithms are computationally expensive when applied to large networks because they must repeatedly derive relations between every node and every other node, leading to quadratic scaling. Also, the available implementations compute the layout from the raw data instead of the network, making customization difficult.

In this thesis, I introduce a modular algorithm that removes the need to consider all node pairs by approximating groups of pairwise relations. The groups are determined by clustering the network into densely connected groups of nodes with a community-detection algorithm. The implementation accepts a network as input and returns the layout coordinates, enabling modular and straightforward integration in a data analysis pipeline. The approximations improve the new algorithm's scaling to an order of $2N^{1.5}$ compared to the original N^2. For a network with one million nodes, this scaling improvement gives a 500-fold performance boost such that a computation that previously took one week now completes in about 20 minutes.

Keywords

Network, visualization, modularization, t-SNE, gradient descent

Contents

1
Introduction

Companies gather data on an increasingly detailed level as access to new types of information is continuously becoming available. Technological advancements allow collection of detailed consumer purchase data, providing extensive information about product and consumer relations. These relations form networks describing the company's products from a consumer market stand point. Each product is represented by a node with connections to other nodes generated from the data.

Generating and analyzing these networks is a service that the Umeå based company InfoBaleen provides to its clients. They help their clients interpret their data to draw conclusions about their consumers and dynamic markets that go way beyond what traditional assumptions can offer. A common method of doing this is through visualizing networks where InfoBaleen offer services that generate networks from raw data and visualizes them. By visualizing a map-like interface of the nodes, the data owner can explore their data and learn things about their products that would otherwise be difficult to observe.

As an example, imagine an electronics company that manufactures PC peripheral products. From the suppliers perspective, the most logical categorization is per product group, such as keyboards in one category and mice in another. From the customers point of view, if they just recently bought a keyboard then it is more likely that they are interested in a mouse than another keyboard. This means that recommending products from the same category is unlikely to be successful. However, which category is most likely to be of interest is not always trivial. So, how do we determine what other categories may be of interest, and how do we decide which products within these categories to recommend?

Using network visualization, this is one of the many questions InfoBaleen can help to answer. Products that are often bought together or in succession get represented by nodes in closer proximity in the map. Nodes in close proximity tend to form communities of products especially related to each other, seen as cluster formations in the map. By exploring the map, a user gains understanding of how products relate to one another within these clusters, as well as a larger overview of the relations between the clusters.

The current problem is that the majority of existing algorithms for computing the visual layout only manage small systems. As the data sets become larger, so do the networks, which results in the current algorithms taking very long time to complete due to their poor scaling nature. There is currently a shortage of generally reliable methods to compute positions in the map without making computations from each node to every other node in the network, which results in quadratic scaling in terms of network size. The field is in need of algorithms that scale better to enable using these methods on much larger networks than what is currently viable.

Existing implementations usually suffer from these poorly scaling execution times. Moreover, they are often implemented with a black box design, accepting the raw data as input and returning the complete layout. Customization is thereby difficult. By implementing only the layout algorithm with a network as input, developers can control other components in the complete visualization tool's pipeline more conveniently.

In this thesis, I propose a solution to this. I have derived a new formulation of an existing algorithm that optimizes performance by making approximations using properties of these clusters. The new algorithm scales better than the original with very little expense in accuracy. The implementation of it accepts a network as input and can return the map coordinates for all the nodes. Thanks to this, InfoBaleen can provide customizable and faster solutions to their clients with equally rewarding results.

2 *Theory*

A NETWORK represents a real world system using relations between objects. It consists of a set of network nodes that connect to each other by links with a certain weight. A common example is airports, where the nodes are the airports themselves, the links are the flight paths and the weights can be some measurable quantity like number of flights per week.

A property commonly observed in networks is a modular structure. This is when a limited subgroup of network nodes connect to each other more densely than to the rest of the network, forming a cluster, also referred to as a community or module. There can exist layers of clusters, where subgroups of clusters are related in the same way, constituting a multi-layered modular structure.

Using the airports example, one country would typically form one cluster. This assumes that the airports within the country have connections to each other, but only a select few have connections to the rest of the world. Expanding on this idea, assuming flights to closer countries are generally more common, the country modules can themselves form another layer of clusters on each continent based on the weights of the flight paths between them. The network *Airports in the world* would in this case be layered by *Airport*, *Country*, *Continent*, and *World*.

The weights in the network can be binary, meaning two nodes are either connected or disconnected. This is referred to as an unweighted graph. Similarly, the links can either be directed or undirected. A directed link means that there can exist a link from node A to B, but not from B to A, while an undirected link means that the network can always be traversed in both directions. Throughout this thesis I am considering networks that are unweighted and undirected.

The nodes also have a degree that is defined as the number of links connecting to it, or in other words, how many neighboring nodes it has. The average degree in a network is thereby a measurement of the sparsity in the network.

To better understand where in the typical work flow this thesis has its relevance, one can think of the general procedure as following four main steps. The first is network creation from a data source. The second analyzes the generated network to identify modules. The third is computing the visual layout of the entire network. The fourth and final step is visualizing the network to the end user. The algorithm discussed in this thesis aims to optimize the third step, so the other steps are done using third party libraries or simple solutions that will not be discussed in detail.

2.1 *Modularizing networks*

The algorithm in this thesis decreases execution times using modularization of networks. Modularization means that more closely related nodes in the network are labeled to belong to the same module. Below is an example showing a real use case graph before and after being divide into modules.

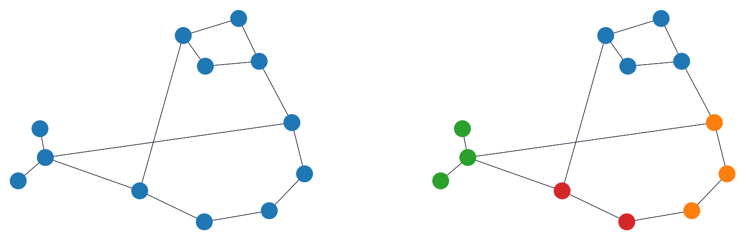

Figure 2.1 – A visualization of a network being divided into modules by color code. *Each module is given a unique color which is applied to all nodes within that module.*

I use a third party library for the process of dividing the network into clusters. Therefore, the theory behind how this division is preformed is not covered, but the understanding of its result and significance is crucial in understanding the implemented algorithm.

