

UTRECHT UNIVERSITY

MASTER THESIS

---

# Large Weighted Graph Layouts by Deep Learned Multidimensional Projections

---

*Author:*

Ilan HARTSKEERL

*Supervisors:*

Alexandru C. TELEA  
Tamara MTSENTLINTZE

*A thesis submitted in fulfillment of the requirements  
for the degree of Master of Science*

*in the*

Department of Information and Computing Sciences

July 2, 2024

UTRECHT UNIVERSITY

# *Abstract*

Department of Information and Computing Sciences

Master of Science

**Large Weighted Graph Layouts by Deep Learned Multidimensional Projections**

by Ilan HARTSKEERL

tsNET is able to create very high quality graph layouts, but is too slow to run on large graphs. We propose a new graph layout method, NNP-NET, based on tsNET, with the aim of generating layouts for very large graphs. NNP-NET uses NNP to approximate the t-SNE step of tsNET with neural networks with a similar quality compared to layouts generated by tsNET. This thesis will go into the challenges of adapting NNP to a graph layout context and how we solved them. NNP-NET is compared to other state of the art methods, where we show that NNP-NET gets good quality results when compared to other fast methods. Here we also show that NNP-NET is able to create layouts for graphs with millions of nodes in a reasonable amount of time. For very large graphs, the execution time of NNP-NET ends up lower than competing state of the art methods.

## Chapter 1

# Introduction

Generating good layouts for graphs is an important problem to solve in order to easily visualize graph data. There are numerous graph layout methods created in order to fulfill this purpose, all taking different approaches to create a good graph layout. These approaches largely fall into two main categories: force-directed and dimensionality reduction based. Lately however, a lot of research is focusing on leveraging graph neural networks in order to generate graph layouts.

All graph layout methods try to solve the same problem however: Trying to generate readable, pleasing graph layouts from a given input graph. Mathematically defining what constitutes a good graph layout is not easy however, leading to different methods optimizing different metrics of what makes a good layout. Another important factor some methods look at is running time/scalability. These methods try to generate the best possible graph layouts within a reasonable time, making them suitable for generating layouts for bigger graphs. Methods that sacrifice runtime for better quality graphs are often unable to generate layouts for larger graphs, as it would either take too much time, or too much memory.

Graphs can come with weights attached to the edges, that say something about the importance of that edge. These edge weights would ideally be reflected in the resulting layout. However, this data is often ignored by most graph layout methods, choosing to instead assume as constant weight for each edge. In this thesis, we introduce a method that takes these edge weights into account. It is also scalable, making it suitable for large graphs with 100.000 or more nodes. It is also important that the layout quality does not suffer, where we are particularly interested in the neighborhood preservation metric.

In this thesis, we incorporate edge weights into tsNET, a promising graph layout technique that uses t-SNE dimensionality reduction in order to preserve neighborhoods of the original graph. A large problem with tsNET however is its runtime. tsNET gives good results, but takes a long time to execute and is unable to run on larger graphs due to its memory requirement. Another goal of this thesis is to modify tsNET in order to get better performance. This is achieved using NNP, a technique which uses neural networks in order to imitate any dimensionality reduction method, which includes the t-SNE step used by tsNET.

The final method should fulfill the following criteria:

- **Scalability:** The method needs to be able to handle large graphs in a reasonable amount of time, comparable to other fast layout algorithms. The easiest way to assess this is by looking both the time complexity and the memory usage. Both of these should ideally scale linearly with the input size, and should be less than quadratic. Layout methods that have a quadratic memory footprint will run out of memory on larger graphs. If the time complexity of the method is not linear, then it will end up significantly slower than other methods that do scale linearly with input size. There are two different times to consider in this case: The training time, and the layout time. In theory, a model trained for a graph could be reused later use on a similar

graph. While it would be nice to get the training time also down to linear time complexity, this will not be the focus. The main scalability goal will be to lay out graphs using a trained model in linear time.

- **Layout Quality:** The generated layouts needs to be of comparable quality to other graph layout algorithms. When looking at graphs with hundred thousands/millions of nodes, the visual clutter will become very significant, which could also make judging the quality of a graph by looking at it hard. Judging layouts by hand would also introduce bias. Neighborhood preservation is used an objective metric to assess graph quality.
- **Robustness:** The layout algorithm should work for all types of graphs, not just on certain types of graphs.
- **Ease of use:** The method should be easy to use for an end user, without requiring fine tuning of multiple complex parameters.
- **Edge Weights:** The method should be able to take edge weights into account.

The final method, NNP-NET, creates a high dimensional embedding using PMDS, which is used as the input for NNP. NNP is trained on a subset of the complete graph, which is created by reducing the graph into a smaller representation of itself. A ground truth is created using tsNET on the smaller graph, which is used as the training data together with the high dimensional embedding created by PMDS. This network is then used to infer the position of all points in the graph using the high dimensional embedding as the input.

This thesis starts with a literature review of the graph layout problem, as well as how other state of the art methods are generating graph layouts. Section 3 explain our proposed method, NNP-NET, along with some alternative options that were also considered and tested. Section 4 will go over the results gathered both from tests comparing different versions of our algorithm, as well as comparisons to state of the art graph drawing methods. This is followed with a discussion and a subsequent conclusion.

## Chapter 2

# Background and Related Work

In this chapter, we give a definition of the graph layout problem, along with metrics to evaluate the solutions. We then take a look at dimensionality reduction, which is closely linked to graph layout. Current graph layout algorithms are then explained.

Some common notations that are used in the following sections are summarized in table 2.1

Symbol	Definition
$\mathbf{G}$	The input graph
$\mathbf{V}$	List of all nodes in $\mathbf{G}$
$\mathbf{x}_i$	Node in $\mathbf{V}$
$\mathbf{E}$	List of all edges in $\mathbf{G}$
$\mathbf{Y}$	List containing the output positions
$\mathbf{y}_i$	Output position corresponding to input node $\mathbf{x}_i$
$d_{ij}$	Distance between $\mathbf{x}_i$ and $\mathbf{x}_j$
$n$	Number of input dimensions used for dimensionality reduction
$m$	Number of dimensions used for the resulting graph layout

TABLE 2.1: Common notations used throughout the thesis

### 2.1 Problem Definition

Graph layout problems are given a graph  $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ . Here,  $\mathbf{V} = \{\mathbf{x}_i\}_{i=1}^N$  and  $\mathbf{E} = \{(\mathbf{x}_i, \mathbf{x}_j)\} \subseteq \mathbf{V} \times \mathbf{V}$ . Alternatively,  $\mathbf{E} = \{(\mathbf{x}_i, \mathbf{x}_j) \in \mathbf{V} \times \mathbf{V}, w_{ij}\}$ , where  $w_{ij}$  is a weight associated with that edge. Most graph layout methods assume a constant weight however.

Graph layout algorithms will map  $\mathbf{G}$  into positions  $\mathbf{Y} = \{\mathbf{y}_i\}_{i=1}^N$ . Each  $\mathbf{y}_i$  is a position in  $m$  dimensions, where typically  $m \in \{2, 3\}$  for visualization purposes. The goal is to place the output positions  $\mathbf{y}_i$  in an intuitive and easy to interpret way so that the user can easily read information from the resulting graph.

There are a lot of methods to generate graph layouts. We do not cover all of these methods in this thesis. Only recent developments and methods that are very relevant to our research are discussed. To learn about other methods, we refer to the following literature (Gibson, Faith, and Vickers, 2013, Tamassia, 2013).

### 2.2 Quality metrics

How "good" a given graph layout is hard to define. A multitude of metrics have been proposed to quantify how good a layout is perceived by end users. Some of the more prevalent metrics will be discussed in this section.

### 2.2.1 Stress

Stress measures how far the distances in  $\mathbf{Y}$  are deviating from the the graph theoretical distances from  $\mathbf{G}$ . The stress  $\sigma$  is calculated using the following formula:

$$\sigma = \sum_{i,j} \left( \frac{d_{ij} - \|\mathbf{y}_j - \mathbf{y}_i\|}{d_{ij}} \right)^2. \quad (2.1)$$

Here,  $d_{ij}$  is the graph theoretical distance between  $\mathbf{x}_i$  and  $\mathbf{x}_j$ . A lower value for  $\sigma$  is better, with 0 as the minimum value and no upper bound.

### 2.2.2 Neighborhood preservation

The neighborhood preservation metric represents how well neighborhoods from  $\mathbf{G}$  are preserved in  $\mathbf{Y}$ . The formula used by Krueger et al., 2017 is based on Gansner, Hu, and North, 2013. Firstly, the neighborhood  $N_G(\mathbf{x}_i, r_G)$  of each node is defined as

$$N_G(\mathbf{x}_i, r_G) = \{\mathbf{x}_j \in \mathbf{V} | d_{ij} \leq r_G\}, \quad (2.2)$$

where  $r_G$  is the maximum distance nodes can be apart to be considered neighbors, and  $d_{ij}$  is the graph theoretical distance between  $\mathbf{x}_i$  and  $\mathbf{x}_j$ . This can then be used to compute the neighborhood preservation metric  $v$  as follows:

$$v = \frac{1}{|\mathbf{V}|} \sum_i \frac{|N_G(\mathbf{x}_i, r_G) \cap N_G(\mathbf{y}_i, r_G)|}{|N_G(\mathbf{x}_i, r_G) \cup N_G(\mathbf{y}_i, r_G)|}. \quad (2.3)$$

$v$  will have a value between 0 and 1, where a higher value is better.

### 2.2.3 Crosslessness

Crosslessness (Purchase, 2002) looks at the number of edge crossings in the graph, where less edge crossings is better. The crosslessness metric  $k$  is calculated using an estimated upper-bound on the maximum number of possible edge crossings, and using that to scale the actual number of edge crossings as followed:

$$k = \begin{cases} 1 - \sqrt{\frac{c}{c_{max}}} & , \text{ If } c_{max} > 0 \\ 1 & , \text{ Otherwise} \end{cases} \quad (2.4)$$

$$c_{max} = \frac{|E|(|E| - 1)}{2} - \frac{1}{2} \sum_{\mathbf{x} \in \mathbf{V}} \text{degree}(\mathbf{x})(\text{degree}(\mathbf{x}) - 1). \quad (2.5)$$

Here,  $c$  is the number of crossings,  $c_{max}$  is the estimated upper-bound on the number of crossings and  $\text{degree}(\mathbf{x})$  the number of edges connected to  $\mathbf{x}$ .  $k$  has a value between 0 and 1, with a higher value being better. This metric is the reversed version of the number of edge crossings, which can instead be used as a metric depending on preference.

### 2.2.4 Minimum Angle

The minimum angle metric looks at the smallest angle created between all edges attached to each  $\mathbf{x}_i$ . This will be compared to the theoretical maximum for the smallest angle between each of those edges. Minimum angle  $a$  will then be the average value across all nodes:

$$a = 1 - \frac{1}{|\mathbf{V}|} \sum_i \left| \frac{\theta(\mathbf{x}_i) - \theta_{\min}(\mathbf{x}_i)}{\theta(\mathbf{x}_i)} \right| \quad (2.6)$$

$$\theta(\mathbf{x}_i) = \frac{360}{\text{degree}(\mathbf{x}_i)}. \quad (2.7)$$

Here,  $\theta_{\min}(\mathbf{x}_i)$  is the actual minimum angle calculated from  $\mathbf{V}$ , and  $\theta(\mathbf{x}_i)$  is the theoretical minimum angle.  $a$  has a value between 0 and 1, with a higher value being better.

## 2.3 Dimensionality Reduction

Dimensionality reduction (DR) maps a high  $n$  dimensional input to a low dimensional  $m$  output. Here,  $\mathbf{X} = \{\mathbf{x}_i\}_{i=0}^N$  is the input data, where each  $\mathbf{x}_i$  is a  $n$  dimensional datapoint, and  $\mathbf{Y} = \{\mathbf{y}_i\}_{i=0}^N$  is the corresponding output data, where each  $\mathbf{y}_i$  is a  $m$  dimensional output point. Typically,  $m \ll n$  and  $m \in \{2, 3\}$  for visualization purposes.