2.2 *The t-SNE layout algorithm*

The algorithm I expand upon in this thesis is the t-SNE [1] algorithm. Modifications of this algorithm exist, but they all share the same trade-off between execution time and accuracy. The algorithm that I propose in this thesis modifies the t-SNE algorithm in a way that improves execution times, with very minimal effects on accuracy. The first step in grasping how the algorithm works is to understand the concept of the t-SNE algorithm in its native form.

[1] Bibliography: *Visualizing Data using t-SNE*, 2008

The native form of t-SNE requires computation of a relation between every node and every other node, meaning that with all possible precomputations done properly, the algorithm is left at a complexity of N^2. Described using words, the relation between two nodes is defined by a loss function that the algorithm aims to minimize. In turn, this loss function uses a similarity function, which is related to the inverse of the distance between two nodes, along with the weight of the link between the two. By starting from a random layout and then moving nodes in the network using gradient descent, the algorithm converges toward a local minimum where the loss function is minimized. A consequence of this procedure is that the local minimum is generally not global. This means that different random initial conditions typically have different outcomes. The quality of the outcomes is comparable by the resulting loss function value.

Introduction of an approximation in the process can be done in two ways. Either the loss function is approximated and the gradient descent is performed with the exact gradient of the approximated loss function, or the gradient itself is approximated. In short, there is a choice between reaching an approximate solution exactly, or approximately reaching an exact solution. The benefits of approximating the gradient in favor of the loss function is that it opens up some possibilities for customization and versatility. To exemplify this, the algorithm can instead be used to compute several layouts, then easily using the exact gradient to fine tune the best result.

2.3 *Mathematical Definitions*

Now that you understand the t-SNE algorithm as a concept, I will introduce mathematical expressions to define the algorithm explicitly. This provides deeper understanding, which is required to follow the steps I take when formulating the approximation in section 2.5. First, I use a definition of the similarity function between two points u and v. This function returns 1 if the points are the same and decreases as the points move further apart. It is defined as

$$S(u,v) = \frac{1}{1+|u-v|^2} \tag{2.1}$$

The similarity norm of node i at position r_i is defined as the sum of its similarity to all other nodes

$$||S||_i = \sum_j S(r_i, r_j) \tag{2.2}$$

Similarly, the weight probability norm of node i is defined as the sum over the weights w_{ij} to all other nodes j. Note that in a binary graph this equates to the number of neighbors of node i

$$||w||_i = \sum_j w_{ij} \tag{2.3}$$

Using these definitions, the loss function for a node i that I aim to minimize is defined as

$$L_i = -\frac{1}{||w||_i} \sum_j w_{ij} ln\left(\frac{S(r_i, r_j)}{||S||_i}\right) \tag{2.4}$$

The full loss function of the layout is then the sum over the loss functions for all nodes

$$L = \sum_i L_i \tag{2.5}$$

2.4 *Gradient derivation*

The full derivation of the gradient can be seen in section A.1 of the appendix. The resulting expression for the gradient can be written using definitions for Attractive force:

$$A(i,k) = -2w_{ik}\left(\frac{1}{||w||_i} + \frac{1}{||w||_k}\right) S(\bar{r}_i, \bar{r}_k)(\bar{r}_i - \bar{r}_k) \tag{2.6}$$

and Repulsive force:

$$R(i,k) = 2\left(\frac{1}{||S||_i} + \frac{1}{||S||_k}\right) S(\bar{r}_i, \bar{r}_k)^2(\bar{r}_i - \bar{r}_k) \tag{2.7}$$

Then the full gradient of the loss function[2] for a node k is expressed simply by

[2] Note that both A and R evaluate to 0 when $i = k$.

$$\nabla L_k = \sum_i A(i,k) + R(i,k) \tag{2.8}$$

2.5 *Gradient approximation*

To formulate an approximation of the gradient in equation 2.8, I introduce the concept of nodes belonging to modules. I define the subset of all nodes in module number i as m_i. To help in the procedure, I also need to make an unconventional definition that reads *m of k* written as $m_{(k)}$, which defines the subset of nodes within the same module as node k. This is so that I can refer to a module from a node index. I also define the subset n_k as the nodes with a direct link to node k. With these subsets the sum in equation 2.8 can be split into three sums. The first sum only adds contributions from neighbors, since $w_{ik} = 1$ in the function A if $i \in n_k$ and is otherwise zero. The second sum adds repulsion per node that is either in the same module or a direct neighbor. The final sum adds repulsion from the remaining nodes. The upper limit N of the summations signifies summation over nodes within a given set.

$$\nabla L_k = \sum_{i \in n_k}^{N} A(i,k) + \sum_{i \in m_{(k)} \cup n_k}^{N} R(i,k) + \sum_{i \notin m_{(k)} \cup n_k}^{N} R(i,k) \tag{2.9}$$

The final sum can now be approximated over the modules by introducing necessary notations. Defining the size of a module subset $|m_k|$ as the number of nodes in module k we can define the position of a module as its centroid:

$$\bar{r}_{m_i} = \frac{1}{|m_i|} \sum_{j \in m_i}^{N} \bar{r}_j \tag{2.10}$$

Redefining the similarity norm to account for the modules we sum over all nodes within the same module, then add contribution from each module by the similarity to its centroid, scaled by its size. The upper limit M signifies summation over modules rather than individual nodes.

$$||S||_i = \sum_{j \in m_{(i)}}^{N} S(\bar{r}_i, \bar{r}_j) + \sum_{m_j \neq m_{(i)}}^{M} |m_j| S(\bar{r}_i, \bar{r}_{m_j}) \tag{2.11}$$