Dimensionality reduction has its own set of metrics to evaluate the output of the method. Some of these are very similar to metrics used for graph layout. Some of the metrics used for DR are:

- **Normalized Stress:** Very similar to the stress metric used for graph layouts. Measures the difference between the distances in the input space  $\mathbf{X}$  and output space  $\mathbf{Y}$ . This used almost the same formula (Equation 2.1), except for how  $d_{ij}$  is defined. When using stress for graph layouts,  $d_{ij}$  is defined as the graph theoretical distance between  $\mathbf{x}_i$  and  $\mathbf{x}_j$ . For DR,  $d_{ij}$  is defined as  $d_{ij} = \|\mathbf{x}_i - \mathbf{x}_j\|$ .
- **Trustworthiness and Continuity:** Trustworthiness (Venna and Kaski, 2001) is very similar to the neighborhood preservation metric used for graph layout as it looks at how many points that are close in  $\mathbf{Y}$  are also close in  $\mathbf{X}$ . Continuity is the same, except it looks the other way around, how many points that are close in  $\mathbf{X}$  are also close in  $\mathbf{Y}$ . These metrics say how well local patterns and neighborhoods are preserved between the input and output space, just as the neighborhood preservation metric for graph layouts.
- **Neighborhood hit:** Neighborhood hit (Paulovich et al., 2008) looks at how many of a point  $\mathbf{y}_i$ 's closest neighbors have the same label. This tells how well separable the output plot is. Graph layout does not have a similar metric, as Graphs generally do not come with labels attached to its nodes.
- **Shepard diagram correlation:** A shepard diagram (Joia et al., 2011) is a scatterplot that is drawn where every pair of points is plotted, where one axis is the distance between those points in  $\mathbf{X}$ , and the other axis the distance in  $\mathbf{Y}$ . Ideally, the resulting scatterplot would be a diagonal line. A point that is not on the diagonal would indicate either a missing/falls neighbor. The spearman rank correlation is then calculated on the scatterplot which is used as the metric. Graph layout does not have a similar metric.

DR shares a lot of commonalities with generating graph layouts, which results in multiple metrics that are very similar in what they try to achieve. Similarly to what DR tries to achieve, generating Graph layouts reduces the input data into a  $m$  dimensional output. The dimensionality of this graph data however is not really known, only the distances between the nodes are. The difference here comes from the structure of the data. Graphs can't be given to DR algorithms directly, as they use  $n$  dimensional points as input.

There are a lot of different DR algorithms, but the one relevant for this research is t-SNE, as it is the basis of tsNET. Because of this, other methods are not listed in this thesis. For information about other DR methods, we refer to an extensive literature study done by Ayesha, Hanif, and Talib, 2020.

### 2.3.1 t-SNE

t-SNE (Maaten and Hinton, 2008) is the DR method that is the basis for tsNET, which is explained later in this thesis. t-SNE starts by defining the probability  $p_{j|i}$ , which is the probability that  $\mathbf{x}_j$  would pick  $\mathbf{x}_i$  as its neighbor. Probability  $p_{j|i}$  is defined as follows:

$$p_{j|i} = \frac{\exp(-\|\mathbf{x}_i - \mathbf{x}_j\|^2 / 2\sigma_i^2)}{\sum_{k \neq i} \exp(-\|\mathbf{x}_i - \mathbf{x}_k\|^2 / 2\sigma_i^2)}, p_{i|i} = 0, \quad (2.8)$$

where  $\sigma_i$  is the Gaussian standard deviation for  $\mathbf{x}_i$ .  $\sigma$  has to be chosen so that perplexity  $Perp(P_i) = 2^{-\sum_j p_{j|i} \log_2 p_{j|i}}$ . This is done using binary search to get to a right value for  $\sigma_i$ . The perplexity value itself is given by the user. This probability is symmetrized where

$$p_{ji} = p_{ij} = \frac{p_{i|j} + p_{j|i}}{2}. \quad (2.9)$$

The probabilities  $q_{ij}$  are the corresponding probabilities for the output space, where  $p_{ij}$  is the probability that  $\mathbf{y}_i$  would pick  $\mathbf{y}_j$  as its neighbor in the output space.  $q_{ij}$  is defined with a different function as opposed to  $p_{i|j}$ .  $p_{i|j}$  is defined with a gaussian distribution, where  $q_{ij}$  uses a student t-distribution:

$$q_{ij} = \frac{(1 + \|\mathbf{y}_i - \mathbf{y}_j\|^2)^{-1}}{\sum_{k \neq i} (1 + \|\mathbf{y}_i - \mathbf{y}_k\|^2)^{-1}}, q_{ii} = 0. \quad (2.10)$$

These probabilities are then used to calculate the Kullback-Leibler divergence  $C_{KL}$  between the probabilities:

$$C_{KL} = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}} \quad (2.11)$$

The desired output positions  $\mathbf{Y}$  can then be found by optimizing for  $C_{KL}$ . The focus of t-SNE is not to have an accurate distance from each point to each other point, but instead for the k-nearest neighbors for each point to be the same in both  $\mathbf{X}$  and  $\mathbf{Y}$ , where k is the perplexity given by the user. This results in good overall performance relative to other dimensionality reduction method (Maaten and Hinton, 2008).

t-SNE uses the distances between points, which can be given as either the full distance matrix, or as a list of feature vectors, from which the distance matrix is calculated. Being able to give the distance matrix directly gives as an advantage that t-SNE can be run on data where you do not have feature vectors, but only distances between points, which tsNET takes advantage of. t-SNE performs very well on the trustworthiness and continuity metrics, as it is build to preserve neighborhoods instead of distances (stress).

t-SNE does have some downsides. Firstly, it is very slow. Each iteration has a time complexity of  $O(N^2)$ , resulting in long running times on larger datasets. Secondly, t-SNE is not stable. This means, running t-SNE twice on the same dataset does not result in the same output twice. This is the result of the random initialization used by t-SNE. This could be solved by either using a different initialization method, or a set random seed. t-SNE will however always lack out of sample (OOS), meaning that data points can not be



added to  $\mathbf{Y}$  once  $\mathbf{Y}$  is already created. The only way to add additional points to  $\mathbf{Y}$  once  $\mathbf{Y}$  is calculated is by redoing t-SNE from the start.

### 2.3.2 Deep Learning t-SNE

Espadoto, Hirata, and Telea, 2020 found that it is possible to simulate any dimensionality reduction technique, like t-SNE, using neural networks. The network used consisted of an input layer the size  $n$ , corresponding to the higher dimensional space of  $\mathbf{x}_i$ , three hidden layers sized {256, 512, 256} and an output layer size 2, one for each output dimension  $m$ . It was mentioned that there is nothing particularly special about this setup, and that it can likely be changed to something else with similar results.

The network is trained using a ground truth output obtained from running the dimensionality reduction method that will be simulated. Afterwards,  $\mathbf{x}_i$  can be inputted into the neural network, which results in the corresponding  $\mathbf{y}_i$ . This is then repeated for every  $\mathbf{x}_i \in \mathbf{X}$  to get the desired scatterplot.

This method is very fast. It has a linear time complexity with respect to number of points plotted. The network is evaluated once per data point. Evaluating the network only dependent on the point being evaluated, giving the linear time complexity. It is also stable, plotting every point  $\mathbf{x}_i$  in the same place as the ground truth did. This method also has OOS, meaning a plot can also be extended with new unseen points after it has already been drawn, even if the method it is trained on can not do that.

This approach does have some limitations however. Most importantly, it is not generalizable. It has to be retrained for each different dataset, which requires a ground truth from the original dimensionality reduction method. Having to run the original method to obtain the ground truth for training gets rid of all time advantage this method had over running the base algorithm.

Another limitation is that it has to use the input data  $\mathbf{X}$ , and can not use the distance matrix directly, like t-SNE was able to do. Because of this, this method is usable in less situation compared to using t-SNE directly.

## 2.4 tsNET

From here on out, different already established methods will be discussed that create graph layouts. The first one discussed is tsNET, as it is the method that we are trying to improve.

tsNET is a DR-based graph drawing algorithm proposed by Krueger et al., 2017. It leverages a modified version of t-SNE to create a scatterplot of all vertices of  $\mathbf{G}$ , over which all edges are drawn. The value that is optimized for in tsNET is not just  $C_{KL}$ , but instead:

$$C = \lambda_{KL} C_{KL} + \frac{\lambda_c}{2N} \sum_i \|\mathbf{y}_i\|^2 - \frac{\lambda_r}{2N^2} \sum_{i \neq j} \log(\|\mathbf{y}_i - \mathbf{y}_j\| + \epsilon_r). \quad (2.12)$$

Here,  $\lambda_{KL}$ ,  $\lambda_c$  and  $\lambda_r$  are weights for the three different terms of the equation. The first term is the Kullback-Leibler divergence as used by t-SNE. The second term is the compression term, which is known to reduce optimization time (Maaten and Hinton, 2008) as described in the original paper. Term 3 is used to repulse nodes that are too close together.  $\epsilon_r$  is a small regularization constant which is set to  $\frac{1}{20}$ . This is done for situation where nodes are almost in the exact same location. This term is the entropy model from Gansner, Hu, and North, 2013.

tsNET does its optimization in three different steps. In the first step, the values of  $\mathbf{Y}$  are given random initial values. The second step runs the modified t-SNE with weights

$\{\lambda_{KL}, \lambda_c, \lambda_r\} = \{1, 1.2, 0\}$ . In the third stage, the weights are changed to  $\{\lambda_{KL}, \lambda_c, \lambda_r\} = \{1, 0.01, 0.6\}$ .

An alternative method was also proposed in the paper, called tsNET\*. In this variation, the first stage is replaced with PMDS (Brandes and Pich, 2007), and the weights for the second stage are changed to  $\{\lambda_{KL}, \lambda_c, \lambda_r\} = \{1, 1.2, 0\}$ . Overall, tsNET\* is faster than tsNET (reported to be about 15% (Kruiger et al., 2017)), and produces better results. tsNET\* is a stable algorithm as opposed to tsNET, thanks to the initialization step done by PMDS.

tsNET inherits all problems from t-SNE. It does not scale well to larger graphs. tsNET memory usage is also  $O(N^2)$  to store the distance matrix, resulting in high memory costs for large graphs. Just like t-SNE, tsNET is not stable, although using tsNET\* does solve t-SNE's stability problem. Both tsNET and tsNET\* do not have OOS.

## 2.5 DR based Graph Layouts

There are more DR based graph layout algorithms aside from tsNET. In this section we look at what other algorithms are doing when compared to tsNET.

### 2.5.1 PMDS

Pivot Multidimensional scaling (PMDS, Brandes and Pich, 2007) is an extension of Classic MDS (Torgerson, 1952) and Landmark MDS (Silva and Tenenbaum, 2002). MDS is not only a graph layout method, but is also used for dimensionality reduction. Classic MDS makes use of the same distance matrix used by tsNET. A matrix  $\mathbf{B}$  is then constructed by double-centering the distance matrix so that all rows and columns sum up to a total of 0. Eigendecomposition can then be applied to  $\mathbf{B}$  to calculate the resulting output positions.

Pivot MDS changes this by not considering the full distance matrix, but instead using a set number of pivot points. This results in a distance matrix that is not  $N \times N$ , but instead  $N \times k$ , where  $k$  is the number of pivot points used. Here, the distance matrix only contains the distance from each point to every pivot point. The pivot points are chosen using a *max - min* scheme. In a *max - min* scheme, the next pivot point chosen is the point that has the largest minimum distance to any other point pivot point already chosen.

A strong point of PMDS in comparison to other graph layout algorithms is its running time. The time complexity of PMDS ends up at  $O(k^2N)$ . This is a linear time complexity with respect to the input size  $N$ . In other papers that compare the running time of multiple graph layout algorithms, PMDS ends up faster than other methods (Zhu et al., 2020). This speed does come at the expense of quality. The layouts produced by PMDS are of lower quality compared to other available methods (Kruiger et al., 2017, Zhu et al., 2020).

### 2.5.2 DRGraph

DRGraph (Zhu et al., 2020) uses tsNET as a basis, and made changes in order to greatly improve both performance and memory cost. Three major changes were made to the pipeline:

Firstly, DRGraph uses a sparse distance matrix instead of the full distance matrix. The only distances that are taken into account are the  $k$ -order nearest neighbors, in other words, all nodes that are at most  $k$  edges away from the current node. These distances are stored in a sparse matrix, which is then used for the next steps. Only the points that are in the sparse matrix are used for all calculations afterwards, giving a large performance increase, as only the closest, most relevant nodes are used. This can be done without a significant impact on quality because of how t-SNE works. Large distances have almost no effect on

$p_{ij}$ , and can thus be omitted for a performance improvement with only a minimal loss in quality.

Secondly, a different way to calculate the gradient is used. tsNET uses gradient decent in order to minimize its cost function. Calculating the gradient for this gradient decent has a time complexity of  $(N^2)$ , where  $N$  is the number of vertices. DRGraph uses a negative sampling technique, which approximates the gradient (Mikolov et al., 2013). This method does not sample every point when calculating the gradient for a single point, but instead takes  $M$  points, where  $M$  is the number of negative samples used. This sampling method reduces the time complexity for the optimization step from  $O(tN^2)$  to  $O(tMN)$ , where  $N$  is the number of vertices and  $t$  is the number of iterations.