Using this definition of the similarity norm of a node, the similarity norm of a module is defined such that

$$\frac{1}{||S||_{m_i}} = \frac{1}{|m_i|} \sum_{j \in m_i}^{N} \frac{1}{||S||_j} \tag{2.12}$$

The gradient can now be approximated by

$$\nabla L_k \approx \sum_{i \in n_k}^{N} A(i,k) + \sum_{i \in m_{(k)} \cup n_k}^{N} R(i,k) + \sum_{m_i \neq m_{(k)}}^{M} |m_i| R(m_i,k) \tag{2.13}$$

The final step is to account for overlapping effects with neighbors inside other modules being counted twice. As neighboring nodes will be closer, the module averages that include neighbors will be positioned closer than they should and will be weighted higher, resulting in an overestimation of repulsion between clusters. To compensate this effect, the module sets are reduced by subtracting the nodes neighboring to k from the module subset. The notation for a reduced module m_i with respect to node k is introduced as m_i^k and is defined by

$$m_i^k = m_i - m_i \cap n_k \tag{2.14}$$

Using the reduced subset of the module, the complete form of the approximated gradient becomes

$$\nabla L_k \approx \sum_{i \in n_k}^{N} A(i,k) + \sum_{i \in m_{(k)} \cup n_k}^{N} R(i,k) + \sum_{m_i \neq m_{(k)}}^{M} |m_i^k| R(m_i^k, k) \qquad (2.15)$$

Descriptively, the gradient computation adds the exact attraction and repulsion for its neighbors and for nodes within the same module. The repulsion from the rest of the system is then approximated from the centroid of each neighbor reduced module, weighted by the number of nodes in the reduced module.

2.6 *Expected Scaling*

To estimate the complexity of the algorithm, I want an expression in terms of network size, in this case the number of nodes N and the number of modules M. In order to derive this expression I need to make some assumptions of the network itself. Assuming all modules have the same number of nodes I can define a variable for the cluster size as C_S. This means that the total number of nodes in the network equates to the product between the number of modules and their size. Division by the number of modules gives an expression for C_S as shown in equation 2.16.

$$C_S = \frac{N}{M} \qquad (2.16)$$

Referring back to equation 2.15 I can estimate the complexity of the sum over a module by the number of nodes within that module, which is then C_S, and the sum over all modules as complexity M. I discard the sum over Attractive forces as it is unchanged compared to the exact gradient. I also assume the majority of neighbors to a particular node are within the same cluster, as the contrary would oppose the concept of a cluster in itself[3]. Using these approximations along with equation 2.16 I can formulate an expression for the expected order of complexity $\mathcal{O}$ of the gradient.

[3] This would be like saying a node is most related to something, while simultaneously having evidence that it is even more related to something else. This can only happen if the network is very scattered, which in turn means that the network is ill posed to begin with.

$$\mathcal{O}(\nabla L_k) = \frac{N}{M}\mathcal{O}(R) + M\mathcal{O}(R) \qquad (2.17)$$

When computing the gradient for the full system the gradient will be computed for every node k so the complexity of the full gradient is simply

$$\mathcal{O}(\nabla L) = N\mathcal{O}(\nabla L_k) = N\mathcal{O}(R)(\frac{N}{M} + M) \qquad (2.18)$$

Before finding the explicit complexity of R from equation 2.7, I note that the complexity of the function stems from the similarity norm expression in equation 2.11. This can be precomputed for all nodes and modules and does not need to be updated for each individual nodes. Given precomputed values of the norms, the complexity of R within the sum itself reduces to order 1. Using this I can rewrite equation 2.18 by adding the complexity of the norm precomputation for all nodes as a separate term.

$$\mathcal{O}(\nabla L) = N\mathcal{O}(||S||) + N(\frac{N}{M} + M) \tag{2.19}$$

The similarity function S has a complexity of order 1 which means that the order equation 2.11 is given by equation 2.20, using the same previously applied procedure for the two sums.

$$\mathcal{O}(||S||) = \frac{N}{M} \cdot 1 + M \cdot 1 \tag{2.20}$$

Inserting the order of the norm computation into the expression for the order of the gradient approximation in equation 2.19 I arrive at the final form shown in equation 2.21.

$$\mathcal{O}(\nabla L) = N(\frac{N}{M} + M) + N(\frac{N}{M} + M) \tag{2.21}$$

The final form has one term for precomputing the norms and one for gradient computation. The same procedure applied to the exact gradient results in $N^2 + N^2$ as the sums are over all nodes individually. Both expressions have a common factor of $2N$ that can be discarded when comparing the two. The improvement in complexity from the exact gradient to the approximation is then the transition shown in equation 2.22.

$$N \rightarrow \frac{N}{M} + M \tag{2.22}$$

To formulate a hypothesis for how the algorithm should scale, I use the equilibrium ratio where the number of modules is equal to the number of nodes within each module, meaning that $M = C_S = \sqrt{N}$. Given these circumstances the expected change in scale is

$$N \rightarrow \frac{N}{\sqrt{N}} + \sqrt{N} = 2\sqrt{N} \tag{2.23}$$

Dividing by N, I obtain the relation I expect to observe between the execution time using the exact gradient T_e and the approximated gradient T_a shown in equation 2.24.

$$T_a \approx \frac{2}{\sqrt{N}} T_e \tag{2.24}$$

Taking into consideration the practical implications of the assumptions made in this section, I expect this expression to provide a decent approximation to the execution times. As the two algorithms will not contain the exact same number of operations, I expect to see some deviation due to slightly different proportionality factors.

3
Implementation

Using the approximation in equation 2.15 I can now implement the algorithm for verification. As the purpose of this algorithm is to improve execution times, it is most sensible to use a programming language that is designed for fast operations. However, since the language used does not affect the scalability of the algorithm, I choose to implement it in a comfortable environment and familiar language to test the algorithm's scaling rather than its actual execution time.

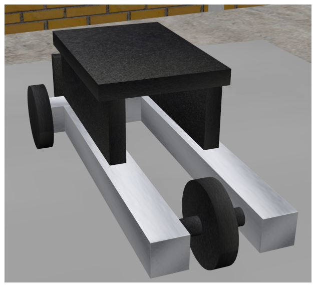

Figure 3.1 – A snapshot of a component based construction in MiroSimulint. *This will be referenced to as a robot further on in the thesis.*

When starting this, I am in parallel working with an environment called MiroSimulint designed for simulating rigid body mechanics of objects built using a component based modular system. This system is the perfect starting point for testing the algorithm by reducing the 3D rigid body objects to a 2D space using their components as nodes and their joints as links in the network. It is perfect since it is the most simple and intuitive form of dimensional reduction, so it is very simple to observe if the resulting layout makes sense when comparing to the original object.

The starting point of the program itself is inevitably to read a defined network from a file and create a graph object. Since the first networks I plan on applying the algorithm on are from the robot, I create a function to export a graph of its components as a .json[1] file. I then create a stand alone script for reading a graph from a .json file which then generates the layout. This way, any other network stored as .json can use the same scripts, and the same script can easily be extended to support other file types. I will then be able to generate larger artificial networks for further testing, once the results from mapping the robot are satisfactory. I use an existing script to generate these networks with suitable sizes and properties.

[1] https://www.json.org/json-en.html

This writing and reading of files is implemented in a *Graph* class along with other operations related to the graph itself. I then create another class for visual representation of the graph and all operations related to the visualization process. Since the intended use of the algorithm is not related to the visualization process itself, I simply use the Python library matplotlib[2] in my implementation which has sufficient functionality to determine if the algorithm works as intended, but it is not a preferred visualization tool for this kind of data. The algorithm itself is implemented in a script called *NodeMap* which performs the gradient computation and sequential stepping procedure.

[2] https://matplotlib.org

3.1 *Code Description*

In this section I describe the mechanisms of the code in more detail. A class diagram can be seen in figure 3.2 and further descriptions of the main functionalities in each class are described in the following subsections.

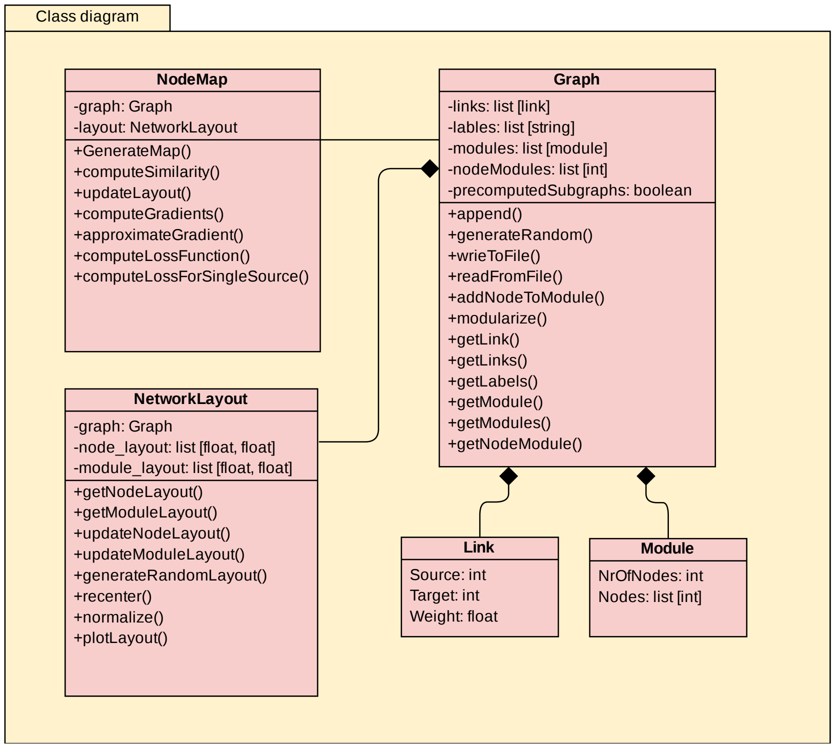

Figure 3.2 – Class diagram of the implementation. NodeMap is initiated through the GenerateMap function. It accepts a filename and creates a Graph object that is read from the input file. It then creates a NetworkLayout object based on the Graph object. By computing the gradients it updates the NetworkLayout object to minimize the loss function.

3.1.1 *Graph*

I designed the *Graph* class so that it stores a list *labels* of unique node names when a graph file is read. I store the links as the pairwise indices that the connected nodes have in the *labels* list. Adding a new link in this way is done through the *append* function which accepts a link as input. If the input is a list of links, the function calls itself recursively with each element in the list. I designed the function in this way because it allows the user to append links in any form of nested lists and the function will be able to traverse the data structure, however deep it is. The user can also choose if the link is symmetric, which appends a link between the same nodes but in the opposite direction. This is a precaution for potentially directed graphs in the future.

The *Graph* class also contains function handles to read from or write to a file. Writing to a file will always export as a .json file, but the reading function has routines for both .json and .dat file types.

I also implemented the modularization routine as a function call within the *Graph* class. This function creates an InfoMap object and appends its links to the InfoMap object. Then it runs the InfoMap search algorithm and stores the resulting modules as a set of lists with nodes belonging to the same module. When requesting a module from a *Graph* object, a structure with the total number of nodes in the module and a list over all nodes within the module is returned.

3.1.2 NetworkLayout

My intention with storing the layout as its own object is to keep the graph implementation itself as intact as possible. This enables the program to in theory be compatible with other graph implementations as well. It is written in a dedicated class so that the layout of the nodes and the layout of any potential modules and layers of modules can all be stored within the same object.

I create a *NetworkLayout* object with a graph as input, from which it allocates lists for node and module coordinates. I can then generate a random layout, which will place any clusters randomly and then place the nodes belonging to each cluster around their cluster coordinates. I also have functions to recenter the layout so that the center of mass is positioned at the origin, as well as a function to normalize the layout, meaning it is rescaled to fit the unit box where all coordinates have an absolute size at most 1.