Lastly, DRGraph uses a different initial layout. Where tsNET and tsNET\* uses a random layout and PMDS respectively, DRGraph uses a multi-level layout scheme. First, the graph gets coarsened. This is done by taking a random vertex, finding its first order neighbors (directly connected by an edge), and reducing them into a single vertex. This is repeated up to a certain point, after which the resulting graph will be initialized with a random initialization. From here, the coarsening steps can be reverted one by one, with refining steps in between.

In testing, DRGraph was significantly faster than tsNET. This is unsurprising, as the time complexity of DRGraph ends up as  $O(N)$ . Important to note is that the implementation of tsNET used changed from the original paper. A GPU accelerated version of t-SNE was used instead of the original python implementation. This has no impact on the resulting graphs, but is significantly faster than the original implementation. DRGraph also used significantly less memory, allowing it to draw bigger graphs than possible with tsNET.

The quality of the resulting graph depended on the metric. tsNET and DRGraph got comparable results when looking at neighborhood preservation. DRGraph seemed to outperform tsNET on the other metrics that were calculated. Important to note is the DR-Graph was not compared to tsNET\*, which got better results than tsNET in the original paper.

The quality for the resulting graph does rely on 5 different parameters. A similar configuration of parameters can be used for most graphs however. The sparse matrix approach to the iteration steps makes it hard for DRGraph to create a good global graph structure, as the global structure is not represented in the sparse matrix. This is partially alleviated by the coarsened graph initialization. There is however no guarantee that all nodes are coarsened precisely, as the coarsening is done randomly.

Expanding DRGraph to include edge weights might be harder than for other methods like tsNET. With tsNET, the calculations for the graph theoretical distances can be expanded to include the edge weights. Doing this with DRGraph would present potential problems. With edge weights, there is no guarantee that the  $k$ -order nearest neighbors are the closest points, as it looks at all nodes at most  $k$  edges away without taking the edge length into account.  $k$ -order nearest neighbors could be replaced by either  $k$ -nearest (the  $k$  closest nodes) or  $r$ -nearest neighbors (all nodes at most  $r$  distance away). Using the  $k$  nearest points can end up ignoring certain edges if either a node has more than  $k$  edges, or when there are nodes more than one edge away that are closer.  $r$ -nearest neighbors looks at all nodes that are at most  $r$  distance away.  $r$  would have to be chosen sufficiently high, so that it is at least as long as the longest edge, else some edges would never be taken into account. Taking a too high  $r$  value could have a negative performance impact however. A lot of neighbors would have to be considered if a graph has a lot of very short edges in comparison with the chosen  $r$  value. Whether any of these potential problems would result in real world problems for the algorithm would have to be tested.

### 2.5.3 Graph Drawing by High-Dimensional Embedding

Harel and Koren, 2002 start by generating a  $n$ -dimensional embedding of the graph. This is done using  $n$  pivot points. The pivot points are chosen in the same way as done by PMDS (Brandes and Pich, 2007), where the first is chosen randomly, and the rest by maximizing the minimum distance to the other already chosen pivot points. This results in each  $x_i$  having a  $n$  dimensional embedding with the graph theoretical distances between it and each pivot point. This  $n$  dimensional embedding is then reduced to a 2 or 3 dimensional graph layout using Principal Component Analysis (PCA). In the paper,  $n$  was typically set to a value of 50, which proved to be sufficient for all input sizes.

This approach is very fast. Its time complexity is linear ( $O(N)$ ) with respect to the input size. This is at the cost of layout quality however. PCA tries to fit a plane through the  $n$  dimensional space. This will only result in an accurate representation of the data when all the data directly falls on that plane in high dimensional space. This often is not the case, resulting in worse quality layouts in comparison to layouts generated by other methods. Harel and Koren, 2002 mentioned that force directed methods generally generate more pleasing layouts. This was also from 2002, after which even better layout methods have emerged. This method for creating high dimensional embeddings for a graph does seem promising, although using PCA to generate the low dimensional embedding does not seem suitable.

## 2.6 Force-directed methods

Force-directed methods are a different class of graph layout methods opposed to DR based methods. Force-directed methods work by modeling the graph as a physical system with forces. These methods are often easy to understand, but tend to have a quadratic time complexity, making them slow on large graphs. There are a lot of different Force-directed graph layout algorithms, but we will only talk about a few recent developments in the field.

### 2.6.1 Stochastic Gradient Descent

Zheng, Pawar, and Goodman, 2019 proposed a graph layout method using Stochastic Gradient Descent (SGD) that tries to minimize stress. Often, the gradient at a specific vertex is calculated by looking at every other vertex in the graph. SGD does not calculate the gradient on a vertex basis, but instead by only looking at a pair of vertices at a time. A small modification was made, adding an additional step size modifier  $\eta$  when applying the forces.  $\eta$  decreases every iteration towards 0.

SGD was compared to other stress minimization graph layout methods, where SGD took less iterations to converge then the alternative methods, while also reaching lower stress. SGD was not tested for neighborhood preservation and was also not compared to methods that aim for neighborhood preservation. It is likely however that SGD would perform worse then methods that do aim for high neighborhood preservation. Although SGD has better performance compared to other similar methods, it still has time complexity  $O(N^2)$ , making it to slow for large graphs.

### 2.6.2 $(GD)^2$ and $(SGD)^2$

Ahmed et al., 2020 use gradient decent in order to create graph layouts. Where  $(GD)^2$  differs from other methods using gradient decent is what they are minimizing. Gradient

decent methods typically try to minimize stress.  $(GD)^2$  argued that you can use any aesthetic metric to optimize a graph layout, or even combine multiple metrics. The metric functions are required to be smooth function to make this possible. Non-smooth metric functions were extended in order for them to be usable for gradient decent.

$(GD)^2$  was tested by giving it either a random layout, or a layout generated by a different graph layout algorithm.  $(GD)^2$  was then tasked to improve a specific quality metric. When given a random layout, it was almost always able to improve the layout for the given metric. The results were more mixed when given a layout from a different graph layout algorithm. Whether  $(GD)^2$  was able to improve the metric was dependent on the metric that was optimized for. In none of the cases did the results get worse however.

$(GD)^2$  is slow and not suitable for large graphs as it had a  $O(N^2)$  time complexity. To partially remedy this, SGD was used to optimize  $(GD)^2$ , which was named  $(SGD)^2$  (Ahmed et al., 2022). This gave a significant speed up, while still getting similar results.  $(SGD)^2$  still is not fast enough to draw very large graphs, as it relies on SGD which itself was also not fast enough.

## 2.7 Graph layouts using Deep Learning

More and more research is being done looking at applying neural networks to graph problems. One of these problems is the graph layout problem. This section will talk about how neural networks are applied to graph problems, and the methods developed to generate graph layouts using them.

### 2.7.1 Graph Neural Networks

Graph neural networks (GNN) is a concept first introduced by Gori, Monfardini, and Scarselli, 2005, which has seen a lot of research and interest in the last few years. At its very core, GNN's are neural networks that are able to take in graph structured data. GNN's can be applied to any type of problem involving graphs, including graph drawing. There are numerous variations and extensions of the original GNN, which can be read about in literature studies like Zhou et al., 2020 and Wu et al., 2021.

The variants that will be relevant for graph drawing are Convolutional Graph Neural Networks (ConvGNN). ConvGNN is an overarching term for GNN that apply ideas from conventional Convolutional Neural Networks to graph structured data. This is done using message passing between nodes in the graph. Each node will send the data located at the node along its connected edges to its neighboring nodes. Per node, the data used will be its own data together with the data received from its neighboring nodes. The exact details of how this is done differs between variants within ConvGNNs.

### 2.7.2 $(DNN)^2$

$(DNN)^2$  (Giovannangeli et al., 2021) build upon the work of Krueger et al. and uses the foundation of tsNET is a novel way using Deep Neural Networks.  $(DNN)^2$  replaces the entire pipeline with a convolutional neural network using graph convolutions, which they train using  $C$  from tsNET (Equation 2.12) as the loss function. The weights used for  $C$  are the same weights used by tsNET. The best results were when the network was first trained on a large set of randomly generated graphs, and subsequently fine-tuned on the training+validation dataset used. A alternative was also proposed,  $(DNN)^{2*}$ , mirroring tsNET and tsNET\*. Here, PMDS is also used in the first phase. The second phase also uses the second phase weights used by tsNET\*.  $(DNN)^{2*}$  tended to give slightly better results, but also took longer to execute due to the extra PMDS step.



$(DNN)^2$  and  $(DNN)^{2*}$  do have some limitations. The main limitations is on the input size. A max input size has to be defined before the training can start. A network with more vertices than the max input size of the network can not be drawn by the network. The input size could be set arbitrarily large, but this would greatly increase the training and execution time. It is also not known how well a larger model would work, as it was only tested with a max size of 128 nodes. Training the model also takes considerable resources. The resulting model is also not able to draw every type of graph, and would have to be retrained to handle different type of graphs.

$(DNN)^2$  is not suited for our purposes as it does not scale well, as well as not working on all graphs without retraining.

### 2.7.3 GraphTSNE

GraphTSNE (Leow, Laurent, and Bresson, 2019) is a graph layout algorithm using similar ideas to tsNET, using t-SNE to draw graphs. GraphTSNE however focuses on drawing multivariate graphs (graphs that include a feature vector at each vertex). GraphTSNE uses a Graph convolutional network (GCN, which falls under ConvGNN) to map between the feature vectors and low dimensional output space. The GCN is trained with a loss function that is made up from two different parts: The Graph distance loss  $C_g$ , and the feature distance loss  $C_x$ . These two are then combined into a single loss metric with the using  $C_t = aC_g + (1 - a)C_x$ . Here,  $a$  is a user defined variable telling the system whether to prioritize optimizing feature distance or graph distance. GraphTSNE's was not compared to any other other graph drawing methods, so the relative performance to other methods is not known.

### 2.7.4 DeepDrawing

DeepDrawing (Wang et al., 2020) uses a Long Short Term Memory (LSTM) network as a basis for their method. In such a network, each node is fed into the network one at a time. A LSTM network is able to remember the nodes that it has seen previously, and use that information for the node that it is currently processing.

This approach raised three important questions to be solved:

- How to represent a node as a feature vector that can be fed to the network
- In which order should the nodes be fed into the network
- On what type of data should the network be trained

The feature vector used by DeepDrawing is a  $k$ -sized adjacency vector, where  $k$  is set to a fixed number before training the network. This vector represents for the  $k$ -previous nodes whether this node is a direct neighbor, where a one means it is neighbor, and zero means it is not.

The order that the nodes are fed into the network becomes very important as a result of this encoding. A random order would not work optimally. The network would have to be trained on every possible permutation of random orderings. The order chosen by DeepDrawing is a breadth-first-search (bfs) ordering. This way, the network only has to be trained on inputs that are ordered by bfs, making the problem significantly easier.

The model is trained using ground truth data generated by a different graph layout method. DeepDrawing is then trained to replicate those layouts. The aim was to make DeepDrawing replicate the layout style of the ground truth input, which they were able to accomplish by giving the ground truth results of that method. The loss function used

in training was thus set to how far the network was from the ground truth created by a different graph layout method.

DeepDrawing has some limitations. Firstly, it was only tested on relatively small graphs (with around 40-50 nodes), and it is thus unknown how well this method will scale to bigger graphs. Poor drawing performance was also observed for graphs that have a significantly differing structure from the graphs in the training set. This would result in having to re-train the network to handle a specific case, which requires you to run the base algorithm that it is imitating, making DeepDrawing redundant in such cases.

### 2.7.5 DeepGD

DeepGD (Wang et al., 2021) uses Convolutional Graph Neural Networks in order to generate graph layouts. The network takes as input an adjacency matrix, as well as a matrix for both node and edge features. This allows DeepGD to also take the features of the a vertex/edge into account. Multiple loss functions were tested in the paper, each optimizing for different quality metrics. Multiple loss functions also could be combined to optimize for multiple quality metrics at the same time.

DeepGD was only tested on smaller graphs due to long training times on larger graphs. DeepGP also suffers from a common problem with deep learning methods: The ability to generalize on unseen data. If a graph has a structure that differs from the structures that were present in the input data, DeepGD will not provide satisfactory results. DeepGD also requires the user to choose a loss (or multiple) themselves, which requires knowledge about how the underlying metrics work and how that will reflect in the resulting graphs.

## 2.8 Multilevel methods

Multilevel layout methods optimize other layout methods by introducing a multi level layout scheme. The general idea is to iteratively reduce the size of  $G$  until a certain stopping criteria. A graph layout is then created for the smallest graph. This layout will then be used as the initial layout for the graph one iteration back. After refining that layout, the process will be repeated until we arrive back at the original graph  $G$ . DRGraph (explained back in section 2.5.2) also falls in this category, as well as the DR section it is placed in.

### 2.8.1 Fast Multipole Multilevel Method ( $fm^3$ )

Fast Multipole Multilevel Method (Hachul, 2005) is a multilevel method that uses force directed ideas as a base for creating the layouts. The authors explained their multilevel step by relating it to solar systems. Sun nodes are picked. The shortest path between each pair of sun nodes needs to have at least two other nodes in between. Each node directly connected to a sun nodes will become a planet node of that sun. Any leftover nodes become moon nodes of a neighboring planet node. These solar system are than collapsed into one node for the next iteration. This is repeated until the graph size is not reduced enough for a set number of iterations.

The suns are not chosen randomly however. Each nodes will be assigned a weight. At the start, each node has weight 1. When nodes are merged into a single node, all weights are added together. For choosing the next sun node, the node with the lowest weight that is eligible is chosen. This is done in order to ensure that enough iterations are performed. The time complexity of this method ends up as  $O(N \log N)$  in the end.

### 2.8.2 SFDP

SFDP (Hu, 2005) is a popular graph drawing method that uses multilevel techniques. SFDP is in essence a force directed method, that applies those ideas to smaller graphs that are created by the multilevel layout scheme. While this is similar to what  $fm^3$  does, the multilevel approach used differs between the two methods. SFDP start by using edge collapse. Edge collapse chooses pairs of nodes that are connected with an edge. This is done until no more nodes can be matched with another node. Each pair of nodes is collapsed into a single node. This is repeated until the node count does not decrease far enough within one iteration.

This is not where SFDP stops however. After having performed edge collapse on  $G$ , maximal independent vertex set (MIVS) will be ran on the result. MIVS works by first choosing as many points as possible that all are not connected by an edge. This subset will be all points in the reduced graph. Edges will be places between each node pair whose distance is at most 3. This will be done until only 2 nodes are left in the graph. The final time complexity of SFDP ends up the same as  $fm^3$ ,  $O(N \log N)$ .

## 2.9 Comparing methods

In this section, we compare all methods discussed using multiple criteria. The criteria looked at are the following:

- **Layout quality:** A general idea of the quality of the generated layouts using a given method. This will be somewhat imprecise, as it is not only hard to define what a "good" graph layout is, but also hard to compare multiple methods without running them on the same dataset. The scale used to compare the various methods consists of  $\{-, -, /, +, ++\}$ , where  $-$  is the lowest value, and  $++$  the highest, with  $/$  as a mid point.
- **Scalability:** This looks at how well a method can handle large graphs. This is mostly determined by the time complexity of a given method. Linear method are able to handle large graphs, while quadratic methods run into problems on large graphs.
- **Requires training:** Whether or not a method requires the user to have a trained model before they can generate graph layouts using the method.
- **Ease of choosing parameters:** Some methods require fine tuning of multiple parameters to get a good results. This makes it harder for an end user to easily use the method.
- **Ease of implementation:** How easy it is to understand and implement a given graph layout method. The scale used in the evaluation for the criteria is the same as the one used for the layout quality.
- **Handles weights:** Whether or not the method can take edge weights into account. If a value of "Can be added" means that adding support for edge weights can be done trivially (e.g by taking them into account when calculating graph theoretical distances) but has not been tested, and thus does not guarantee good results. No is given if it would require a more significant change to the algorithm to support it.

Not all of these criteria will be easily extracted from their papers, and will thus not be filled in. Some other criteria are a bit fuzzy and imprecise, like how "good" the resulting layouts are. Other criteria like how easy a method is to understand and implement are subjective, and are therefore up for debate. Table 2.2 is discussed in the following section.



Method	Layout quality	Scalability	Requires training	Ease of choosing parameters	Ease of implementation	Handles weights
tsNET	++	$O(N^2)$	No	1 (perplexity)	/	Can be added
PMDS	/	$O(N)$	No	0 (optional 1)	+	Can be added
DRGraph	++	$O(N)$	No	5 (most can be default)	---	More Challenging
HDE	---	$O(N)$	No	0 (1 optional)	+	Can be added
SGD	+	$O(N^2)$	No	2	++	Yes
$(SGD)^2$	+	$O(N^2)$	No	1 or more target criteria	+	Yes
$(DNN)^2$	+	$O(N)$ , fixed max size (set before training) ( $O(N^2)$ training)	Yes	0	/	Can be added
Graph- TSNE			Yes	1 (perplexity)	-	Can be added
Deep- Drawing	/	$O(N)$ (Not validated for graphs bigger than 50)	Yes	Multiple NN parameters	/	Can be added
DeepGD	+	$O(N)$ , fixed max size (set before training) ( $O(N^2)$ training)	Yes	1 or more target criteria	/	Yes
fm3	+	$O(N \log(N))$	No	0	-	Yes
SFDP	+	$O(N \log(N))$	No	0	-	No
NNP-NET	++	$O(N)$	Training part of pipeline	8 (Can all stay default)	-	Yes

TABLE 2.2: Table comparing methods discussed in the previous sections

### 2.9.1 Comparison evaluation

A top priority for the method will be very good layout quality. tsNET and DRGraph both score very well in this regard. tsNET main problem is scalability, as it is unable to process large graphs. DRGraph however is able to process large graphs. DRGraph has other downsides. For example incorporating edge weights into DRGraph is significantly harder than for other methods. DRGraph also has significantly more parameters than other methods. While HDE scores very well on a lot of metrics, it has the worst layout quality. This could be remedied by using a different DR method than PCA, which would make it a very strong candidate.

Both SGD and  $(SGD)^2$  suffer from bad scalability. Aside from that, they do score very well on all criteria expect parameters. They are also one of the few methods that have been tested using edge weights. The deep learning methods do have a linear inference time. The training time is most of them is  $O(N^2)$ , making it very costly to train a network that is able to handle graphs of that size. The quality of the resulting layouts also fall short of both tsNET and DRGraph.

Not all methods present in table 2.2 are used for the evaluation of our method. Only methods with a time complexity better than quadratic are tested against, with the exception of tsNET. A lot of bigger graphs are used in the comparisons between methods. Methods with a quadratic time complexity will not be able to create a layout for these larger graphs, making them not useful for comparisons. HDE is also excluded, as it is an old method that did not get good results overall.

## Chapter 3

# Method

The basic idea of our method is to speed up tsNET calculations using NNP in place of t-SNE. This allows us to get similar quality graph layouts in linear time instead of quadratic. However, NNP is not a drop-in replacement for t-SNE. There are two main problems that need to be solved in order to use NNP in place of t-SNE for tsNET. These problems are the following:

- What input would be used for NNP?
- On what data would NNP be trained?

In the next sections, we take a closer look at these two problems individually and see what the requirements for a potential solution are. Figure 3.1 shows an overview of the pipeline that is explained in the following sections.

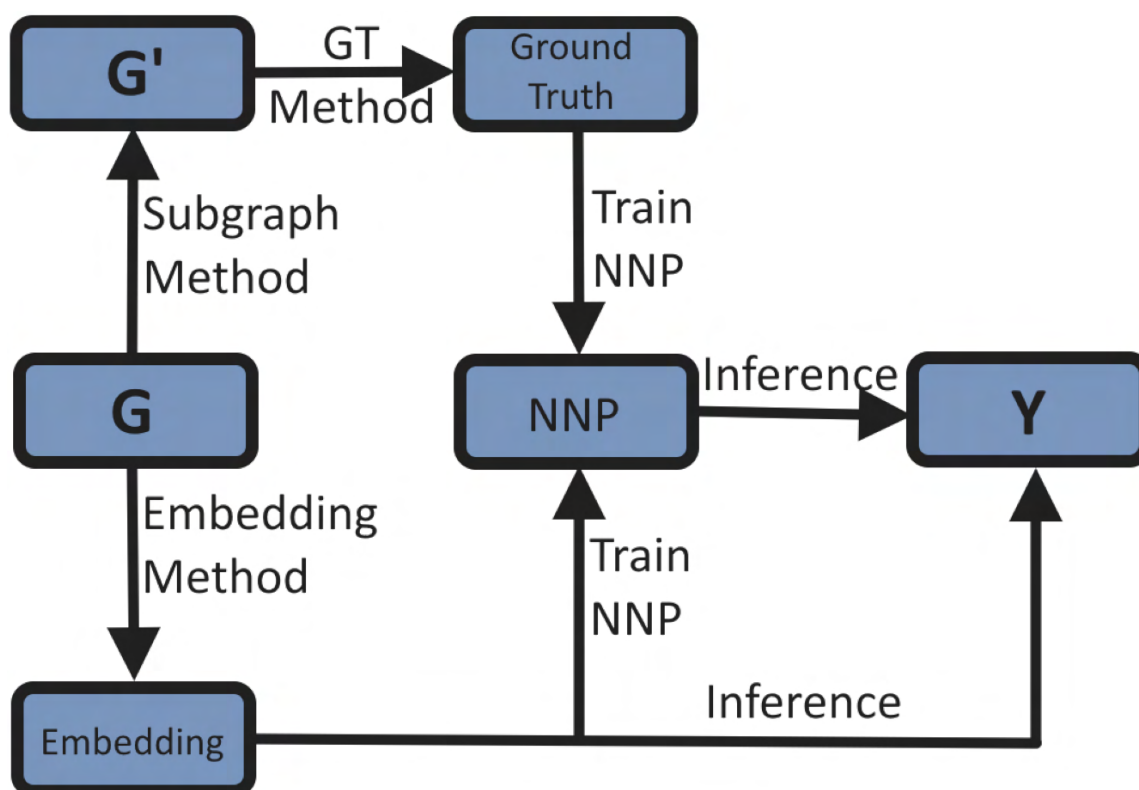


FIGURE 3.1: Overview of NNP-NET that will be explained in the following sections.

### 3.1 Embedding method

Firstly, even though tsNET allows for both feature vectors and a complete distance metric as input, NNP specifically requires feature vectors as input. The graphs used for graph drawing do not provide a feature vector per node in most cases, meaning that an embedding needs to be created per node to function as a feature vector. This embedding has to fulfill a few requirements in order to be useable for this application:

- **Scalability:** The goal is to keep the entire method (near) linear in time complexity. The embedding needs to be created for every node in the graph. This mandates that the creation of the embedding for each node is not dependent on the size of the graph in any way. This rules out any embedding method that would use the entire distance matrix for example, as it would not scale good enough.
- **Out of Sample ability:** NNP needs to be able to learn how to translate the embeddings into layout positions. More crucially, it needs to be able to generate a position for a node that was not in the ground truth by interpolating between positions that were in the ground truth data. NNP will only be able to do this if the embedding contains the nodes position in the graph in some way that is learnable by a neural network. This is hard to measure, but should become apparent by testing how well NNP can learn the embedding method.

Any embedding method that fulfills these requirements should work well for this method. Some of these requirements are not trivial to work out without testing them, so that is the main way we show that the embedding we end up choosing is viable. Next, we will discuss the two potential embedding methods that were considered.

#### 3.1.1 Pivot Point Embedding

The first potential method is the same method that is used by PMDS (Brandes and Pich, 2007) and the high dimensional embedding paper ((HDE)). First, a random first point is chosen. All subsequent points are chosen by taking the point with the lowest maximum distance to the already chosen points. This is done until the required number of points are chosen. The resulting embedding is then the distance from a point to all pivot points.

The time complexity of this embedding method is  $O(Nn)$ , where  $n$  is the embedding size and  $N$  the number of nodes in  $G$ . This is in line with our time complexity requirements for the embedding method. Creating an  $n$  dimensional embedding for  $N$  points can by definition not be faster than  $O(Nn)$ . This embedding method does not guarantee that all embeddings will be unique, although this only happens in specific cases. Consider a graph with four nodes arranged in a square. If two pivot points are chosen, than those will be two opposite corners. The other two corners would then end up with the same embedding, namely being 1 distance away from both pivot points. Not much can be said at the moment about the out of sample support of this embedding method without testing it and comparing to other methods.

#### 3.1.2 Embedding using Graph Layout

An alternative is to use a different, fast graph layout algorithm to generate an embedding in  $n$  dimensions. The value for  $n$  should be higher than the output dimensions of NNP. The logic behind this method is that the fast method might not be able to generate the same quality of layouts as our method, but by giving it more dimensions to work with, it will be able to generate a better layout. This layout is then used as the embedding required

for NNP, which in turn translate the high  $n$  dimensional graph layout into the target  $m$  dimensional layout.

Any graph layout technique that fulfills the criteria needed for this embedding method should work, as long as it can output any arbitrary number of dimensions. For our test we chose to use PMDS, as it very fast and should work well for this purpose. It is important to keep in mind however that PMDS can be swapped out for any other graph layout algorithm without it affecting the rest of the pipeline.