4
Results

THE FOCUS of this thesis is improving the scaling of the native algorithm, though it is equally important that the approximations made preserve a desirable level of accuracy. If the resulting layouts do not reflect the expected outcome, then the algorithm is not viable in practice no matter how well it scales. In the following subsections I will first show examples of the resulting layouts to confirm that the results are of a satisfactory standard. I will then show how the algorithm performs in terms of scaling.

4.1 *Accuracy*

As the entire purpose of network visualization is to give a user visual understanding of something with higher dimensionality, the example using the robot is perfect to determine whether the results provide an intuitive map over the components used to build the robot. In figure 4.1 the original robot is displayed next to its corresponding map, with indicators of which component clusters are visualized as a module in the map.

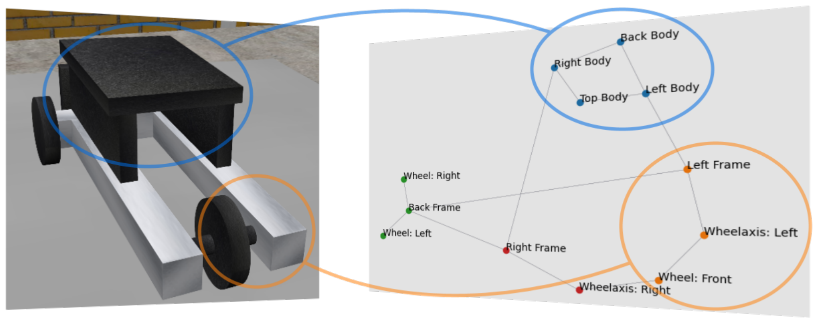

Figure 4.1 – A printscreen of the component based robot being mapped onto its 2D layout. *Nodes are annotated using the components internal names within the program.*

To test larger networks I generate benchmark networks with parameters of degree and cluster size by my choosing, using the script mentioned earlier. To verify that the resulting layout is intuitive for larger networks I test using networks with a significant mixture rate between clusters and confirm that the clusters do not overlap. The resulting layout can be seen in figure 4.2. Here it also becomes clear why better suited visualization tools are preferred, as the links become increasingly difficult to distinguish in larger networks.

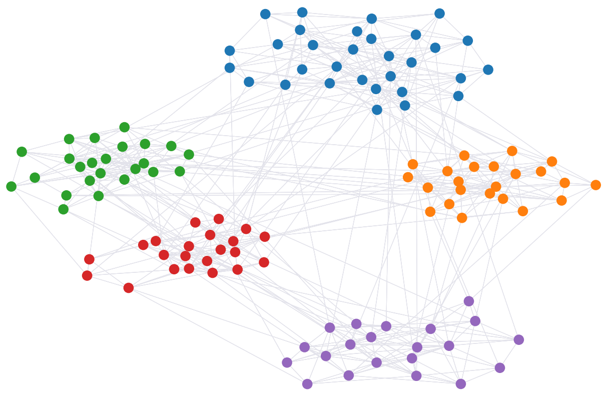

Figure 4.2 – The resulting layout of a benchmark network. *The network used consists of 5 clusters. Each node will on average have roughly 2 links to other clusters and at least 4 to nodes within the same cluster.*

4.2 *Scaling*

The scaling of the modularized algorithm is the core result in this thesis. To generate consistent results I create a script to generate benchmark networks of varying sizes. I use cluster sizes in the range $[10, 40]$ and set the total number of nodes to the square of the cluster size. This sets equally optimal circumstances for each network size so that any bias due to other effects is kept close to constant.

To measure execution times I run the modularized algorithm using 10 differently randomized networks of each size and iterate over each of them 256 times, so that each data point is the average of 2560 iterations. For the native algorithm I reduce the number of iterations per network since execution will take significantly longer to complete.

It is also necessary to compare the numerical data with the expected scaling of the algorithms. I create an analytic line for each of the algorithms by finding a coefficient k to the highest order term in the expected scaling function f such that it equals the first data point in the measured data, as seen in equation 4.1. Here T is the measured time, N_0 is the smallest network size used, f is the scaling function, and k is the parameter I find from the equation. It can then be observed how this line compares to the numerical data for larger networks. I compare execution times for the Native t-SNE algorithm with expected scaling of N^2 and the times for my approximated Modularized t-SNE with expected scaling $N^{1.5}$.

$$kf(N_0) = T(N_0) \tag{4.1}$$

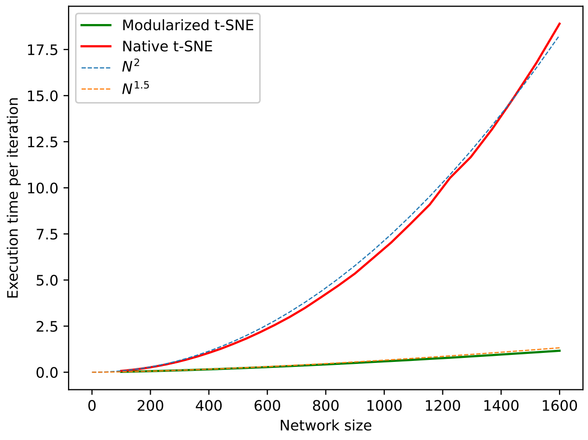

Figure 4.3 – Scaling of the modularized and native algorithms. *Analytic functions assume zero overhead by setting execution time at $N = 0$ to zero.*

Recalling the hypothesis in section 2.6 I can test its validity. I apply the scaling factor from equation 2.24 to the execution times achieved by the native algorithm. This produces a line that can be compared to the execution times of the modularized algorithm which is seen in figure 4.4.