## 3.2 Ground truth

NNP needs to be trained on a ground truth graph layout that it will emulate. Using a pretrained network would be hard, as it has to be trained on very similar network in order to work. The embedding method might also not guarantee that every individual dimension of the embedding means the same thing from graph to graph, making a pre-trained network not a viable option. This forces us to train NNP every time the user wants to create a graph layout. Using a ground truth that contains all nodes from a graph is not a logical solution, as that would require a already complete graph layout to emulate. But if the user already has a graph layout that they like, why would they try to emulate it with our method? Because of this, the decision was made to take a small portion of  $\mathbf{G}$ , a subgraph  $\mathbf{G}'$ , and train NNP on the layout created by running tsNET\* on  $\mathbf{G}'$ .

This changes our question from "On what data do we train NNP" to "Which nodes do we use to generate a ground truth layout using tsNET\*". The method for choosing the points for  $\mathbf{G}'$  comes with the following requirements:

- **Scalability:** It is important that the time complexity does not exceed linear complexity, just as it is for the embedding method. In this case however, the time complexity for a single point can be linear without breaking linear time complexity. This is because the number of points chosen does not have to be dependent on the input size, making it a constant value. A linear time complexity per chosen point will be slow for choosing a lot of points from large graphs, so is ideally avoided.
- **Distance preservation:** Ideally, the distance between two nodes in  $\mathbf{G}'$  is the same as the distance between the same nodes in  $\mathbf{G}$ . Important to note however is that this is only possible by finding the distance from each subpoint to all other subpoints in  $\mathbf{G}$ , which is  $O(N)$  per node, where  $N$  is the size of  $\mathbf{G}$ . The accuracy of the distances in  $\mathbf{G}'$  would have to be compromised in order to avoid this time complexity cost.
- **Representativeness:**  $\mathbf{G}'$  needs to be a good representation of the original  $\mathbf{G}$ . Running tsNET on  $\mathbf{G}'$  has to result in a similar structure as running tsNET on  $\mathbf{G}$ . To accomplish this, all parts of  $\mathbf{G}$  need to be sampled evenly.

A tradeoff between run time and accuracy becomes apparent when looking at these requirements. Next, we look into two potential methods for creating subgraph  $\mathbf{G}'$  from  $\mathbf{G}$ , where each one focusses on a different aspect. Both of the subgraph methods will get their own named variant of NNP-NET, namely NNP-NET-p when using the pivot subgraph method and NNP-NET-c when using the coarsening subgraph method.

### 3.2.1 Pivot (NNP-NET-p)

This is the same method as was explained in 3.1.1 and is also used in PMDS (Brandes and Pich, 2007) and the HDE paper (Harel and Koren, 2002). Here, each pivot point is one point in  $\mathbf{G}'$ . This method has a time complexity of  $O(Np)$  where  $p$  is the number of nodes

in  $\mathbf{G}'$ . Using this method, all points in  $\mathbf{G}'$  will be spaced as far apart as possible, resulting in all parts of  $\mathbf{G}$  being sampled evenly. The distances are also perfectly preserved. The time complexity however might limit the size of  $\mathbf{G}'$  in order to be viable, which likely has a negative impact on the overall performance of this method.

### 3.2.2 Coarsening (NNP-NET-c)

An alternative is to create  $\mathbf{G}'$  by repeatedly coarsening  $\mathbf{G}$  until the number of nodes reaches the target number of nodes. This coarsening is similar to the multi-level approach used by DRGraph (Zhu et al., 2020). We start with the full graph  $\mathbf{G}$  ( $\mathbf{G}^0$ ). A coarsening iteration will then be performed to obtain the next graph  $\mathbf{G}^1$ . This is repeated until a target is reached, from where the last graph can be returned to be used as the subgraph  $\mathbf{G}'$ .

The basic idea of the coarsening steps is to choose a node  $v_i$  from the current graph  $\mathbf{G}^j$ . This node must not be part of the next graph  $\mathbf{G}^{j+1}$  yet. The first order neighbors of  $v_i$  together with  $v_i$  are replaced by a new node, with edges towards all nodes that are connected to the first order neighbors of  $v_i$ , and subsequently is added to  $\mathbf{G}^{j+1}$ . The weights of the edges will also be updated to reflect the distance needed to reach  $v_i$  instead of to the original node that it went to. The iteration will stop once all nodes from  $\mathbf{G}^j$  are represented in  $\mathbf{G}^{j+1}$ .

In other applications, coarsening would stop once the number of nodes does not decrease significantly anymore. More formally, no more iterations are performed once  $\rho \|\mathbf{V}^j\| < \|\mathbf{V}^{j+1}\|$ , where  $\rho$  is a user defined variable with a value between 0 and 1. This stopping method can not be used in our case however, as a certain target value needs to be reached. A problem arises however on certain graphs if you try to reach a specific target value, as the coarsening can become very inefficient at later stages. In the worst case, the graph will collapse into a stargraph, where all nodes are only connected to a center node. When performing the coarsening on a star graph, either only one node will be removed (if a node other than the center node is chosen), or alternatively, all nodes can be collapsed into a single node (if the center node is chosen). This situation needs to be avoided at all cost, as both options are undesirable.

The order in which nodes are chosen to be coarsened was changed in order to mitigate this problem. Before, the nodes were chosen randomly. This was changed to use the degree of the node, where we start with the node with the lowest degree, and end with the node with highest degree. The nodes are sorted using radix sort, in order to keep the time complexity linear. Using this ordering results in the first few iterations reducing the node count slightly less than with a random ordering, but in turn allows for more iterations. This reduces the problem, but does not solve it. While more graphs are able to reach the target node count without problems using this method, some still display undesirable behavior. In these cases, we bring back the original stopping method, with  $\rho$  set to 0.95. The target number of points still needs to be reached however. This is done by giving the result from the coarsening to the pivot subgraph method. The slower runtime of the pivot method is less of a concern in this case, as the graph is already smaller than when we started.

The runtime complexity for each iteration is  $O(N)$ . While this is not better than the pivot method, the constant multiplier is lower, resulting in significantly better real world performance. The distances between nodes does get distorted however, as the paths computed have to go through the nodes that end up in  $\mathbf{G}'$ . This reduces the number of paths available, and can get rid of the true shortest path. Bigger subgraph can be created with this method however, which might offset the quality loss coming from the distortion in the distances. This is tested later on to see which of these methods performs better.

### 3.3 Smoothing

Early results showed that the layouts generated using NNP contained a significant amount of high frequency noise. The layout resulting from NNP are smoothed in order to combat the noise. The smoothing method chosen is a simple laplacian filter. Laplacian smoothing is performed by setting the position of each node to the average each of its first order neighbors as follows:

$$\mathbf{Y}'_i = \frac{1}{\|\mathbf{V}^i\|} \sum_{\mathbf{Y}_j \in \mathbf{V}^i} \mathbf{Y}_j. \quad (3.1)$$

Here  $\mathbf{V}^i$  are the first order neighbors of  $v_i$ , and  $\mathbf{Y}'_i$  is the new position of  $v_i$  after smoothing. This smoothing can be repeated any number of times.

### 3.4 Implementation details

In this section, we will take a look at some details about the implementation used for the tests done in this thesis. The method was implemented in c++. The PMDS implementation used was taken from OGDF and modified to allow for more output dimensions than 3. The NNP-NET and tsNET\* implementation created for this thesis can be found [here](#).

#### 3.4.1 tsNET

tsNET(\*) has been reimplemented for use in this thesis. This was done because the original implementation is fairly slow. Our tsNET implementation is based of the bhtsne implementation (Van Der Maaten, 2014). This implementation contains a faster, approximate t-SNE implementation, which can also be used for our tsNET implementation. The exact version of t-SNE can still be used to generate graph layout if the  $\theta$  that is used by bhtsne is set to a value of 0.

This implementation of tsNET is significantly faster even when  $\theta$  is set to a value of 0. In the original tsNET paper, it took 63.9 seconds to create a graph layout for the `dwt_1005` graph. With our implementation, it took only 1.4 seconds. Different machines were used to obtain these times, making them not directly comparable. But in this case, the difference is large enough to say with certainty that our implementation is significantly faster.

The above comparison only looked at the exact tsNET implementation, while an approximate version is also available. For the approximate variant, a  $\theta$  value of 0.25 was used in all tests. Layouts for graph with a node count of 100000 can be generated using this variant, while the exact variant stopped at 10000 nodes. Both can also be used in the \* variant, where the initial layout is generated using PMDS.

#### 3.4.2 NNP

We originally reimplemented NNP using the `tiny_dnn` c++ library, in order to have NNP nicely integrated in the main code base for the method. It became apparent however that this library does not give the performance that is expected from NNP. Because of this, the switch to `keras tensorflow` was made. While `tensorflow` does have a c++ interface, getting it to work proved to be fairly hard. `Tensorflow` is therefore called from a python script instead of from the c++ code. This python script is ran inside the c++ code using `pybind11`. Using this allows us to directly use the buffers that were created in c++ in the python code, without having to copy any data over.



Tensorflow also allows us to run NNP on the GPU. It was decided after some testing however to not use the GPU for NNP, as it performed slightly worse. While running neural networks on the GPU is almost always a lot faster than running them on the CPU, that was not the case in our testing. This is likely because of the small network size used by NNP. The small network makes the load on the GPU very small, making it so the time taken up by NNP is dominated by the communication speed between the CPU and GPU. While the CPU takes longer to execute the neural network calculations, it does not have to deal with these same communications, making it faster.

	Training time	1 million inference time	10 million inference time
CPU	18 seconds	1 second	11 seconds
GPU	32 seconds	1 second	4 seconds

TABLE 3.1: Performance comparison between GPU and CPU.

Table 3.1 shows both the training time and the inference time of NNP on a randomly generated benchmark. The training was done on a training set with 10000 randomly generated 50 dimensional points. This ground truth labels were 10000 randomly generated 2d positions. The network does not converge to a low error on such a randomly generated dataset, but that does not impact the execution time in any way. The training time is only dependent on the training size, and therefore constant in this test. The input dimension was set to 50, as that is used for the tests later on in this thesis. Inference was done for both 1 million points and 10 million points. This test shows that the training is faster on the CPU, while inference is faster on the GPU. The difference in training time is greater than the difference in inference time however, making the CPU a better option for the data sizes used in this thesis.

Important to note however is that the results are very close, meaning that a different hardware setup might benefit more using the GPU than the one used in this test. Increasing the input size will also increase the work load, which would benefit the GPU more. Because the results are so close, choosing either the CPU or GPU does not have a big impact on the final result.

## 3.5 Complexity analysis

In this section we'll take an in depth look at the time complexity of the method, not just in relation to the input size  $|G|$ , but also to the parameters that have to be chosen.

### 3.5.1 Embedding methods

There are two possible embedding methods proposed. First, lets look at the time complexity for the PMDS embedding:

$$O(npN + p^2N), \quad (3.2)$$

where  $n$  is the size of the resulting embedding, and  $p$  is the number of pivot points used by PMDS. This is the time complexity needed for the eigen decomposition that is part of PMDS. The theoretical lowest time complexity for creating an  $n$  dimensional embedding for  $N$  points is  $O(nN)$ , which the PMDS embedding does reach if the number of pivot points used by PMDS is seen as a constant.

The alternative is to use the pivots directly as the embedding. Such a pivot embedding would have the following time complexity:

$$O(nN). \quad (3.3)$$

This is theoretical minimum, and can not be improved. This method will be faster than using the PMDS embedding, as the work that needs to be done for the pivot embedding is a subset of the work that is done for the PMDS embedding. Whether this performance difference is significant enough to make it the better option will have to be tested.

### 3.5.2 Creating $G'$

There were two proposed methods to create  $G'$ . Let's first take a look at the time complexity for the coarsening method:

$$O(iN), \quad (3.4)$$

where  $i$  is the number of iterations needed to reach the desired size. The value of  $i$  will differ significantly between graphs and is hard to estimate. From testing, it can range anywhere from 2 to 100. Important to keep in mind however is that later iterations take less time, as the size of the graphs has been reduced by that point. This time complexity also only holds if the graph can be reduced with this method. As explained earlier, if the coarsening method fails to reduce the graph to an adequate size, it will switch over to the pivot method. The pivot method for generating  $G'$  has the following time complexity:

$$O(|G'|N). \quad (3.5)$$

The speed of the pivot method depends heavily on the target number of subpoints for  $G'$ . If the pivot method is called from the coarsening method, then the  $N$  will be the lowest size of  $G'$  that the coarsening method was able to accomplish. Especially for larger sizes of  $G'$ , the coarsening method will be significantly faster. Even though the pivot method technically has a linear time complexity, it will likely take quite a long time on large graphs when reducing to a decently sized  $G'$ .

### 3.5.3 Generating the Ground Truth Layout

The ground truth layout gets generated by running the approximate version of tsNET\* on  $G'$ . This has the following time complexity:

$$O(|G'| \log(|G'|)). \quad (3.6)$$

This time complexity indicates that generating the ground truth information is a constant cost, assuming that a constant value for  $|G'|$  is used. It also becomes clear why  $G'$  is needed, as the approximate tsNET has a time complexity that is larger than linear. From later tests it becomes apparent that it is not able to handle graphs larger than about 150,000 nodes.