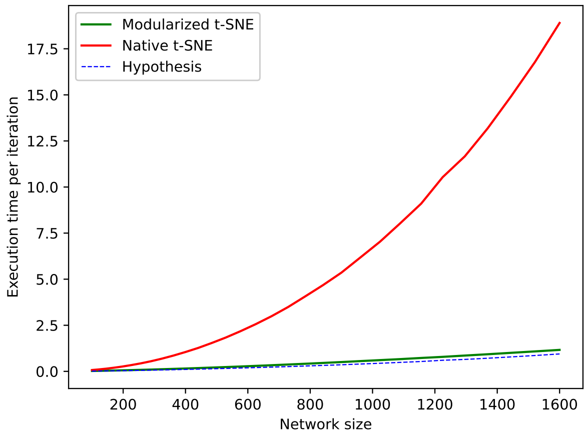

Figure 4.4 – Measured execution times along with times expected by hypothesis. *The Hypothesis line shows the times of the native algorithm rescaled by the factor presented in the hypothesis.*

The hypothesis assumes the networks meet the condition $M = \sqrt{N}$ which is the case for the first set of networks used so far. To see the effects that other cluster sizes have on the scaling I create new sets of networks. One set is created with a constant cluster size of 20 where the number of clusters increases proportionally with the network size. The second set I create using a fixed number of 20 clusters, so that the size of each cluster increases proportionally with network size. Both sets will align with the original set at $N = 20^2 = 400$ and then scale differently.

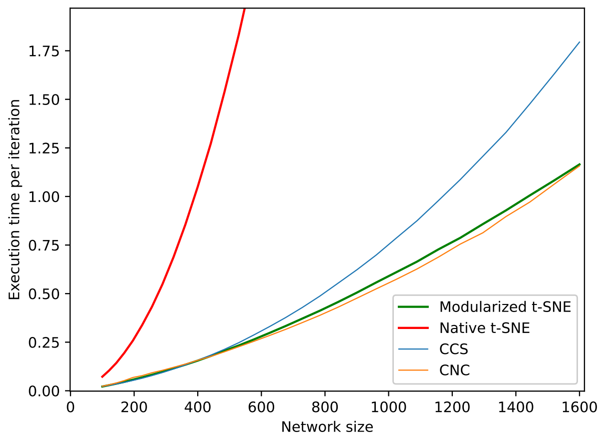

Figure 4.5 – Scaling of the modularized algorithm with different network types.
CCS: Constant Cluster Size
CNC: Constant Number of Clusters

The resulting graphs can be seen in figure 4.5. The graph showing execution times for networks with constant cluster sizes evidently loses the scaling benefits. This is expected as the limiting case of this is essentially a network where the nodes are small clusters, which requires further layering to improve scaling further. The graph showing the networks with constant number of clusters is in fact faster directly after the equilibrium point of $N = 400$. This is explained by the computation to consider an individual node being less demanding than when considering an entire module. When the network size gets larger however, the graph starts trending back toward the times using the original set. This is simply because the clusters start to get large enough that the number of nodes that are handled individually becomes significantly impactful on execution time.

5 Discussion

5.1 Weaknesses

Though the results are overall successful, the algorithm as a whole still has certain drawbacks. Here I elaborate on a selection of these and in the following section 5.2 I propose possible solutions.

5.1.1 Local Minima

As discussed earlier, the general approach of gradient descent suffers from the risk that there exists several local minimum, such that the minima it reaches is not global. When observing figure 5.1 this effect is visual for the rightmost cluster, colored in brown. The cluster is only connected to the two clusters on the very opposite side of the network, but the cluster does not move closer to these clusters due to the bulk of the network repelling it in the opposite direction.

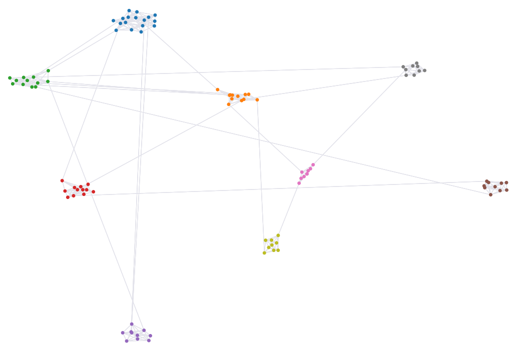

Figure 5.1 – A sparse network which stabilizes at a poor local minima. *The steady state reached depends on the initial randomized layout.*

It can also be observed that the two links to the brown cluster overlap each other. This means that a translation of the cluster by a reflection would retain the internal structure of the cluster, but the links no longer overlap such that they become slightly shorter, giving a slight decrease in the loss function.

Performing the same computation with the same network but using a different random seed to generate another initial layout yields the result shown in figure 5.2. This result is both more comprehensible upon inspection and the loss function evaluates to 712.75 in comparison to 715.43 in figure 5.1. Simply by observation it is clear that clusters with a connection are positioned closer to each other in figure 5.2 than in figure 5.1. Especially noteworthy differences between the two figures are the repositioning of the red and green clusters in relation to the brown cluster, as well as the position of purple cluster in relation to the green and blue clusters.

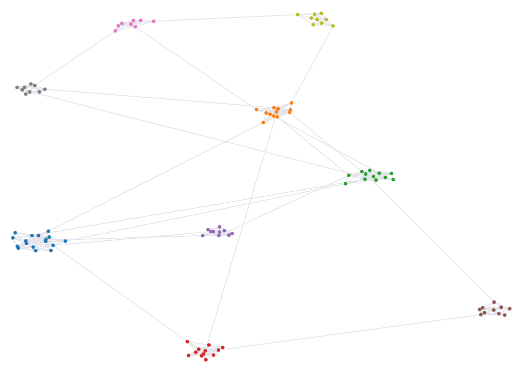

Figure 5.2 – The same network as in figure 5.1 but with different random seed. *The final state at a local minima with a lower value of the loss function.*

5.1.2 *Outliers*

A drawback of the current algorithm is the way it manages disconnected networks. In the current implementation using single layer modularity, every node in the brown cluster in figure 5.3 will evaluate their repulsion from each cluster in the rest of the network. Since the placement of the brown cluster relative to the rest of the network is insignificant for the end result, then it does not matter if it is moved slightly due to more approximations. The significance of this is that large networks that may consist of several disconnected regions do not take advantage of the fact that they are disconnected.

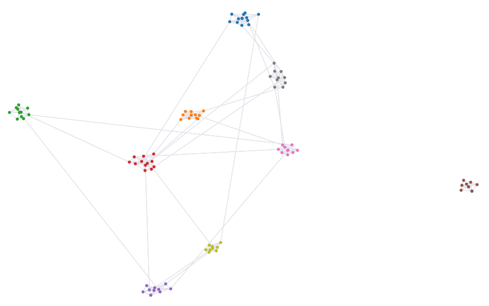

Figure 5.3 – A sparse network with a disconnected cluster. *Computations to and from the disconnected cluster are superfluous.*

5.2 *Improvements*

Regardless of how much I manage to improve the scaling and retain the precision of the original algorithm, there will always exist possibilities of improvement. In the following subsections I mention possible methods to overcome the issues mentioned in the previous section.

5.2.1 *Accuracy*

The issues with poor local minima is something that is inherited from the original algorithm, rather than something being introduced by the approximation. However, in a fully developed product, this issue should be managed in some way. I consider a couple of options for possibly solving this by generating a better starting point than simply randomizing, since it then is more likely that the minima that the algorithm converges to has a lower loss function value.

One solution that I propose is to compute the layout in 3D initially, allowing *"twisting and turning"* of the layout that cannot be done in a 2D space. Once the 3D layout starts to stabilize I would project the layout onto a 2D space using a PCA or similar algorithm. I would then use this projection as a starting point for iterating over the 2D layout until completion. This way, the more related clusters will start closer to each other and in a more suitable orientation when iterating in the 2D space.

Another solution to the issue that I consider is to calibrate only the centroids of the clusters independently before moving individual nodes. The poor local minima issue arises in this example due to weak attraction between clusters in sparse networks. By introducing a new graph between modules, I could potentially overcome this problem. If the weights are proportional to the number of links between each cluster, it would be possible to convert the links to binary form based on some threshold, preferably related to the degree of the modules to avoid very dense or very sparse graphs. Iterating over this smaller graph would then give a better starting point in terms of module-to-module positioning. The algorithm itself follows the same procedure and implementation requires mainly a creation of a step wise solver.

5.2.2 *Efficiency and scaling*

The current implementation uses only single layered clustering. Clusters can themselves possess relations such that there exists several layers of clusters within the network, as was shown in figure 5.3. The mathematical approximation created for this algorithm has no limitation to prevent it from being used recursively on layers of clusters. I would describe an implementation of this from the perspective of an individual node as it taking each close by node into consideration individually, then adds repulsion from nearby clusters on a per cluster basis, then groups distant clusters into larger clusters to approximate their combined repulsion. This recursive solution solves the issue presented in figure 4.5 where the networks with constant cluster size starts to scale poorly.

6
Conclusion

In this thesis, I have derived and implemented a new network layout algorithm. The algorithm is confirmed through both theory and practical testing to improve scaling from N^2 to $2N^{1.5}$. This result opens up future possibilities to create better performing visualization applications. They will be able to support modifications of much larger source networks and still generate updated layouts in real time. The modular design also enables customization possibilities, such as computing the layout only for the current level of zoom in the visualization tool, or using different network generation techniques depending on the data.

The scaling of this algorithm takes the current state of graph embedding one step forward in terms of performance of large data applications. The testing results show that the execution times drop significantly even for small networks, and the benefits only become larger as the networks increase in size. The modular nature of the implementation itself also enables the user to choose the most suitable option of network generation algorithms for the data of the specific task. It ensures that any use of the implementation can be replaced with other layout algorithms possessing another set of strengths and weaknesses, better suited for a particular task. This avoids the scenario where users and developer become committed to certain software.

A
Appendix

A.1 Gradient Derivation

In this section I derive the exact gradient of the loss function, that I approximate in the thesis.

A.1.1 Variable Definitions

$\bar{r}$: Layout coordinates
w : Weight of link
S : Similarity function
L : Loss function

A.1.2 Function and Operation Definitions

The similarity function between nodes u and v is defined as:

$$S(u,v) = \frac{1}{1 + |u - v|^2}$$

Note: $S(u,v) = S(v,u)$
The similarity norm of node i is defined by:

$$||S||_i = \sum_j S(r_i, r_j)$$

The weight probability norm is defined as:

$$||w||_i = \sum_j w_{i,j}$$

The loss function for a node i :

$$L_i = -\frac{1}{||w||_i} \sum_j w_{ij} ln\left(\frac{S(r_i, r_j)}{||S||_i}\right)$$

The full loss function of the layout is then:

$$L = \sum_i L_i$$

The gradient of the full loss function is thereby defined from:

$$\nabla L(r) = \left(\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y} \right)$$

A.1.3 Derivation

Working with only the x-derivative, the change in loss for moving a node k in the layout in the x-direction is expressed by:

$$\frac{\partial L}{\partial x_k} = \sum_i \frac{\partial}{\partial x_k} L_i = -\sum_i \sum_j \frac{w_{ij}}{||w||_i} \frac{\partial}{\partial x_k} ln\left(\frac{S(r_i, r_j)}{||S||_i} \right)$$

Expanding each term in the sum separately:

$$\frac{\partial L_{ij}}{\partial x_k} = -\frac{w_{ij}}{||w||_i} \frac{\partial}{\partial x_k} ln\left(\frac{S(r_i, r_j)}{||S||_i} \right) =$$

$$= -\frac{w_{ij}}{||w||_i} \left(\frac{\partial}{\partial x_k} ln(S(r_i, r_j)) - \frac{\partial}{\partial x_k} ln(||S||_i) \right) =$$

$$= -\frac{w_{ij}}{||w||_i} \left(\frac{1}{S(r_i, r_j)} \frac{\partial}{\partial x_k} S(r_i, r_j) - \frac{1}{||S||_i} \frac{\partial}{\partial x_k} ||S||_i \right)$$