### 3.5.4 Training and inference

Training NNP has the following time complexity:

$$O(|G'|n * \text{epochs}), \quad (3.7)$$



Where *epochs* is the number of training epochs used. Important here is that time needed for training is completely independent from the size of  $\mathbf{G}$ . This is in contrast with the inference time, which has the following time complexity:

$$O(nN). \tag{3.8}$$

Inference has a linear time complexity with respect to the number of nodes inferred, which should not result in any slowdown. As earlier shown in the section 3.4.2, both the inference and training only take a few seconds, even on large datasets.

## Chapter 4

# Results

In this section we will take a look at the results of the different embedding and subgraph methods, as well as compare the complete pipeline to other graph layout algorithms. The full list of graphs used in this section is given in table 4.1. The graphs used are a combination of graphs used in other papers like the tsNET and DRGraph paper. They are sourced from the suitsparce collection (Davis and Hu, 2011). Most graphs used are on the larger side, with a few small graphs added. This is done because our method is most useful when working with larger graphs, as it will always be slower on smaller graphs than tsNET, without an improvement in quality.

Name	V	E	Weighted?
dwt_1005	1005	3808	no
sierpinski3d	2050	6144	no
MISKnowledgeMap	2427	28511	yes
3elt	4720	13722	no
optdigits_10NN	5620	39825	yes
fe_4elt2	11143	32818	no
bcsstk36	23052	1143140	no
k49_norm_10NN	38547	309079	yes
fe_bcsstk32	44609	985046	no
m_t1	97578	9753570	no
ship_003	121728	3777036	no
fe_ocean	143437	819186	no
ok2010	269118	1274148	yes
web-NotreDame	325729	1497134	no
coPapersCiteseer	434102	16036720	no
gsm_106857	589446	21758924	no
tx2010	914231	4456272	yes
com-Youtube	1134890	5975248	no
Flan_1565	1564794	59485419	no
com-LiveJournal	3997962	34681189	no
asia_osm	11950757	25423206	no

TABLE 4.1: Graphs used in the tests.

The dataset contains less weighted graphs than desired. This is because the selection of good weighted graphs is fairly limited. The Suitesparce collection contains three large weighted graphs collections that were suitable (Fully connected, undirected and symmetrical): DIMACS10, Gset and ML\_Graph. Of these, DIMACS10 and ML\_Graph are good candidates. Gset is a collection of randomly generated graphs. We decided to only use graphs from real world data. The largest graph Gset is also only 10000, which is on the smaller side for the graphs used in this thesis. There are some other smaller sets of

weighted graphs, but not all of them clearly say what the weights mean. Our pipeline interprets the weights as a target length. This means that we need to be able to translate/interpret the given weights into a target length in order to use them. It was not clear how to do this for some weighted graphs. Two graph from both DIMACS10 and ML\_Graph were chosen, as more graphs from the same dataset would have been redundant.

The tsNET implementation used for the testing is our own implementation, based on the bhtsne implementation. If the approximate version of tsNET was used, then the  $\theta$  value was set to 0.25. The PMDS used is the implementation present in our code base, which was originally taken from OGDF (Chimani et al., 2013). OGDF was used for the  $fm^3$  implementation. All parameters used were the default used by OGDF. The SFDP implementation used was from GraphViz. For DRGraph, the implementation provided by the original authors was used. The parameters used for DRGraph were the recommended parameters provided by the authors. All tests were run on the CPU using a system with a Intel i7-11370H CPU, NVIDIA 3050 ti laptop GPU and 16 gigabytes if RAM.

Unless stated otherwise, coarsening was used as the subgraph method, and PMDS was used to create the embedding. The default parameters used for NNP-NET-c can be seen in table 4.2.

Parameter	Value
$ G' $	10000
$n$ (embedding size)	50
Smoothing Passes	3
Perplexity	40
$\theta$	0.25
PMDS Pivot Points	250
Batch size	64
Training epochs	40

TABLE 4.2: Parameters used for NNP-NET-c.

## 4.1 Visualization

First some notes on the visualization used in this thesis before getting into the results. To start, nodes are colored black while edges are colored blue. Because of the high density of some of the larger graphs, an alpha value was used to increase visual clarity. The alpha value was set dynamically based on the number of nodes in the graph, using  $alpha = \min(1, 10000/|G|)$ .

Some of the graphs had a large cluster of points, with a few points relatively far away from that cluster. This created images with a large portion of whitespace. We allowed for 1% of the nodes to be outside of the image in order to mitigate this problem. This is accomplished by first creating the smallest possible square box the fits all nodes. We then repeatedly loop over all 4 corners of the box. Each corner tries to move 10% closer to the opposite corner. This will be rejected if the movement causes more than 1% of the nodes to be outside of the box. This loop is stopped once none of the corners can move closer without breaking the constraint.

## 4.2 Embedding Method

Firstly, we'll compare the two different embedding methods, the pivot embedding (section 3.1.1) and the PMDS embedding (section 3.1.2). Both of the embedding methods were run with an embedding size set to 50. The time recorded is only the time that it takes to compute the embedding, as the rest of the pipeline will run at the same speed for both embedding methods. The error to the ground truth is the average euclidean distance between a point  $Y_i$  in the ground truth and the corresponding  $Y_i$  in the result generated using NNP-NET-c. No smoothing was applied to any of the results, as this could have hidden some of the imperfections making it harder to visually compare the results. The neighborhood preservation is also recorded, although the error to ground truth is the more important metric in this case. This is because NNP's goal is to imitate the original layout as closely as possible. The error to the ground truth accurately encapsulate how close the generate layout is to the ground truth, while neighborhood preservation only represents how good a given layout is.

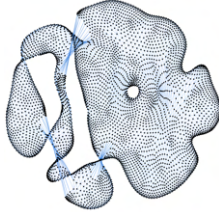
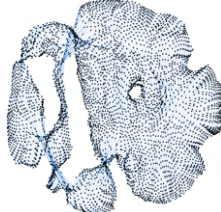
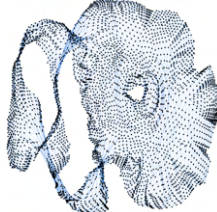
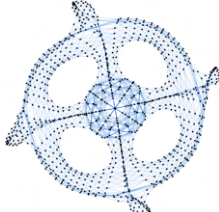
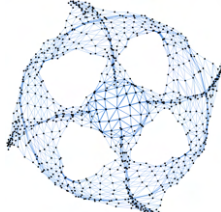
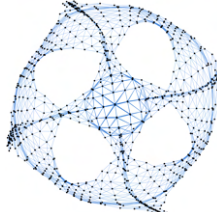
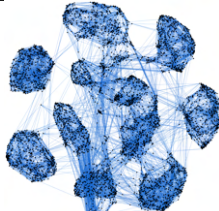
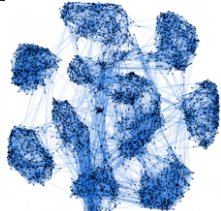
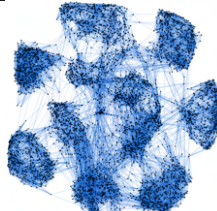
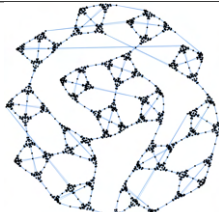
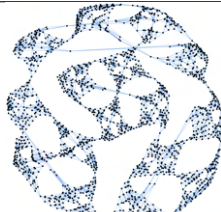
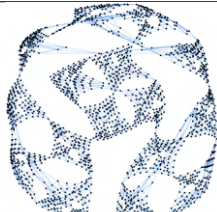
Graph	tsNET* Ground Truth	PMDS embedding	Pivot Embedding
3elt			
dwt_1005			
optdigits_10NN			
sierpinski3d			

TABLE 4.3: Visual results trying to recreate a ground truth using different embedding methods.

Graph	embedding	Time	Neighborhood preservation	Error to GT
3elt	PMDS	1.228	0.547	0.014
3elt	Pivot	0.015	0.499	0.024
dwt_1005	PMDS	2.253	0.521	0.019
dwt_1005	Pivot	0.007	0.484	0.027
optdigits_10NN	PMDS	6.649	0.598	0.017
optdigits_10NN	Pivot	0.799	0.586	0.025
sierpinski3d	PMDS	3.487	0.498	0.017
sierpinski3d	Pivot	0.005	0.462	0.028

TABLE 4.4: Results trying to recreate a ground truth using different embedding methods.

Table 4.3 shows the graphs resulting from using the two different embedding methods, while table 4.4 shows the corresponding metrics. The PMDS embedding outperforms the pivot embedding slightly on all graphs tested. The PMDS embedding does take more time to compute. While the time needed for the PMDS embedding is significantly slower, this does not necessarily have to be a problem, as it is only a part of the total time needed for the entire pipeline. The PMDS embedding is fast enough, while getting results that are closer to the ground truth.

### 4.3 Embedding Dimensions

With the following test, we look at the effect of changing the dimensions  $n$  used for the PMDS embedding. The values used for  $n$  in this test are 10, 25, 50 and 100. For each value of  $n$  for each graph, the neighborhood preservation, execution time and error to ground truth are recorded. The error to the ground truth is the more important metric in this case, as we are trying to recreate the tsNET layout as closely as possible. If we can recreate the ground truth as closely as possible, than the neighborhood preservation will be good as well (unless the ground truth used is of low quality).

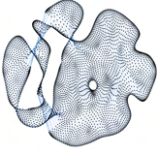

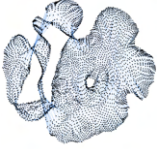
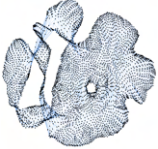
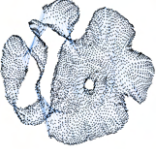
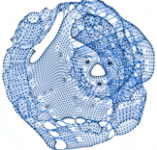
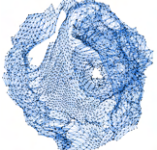
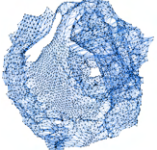
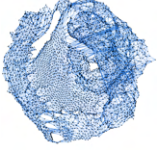
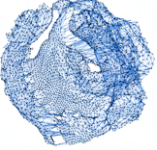
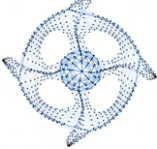
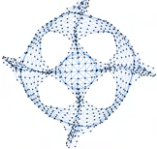
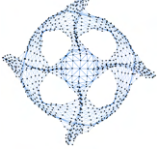
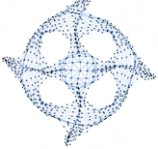
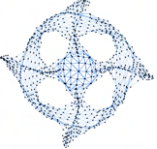
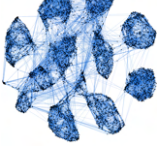
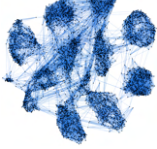
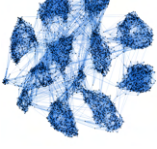
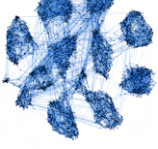
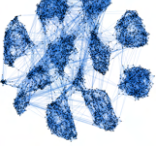
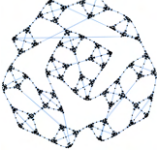
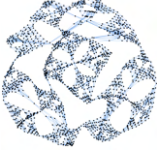
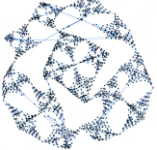
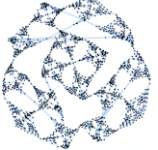
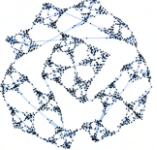
Graph	Ground Truth	$n = 10$	$n = 25$	$n = 50$	$n = 100$
3elt					
bcsstk36					
dwt_1005					
optdigits_10NN					
sierpinski3d					

TABLE 4.5: Visual results using a different number of dimensions  $n$  for the embedding used by NNP-NET-c.



Graph	embedding dim	Neighborhood preservation	Error to GT	Time (seconds)
3elt	10	0.532	0.013	15.5
3elt	25	0.580	0.019	20.9
3elt	50	0.579	0.011	38.2
3elt	100	0.574	0.012	105.7
bcsstk36	10	0.411	0.027	30.2
bcsstk36	25	0.435	0.019	36.3
bcsstk36	50	0.457	0.014	54.4
bcsstk36	100	0.463	0.016	118.9
dwt_1005	10	0.504	0.018	5.5
dwt_1005	25	0.542	0.012	11.3
dwt_1005	50	0.543	0.012	28.8
dwt_1005	100	0.544	0.012	94.0
optdigits_10NN	10	0.607	0.029	57.7
optdigits_10NN	25	0.621	0.019	56.6
optdigits_10NN	50	0.623	0.015	45.6
optdigits_10NN	100	0.624	0.014	106.9
sierpinski3d	10	0.495	0.023	10.7
sierpinski3d	25	0.509	0.019	17.8
sierpinski3d	50	0.507	0.021	35.6
sierpinski3d	100	0.502	0.014	99.2

TABLE 4.6: Performance metrics for the different embedding sizes.

Looking at table 4.5, there is not a very large visual improvement when using more dimensions compared to using only 10. From table 4.6 it does become apparent however that there is an improvement in performance using more dimensions looking at the performance metrics. Both the neighborhood preservation and error to ground truth are better using more dimensions. The difference between 50 and 100 dimensions is very small for all graphs tested with sierpinski3d as an exception. From these results, giving  $n$  a value higher than 50 does not seem necessary.

This is further strengthened when looking at the execution times. Execution seems to scale non-linearly with  $n$ , which is surprising, as nothing in PMDS suggests that it scales worse than linear with respect to the number of output dimensions. After some investigation, this behavior comes from the eigen decomposition done by the PMDS implementation used. While each eigen decomposition iteration is linear with respect to  $n$ , more iterations are needed to archive the desired results when using higher values for  $n$ . The performance impact of using a higher  $n$  value are not consistent between different graphs. For all tests going forward, the value of  $n$  used will be 50 in order to make sure the results are of high quality, at the cost of some performance.

## 4.4 Subgraph Size

Next, we'll look at the performance of different sizes of  $G'$ . These tests were all performed using the coarsening subgraph method, as it is significantly faster for generating bigger subgraphs. The metrics recorded for these tests are neighborhood preservation and execution time.

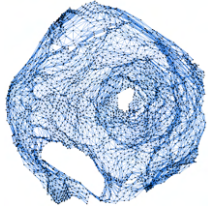
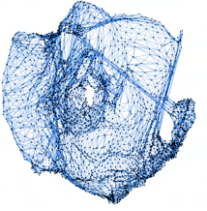
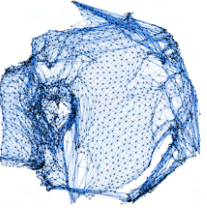
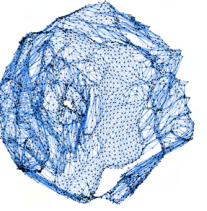
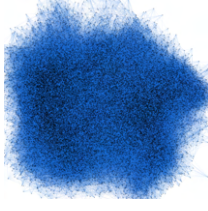
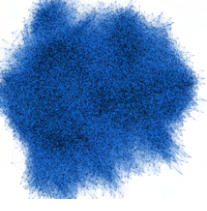
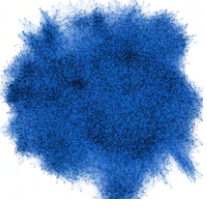
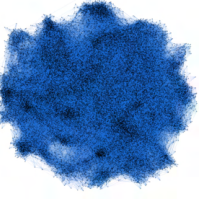
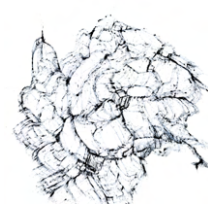
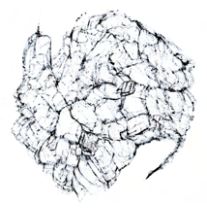
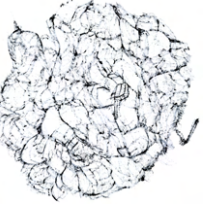
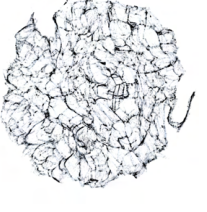
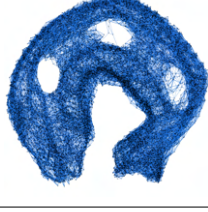
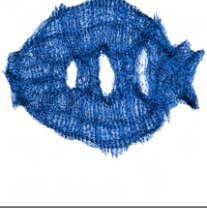

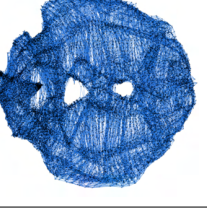
Graph	$ G'  = 1000$	$ G'  = 2500$	$ G'  = 5000$	$ G'  = 10000$
bcsstk36				
k48_norm_10NN				
ok2010				
ship_003				

TABLE 4.7: Visual results using different sizes for the  $G'$ .



Graph	subgraph size	Neighborhood preservation	Time (seconds)
bcsstk36	1000	0.411	32.4
bcsstk36	2500	0.373	41.4
bcsstk36	5000	0.340	58.2
bcsstk36	10000	0.408	103.6
k49_norm_10NN	1000	0.030	74.8
k49_norm_10NN	2500	0.047	102.6
k49_norm_10NN	5000	0.060	127.1
k49_norm_10NN	10000	0.083	219.0
ok2010	1000	0.392	90.2
ok2010	2500	0.408	103.2
ok2010	5000	0.424	107.8
ok2010	10000	0.436	160.2
ship_003	1000	0.155	45.1
ship_003	2500	0.208	56.2
ship_003	5000	0.214	85.0
ship_003	10000	0.229	131.3

TABLE 4.8: Performance metrics for the different sizes for  $G'$ .

Increasing the size of the  $G'$  increases the performance of the entire pipeline in most cases. This size of this effect heavily depends on the graph. Comparing the results shown in table 4.8 for bcsstk36, the neighborhood preservation was nearly the same for both 1000 and 10000 subpoints. In contrast, the neighborhood preservation for k49\_norm\_10NN more than doubled when comparing 1000 and 10000 subpoints. These results can also be visually observed when looking at table 4.7. The results for bcsstk36 are the most unclear when visually looking at them. It is clear that there are different results when changing the number of subpoints, but it is not very obvious which is better. Creating a layout for ship\_003 with 1000 subpoints results in a curve in the structure, which is not supposed to be there. Increasing the number of points used for  $G'$  resolves this issue. k49\_norm\_10NN benefitted the most from increasing the number of subpoints when looking at the metrics. Comparing the layouts generated from 1000 and 10000 points, it becomes clear why the neighborhood preservation increases this drastically for this graph. This graph contains a lot of clustering, which is captured very poorly when using only 1000 subpoints for  $G'$ . With 10000 subpoints, clustering can be observed more clearly, which explains the increase in neighborhood preservation.

Using a bigger  $G'$  does come with a downside however: Increased runtime. This increased runtime does not come from generating  $G'$ . With the coarsening method, generating a larger subgraph is actually faster as it reduces the number of iterations needed. This increase in runtime is due to two factors: Generating the ground truth from  $G'$ , and training NNP. The ground truth is generated using the approximate version of tsNET\*, which has a time complexity of  $O(N \log(N))$ , which heavily contributes to the increase in runtime. Important to keep in mind is that these are constant costs. These cost are independent from the size of  $G$ . If you were to use the pivot method to get  $G'$ , than performance cost for using more points in  $G'$  would scale with the size of  $G$ .

## 4.5 Subgraph Method

Here we'll compare the performance of the two different methods to create  $G'$ . For these tests, a different target size for  $G'$  is used for the methods. For the pivot method (NNP-NET-p), a target size of 1000 is used, while a target size of 10000 is used for the coarsening method (NNP-NET-c). This was chosen as this is the main advantage of the coarsening method. While it does not give an accurate comparison between the performance of each method when using the same settings, it does give a more useful comparison. Namely, this test compares how you would use the two different methods in practice. Saying that the pivot method performs better when using the same target number of points is not very useful if you can not run the pivot method with that number of target points. This test will thus look at what is more important: choosing better points with perfect distance preservation, or having more target points.

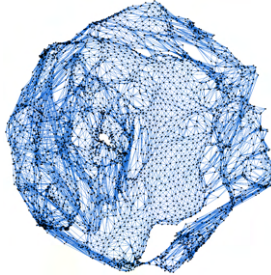
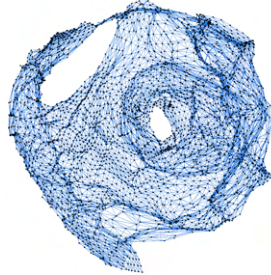
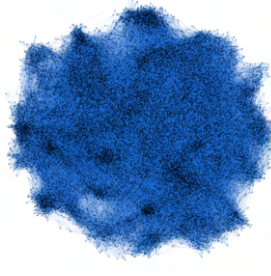
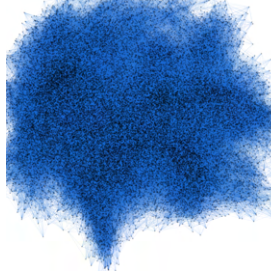
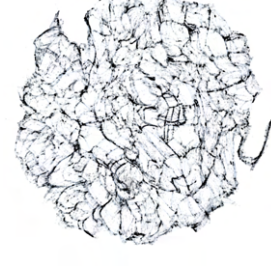

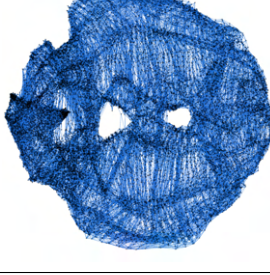
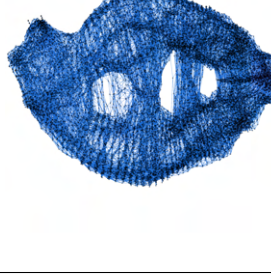
Graph	NNP-NET-c	NNP-NET-p
bcsstk36		
k49_norm_10NN		
ok2010		
ship_003		

TABLE 4.9: Visual results using different methods for creating  $G'$ .

Graph	Method + points	Neighborhood preservation	Time
ok2010	Pivot, 1000 points	0.382	255.4
ok2010	Coarsening, 10000 points	0.436	160.2
ship_003	Pivot, 1000 points	0.194	78.4
ship_003	Coarsening, 10000 points	0.229	131.3
k49_norm_10NN	Pivot, 1000 points	0.045	209.4
k49_norm_10NN	Coarsening, 10000 points	0.083	219.0
bcsstk36	Pivot, 1000 points	0.422	37.9
bcsstk36	Coarsening, 10000 points	0.408	103.6

TABLE 4.10: Performance metrics for the different subgraph method.

it's hard to draw strong conclusions from looking at the resulting layouts in table 4.9. k49\_norm\_10NN seems to have better clustering when using NNP-NET-c. NNP-NET-c also results in more space being utilized for both ok2010 and ship\_003. However, whether this is positive or negative is hard to quantitatively determine.

When looking at table 4.10, NNP-NET-c performs about the same on most graphs, but significantly better k49\_norm\_10NN. With the settings used, NNP-NET-p is faster on smaller graphs, while NNP-NET-c is faster for larger graphs. This is to be expected, as increasing the target size for  $\mathbf{G}'$  is a constant cost which does not depend on the input size  $|\mathbf{G}|$  when using NNP-NET-c. These results indicate that using NNP-NET-c with a higher target size for  $\mathbf{G}'$  is the better option. The execution time is lower for large graph, which is what NNP-NET focusses on, while also having better results on some graphs with similar results on the rest.

## 4.6 Smoothing

While the results are close to the ground truth when looking at the error values, visually, the results contain a lot of noise. To reduce this, a post processing smoothing step is introduced to the pipeline. This gives the following results:

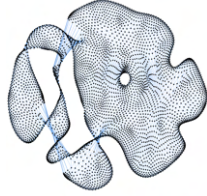
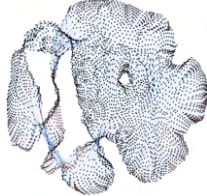
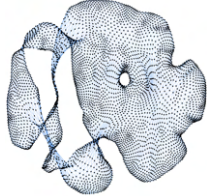
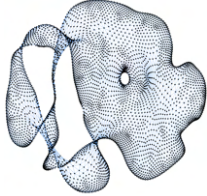
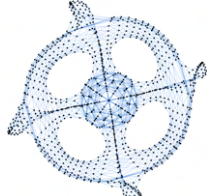
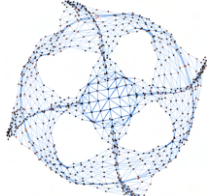
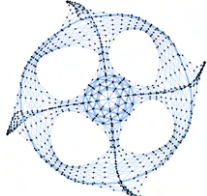
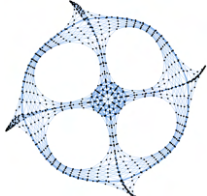
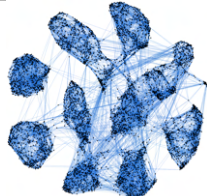
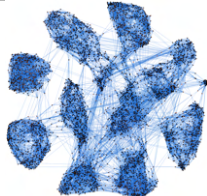
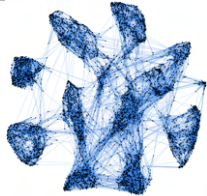
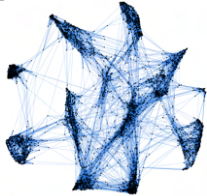
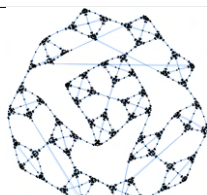
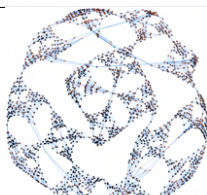
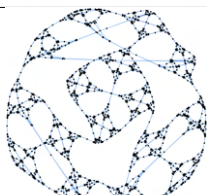
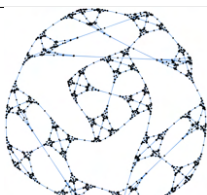
Graph	Ground truth	No Smoothing	2 passes	10 passes
3elt				
dwt_1005				
optdigits_10NN				
sierpinski3d				

TABLE 4.11: Results smoothing the output of our method.

Graph	Smoothing Passes	Neighborhood preservation
3elt	Ground Truth	0.634
3elt	0	0.545
3elt	2	0.599
3elt	10	0.605
dwt_1005	Ground Truth	0.618
dwt_1005	0	0.521
dwt_1005	2	0.520
dwt_1005	10	0.486
optdigits_10NN	Ground Truth	0.628
optdigits_10NN	0	0.622
optdigits_10NN	2	0.615
optdigits_10NN	10	0.595
sierpinski3d	Ground Truth	0.530
sierpinski3d	0	0.495
sierpinski3d	2	0.519
sierpinski3d	10	0.496

TABLE 4.12: Results smoothing the output of our method.

Applying the smoothing pass greatly improves the visual quality of the method, as can be seen in table 4.11. This can already be observed even when using only 2 passes. Using too many smoothing passes gets worse results, as it causes the nodes to get too close to each other. The main purpose of the smoothing step is to create more visually pleasing results, not to increase performance metrics like neighborhood preservation. It is therefore not surprising that, looking at the results in table 4.12, the neighborhood preservation tends to stay at a similar level. Most importantly, this shows that our smoothing passes do not negatively impact our clustering reliability. The neighborhood preservation even went up significantly for 3elt after applying 2 smoothing passes.



## 4.7 Effect of weights

Our method can take edge weights into account. It does this by viewing them as target lengths for the given edge. These lengths are then used when calculating the distances between nodes. The weighted graphs are layed out both with weights and without weights in order to test whether using the weights in this manner has any impact on the resulting layouts. If the resulting layouts change because of the weights, than that implies that the results generate without the weights show an inaccurate representation of the data.

Table 4.13 shows the results from this experiment. The layouts shift around substantially between the runs. This does not have to be because of the weights however. The clearest effect of the weights is visible in both ok2010 and tx2010. When weights are used, a network structure becomes more clear in the layout, where without weights, the resulting layout is more blob-like. This network structure is partially lost when not taking the weights into account, showing that the weights are integrated in a useful way into the pipeline of our method.

Different results do not directly imply correct results however. NNP-NET-c was also ran on a 40 by 40 grid graph with the last 9 edges on one axis having twice the weight as all other edges. The expected result for this graph would be a regular grid with one side stretched out in a single direction. The result of this can be seen in figure 4.1. This image shows that at the right side of the graph, the horizontal edges suddenly become longer at the last 9 edges, showing that they have been taken into account correctly.

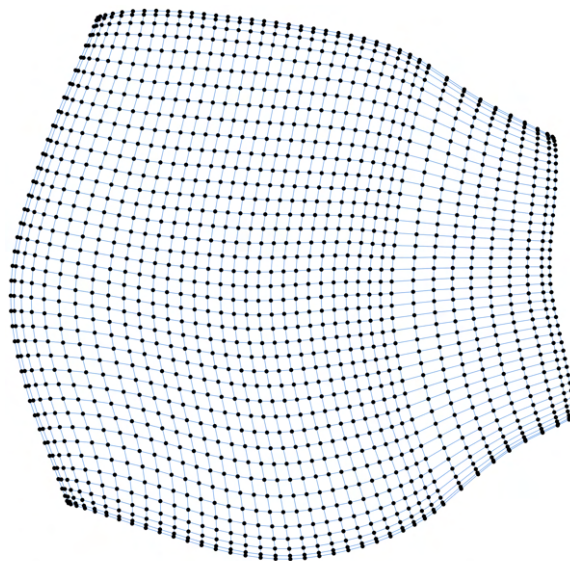


FIGURE 4.1: 40 by 40 grid graph with the last 9 edges on one axis having twice the weight.

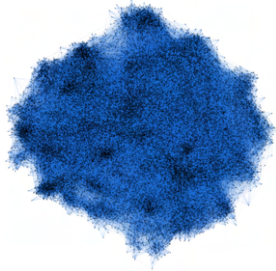
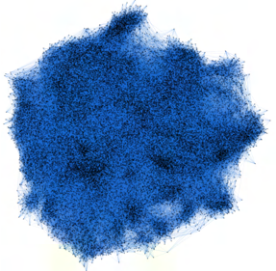
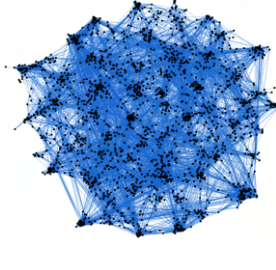
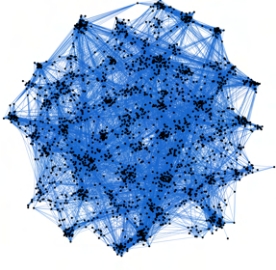
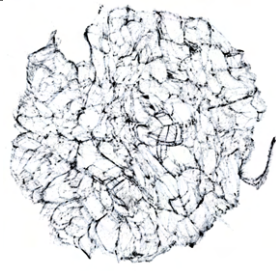
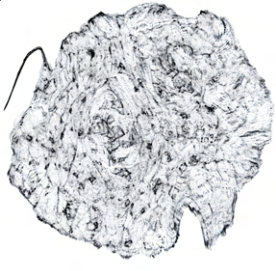
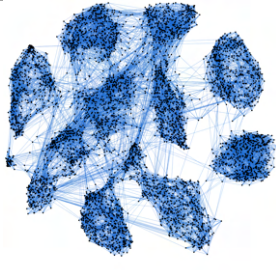
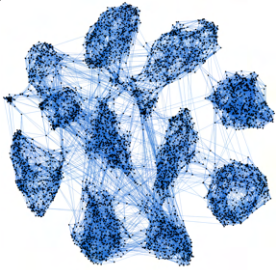
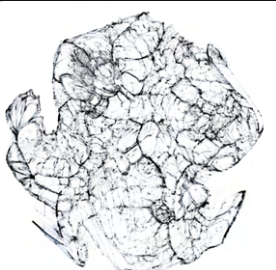
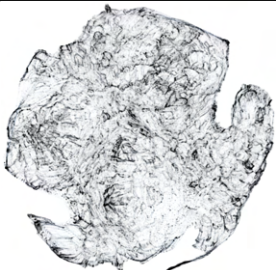
Graph	With Weights	Ignoring Weights
k49_norm_10NN		
MISKnowledge-Map		
ok2010		
optdigits_10NN		
tx2010		

TABLE 4.13: Comparison between layouts created using NNP-NET-c, left took weights into account, while right ignored the edge weights.

## 4.8 Other Ground truth method than tsNET\*

In all tests done up until now, the ground truth was always generated using approximate tsNET\*. NNP is able to imitate any dimensionality reduction method. NNP is also able to imitate graph drawing algorithm tsNET\*, which raises the question whether we could also use any graph drawing algorithm to generate the ground truth. To test this, graph layouts were generated using both SFDP and  $fm^3$  on the entire graph. A subset of these points was then taken using the coarsening method. These points were then used to train NNP, which then tried to recreate the layouts, which gave the following results:

Graph	SFDP truth	Ground truth	Recreated layout	fm3 Ground truth	Recreated layout
bcsstk36					
dwt_1005					
ok2010					
optdigits_10NN					

TABLE 4.14: Results of our method recreating ground truth methods other than tsNET

As can be seen in table 4.14, the results that NNP created are extremely close to the ground truth results. The main difference between the ground truth and the recreated results can be seen in optdigits\_10NN, where the clusters are smaller in the recreation. This is due to the smoothing that is applied, which pushes the nodes closer to each other. Other than that, the results look close to the same visually, which shows that NNP can also recreated multiple different graph layout methods.



## 4.9 Timings

Next, we'll look at the time each individual component of the pipeline takes. There are three different component that will be looked at: Calculating the embedding, getting  $G'$  from  $G$ , generating the ground truth on  $G'$ , and training + inference time. The graphs are ordered based on vertex count, from low to high.

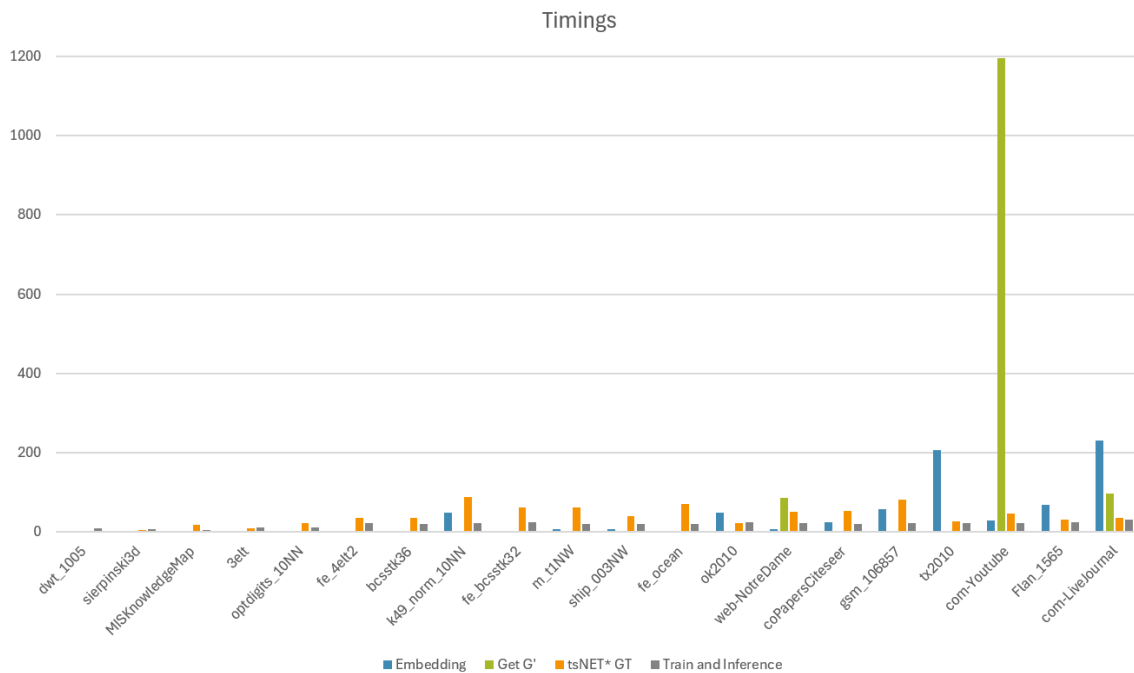


FIGURE 4.2: Chart with execution time per component per graph

Looking at the results in figure 4.2, the time needed for getting subgraph  $G'$  stands out for having a few graphs that do not have results in line with the results for the rest of the graphs. For almost all graphs, getting  $G'$  takes less than a second. There are three graphs where it takes longer: web-NotreDame, com-Youtube and com-LiveJournal. Both web-NotreDame and com-LiveJournal have results in the neighborhood of 100 seconds, while com-Youtube takes more than 1000 seconds to create  $G'$ . These increased execution times are because the coarsening subgraph method was not able to reach the target number of nodes. This results in a fail over to the pivot subgraph method, which is substantially slower. The coarsening method was able to get fairly close for both web-NotreDame and com-LiveJournal, making the time penalty not as big as it is for com-Youtube. These results show that execution time of the coarsening subgraph method is negligible for most graphs, but can be very significant if the graph does not collapse nicely using the coarsening method. Lets remove the subgraph results from the chart to get a better look at the rest of the results.

When looking at figure 4.3, it is important to keep in mind that from fe\_4elt2 onwards the graphs are large enough that  $G'$  is calculated and used. The times for training and creating the ground truth should from then on have a constant time, which is caused by the set size of 10000 used for  $G'$ . This can be clearly observed in the Train and inference times, which follow an almost completely strait line after fe\_4elt2. This is despite the fact the inference times are also included, which are not constant but linear with respect to the input size. The inference time is significantly less in comparison to the training, such that it does not have a visible impact on the results, except for the small uptick at the largest graph.