The derivative of the similarity function reads:

$$\frac{\partial}{\partial x_k} S(r_i, r_j) = \frac{\partial}{\partial x_k} \frac{1}{1 + |r_i - r_j|^2} =$$

$$= -\frac{1}{(1 + |r_i - r_j|^2)^2} \frac{\partial}{\partial x_k} 1 + |r_i - r_j|^2 =$$

$$= -S(r_i, r_j)^2 \frac{\partial}{\partial x_k} |r_i - r_j|^2$$

Inserting that $r_i = [x_i, y_i]$ we get the x-derivative as:

$$\frac{\partial}{\partial x_k} |r_i - r_j|^2 = \frac{\partial}{\partial x_k} \left((x_i - x_j)^2 + (y_i - y_j)^2 \right) =$$

$$= 2(x_i - x_j) \left(\frac{\partial x_i}{\partial x_k} - \frac{\partial x_j}{\partial x_k} \right)$$

Since $\partial x_i / \partial x_k = 1$ if $i = k$ and 0 otherwise, denote the derivatives with Kronecker delta notation. Insertion gives that

$$\frac{\partial}{\partial x_k} S(r_i, r_j) = -2S(r_i, r_j)^2 (x_i - x_j)(\delta_{ik} - \delta_{jk})$$

Inserting this into the first term of gradient expression yields:

$$\frac{\partial L_{ij}}{\partial x_k} = -\frac{w_{ij}}{||w||_i}\Big(-\frac{2S(r_i, r_j)^2}{S(r_i, r_j)}(x_i - x_j)(\delta_{ik} - \delta_{jk}) - \frac{1}{||S||_i}\frac{\partial}{\partial x_k}||S||_i\Big)$$

To expand the second term, we recall the definition of the similarity norm and the derivative of the similarity function and obtain:

$$\frac{\partial}{\partial x_k}||S||_i = \sum_l \frac{\partial}{\partial x_k} S(r_i, r_l) = -2\sum_l S(r_i, r_l)^2 (x_i - x_l)(\delta_{ik} - \delta_{lk}) =$$

$$= 2S(r_i, r_k)^2 (x_i - x_k) - \delta_{ik} 2 \sum_l S(r_i, r_l)^2 (x_i - x_l)$$

Inserting this in the expression we get the full function:

$$\frac{\partial L_{ij}}{\partial x_k} = -\frac{w_{ij}}{||w||_i}\Big(-\frac{2S(r_i, r_j)^2}{S(r_i, r_j)}(x_i - x_j)(\delta_{ik} - \delta_{jk}) - \frac{1}{||S||_i}\big(2S(r_i, r_k)^2(x_i - x_k) - \delta_{ik} 2\sum_l S(r_i, r_l)^2 (x_i - x_l)\big)\Big) =$$

$$= \frac{w_{ij}}{||w||_i}\Big(\frac{2}{||S||_i} S(r_i, r_k)^2 (x_i - x_k) - \delta_{jk} 2S(r_i, r_j)(x_i - x_j) + \delta_{ik} 2S(r_i, r_j)(x_i - x_j) - \delta_{ik}\frac{2}{||S||_i}\sum_l S(r_i, r_l)^2 (x_i - x_l)\Big)$$

We put the leading factor inside the parentheses and denote the terms by:

$$\frac{\partial L}{\partial x_k} = \sum_i \sum_j \frac{\partial L_{ij}}{\partial x_k} = \sum_i \sum_j A + B + C + D$$

We now expand each term separately and remove the terms where any deltas equal zero

Term A:

$$\sum_i \sum_j \frac{w_{ij}}{||w||_i} \frac{2}{||S||_i} S(r_i, r_k)^2 (x_i - x_k) = \sum_i \frac{1}{||w||_i} \frac{2}{||S||_i} S(r_i, r_k)^2 (x_i - x_k) \sum_j w_{ij} =$$

$$= \sum_i \frac{2}{||S||_i} S(r_i, r_k)^2 (x_i - x_k) \sum_j \frac{w_{ij}}{||w||_i} = \sum_i 2\frac{1}{||S||_i} S(r_i, r_k)^2 (x_i - x_k)$$

Term B:

$$\sum_i \sum_j -\delta_{jk} 2 \frac{w_{ij}}{||w||_i} S(r_i, r_j)(x_i - x_j) = \sum_i -2 \frac{w_{ik}}{||w||_i} S(r_i, r_k)(x_i - x_k)$$

Term C:

$$\sum_i \sum_j \delta_{ik} \frac{w_{ij}}{||w||_i} 2S(r_i, r_j)(x_i - x_j) = \sum_j \frac{w_{kj}}{||w||_k} 2S(r_k, r_j)(x_k - x_j) =$$

$$= \sum_i \frac{w_{ki}}{||w||_k} 2S(r_k, r_i)(x_k - x_i) = \sum_i -2 \frac{w_{ik}}{||w||_k} S(r_i, r_k)(x_i - x_k)$$

Term D:

$$\sum_i \sum_j -\delta_{ik} \frac{w_{ij}}{||w||_i} \frac{2}{||S||_i} \sum_l S(r_i, r_l)^2 (x_i - x_l) = \sum_j - \frac{w_{kj}}{||w||_k} \frac{2}{||S||_k} \sum_l S(r_k, r_l)^2 (x_k - x_l) =$$

$$= -\frac{2}{||S||_k} \sum_l S(r_k, r_l)^2 (x_k - x_l) \sum_j \frac{w_{kj}}{||w||_k} = -\frac{2}{||S||_k} \sum_l S(r_k, r_l)^2 (x_k - x_l) =$$

$$= -\frac{2}{||S||_k} \sum_i S(r_k, r_i)^2 (x_k - x_i) = \sum_i 2 \frac{1}{||S||_k} S(r_i, r_k)^2 (x_i - x_k)$$

Combining all the terms into the final expression for the loss function gradient results in:

$$\frac{\partial L}{\partial x_k} = 2 \sum_i -w_{ik} \left(\frac{1}{||w||_i} + \frac{1}{||w||_k} \right) S(r_i, r_k)(x_i - x_k) + \left(\frac{1}{||S||_i} + \frac{1}{||S||_k} \right) S(r_i, r_k)^2 (x_i - x_k)$$

B
Bibliography

Laurens van der Maaten, Geoffrey Hinton. *Visualizing Data using t-SNE*. Journal of Machine Learning Research 9 (2008).
`https://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf`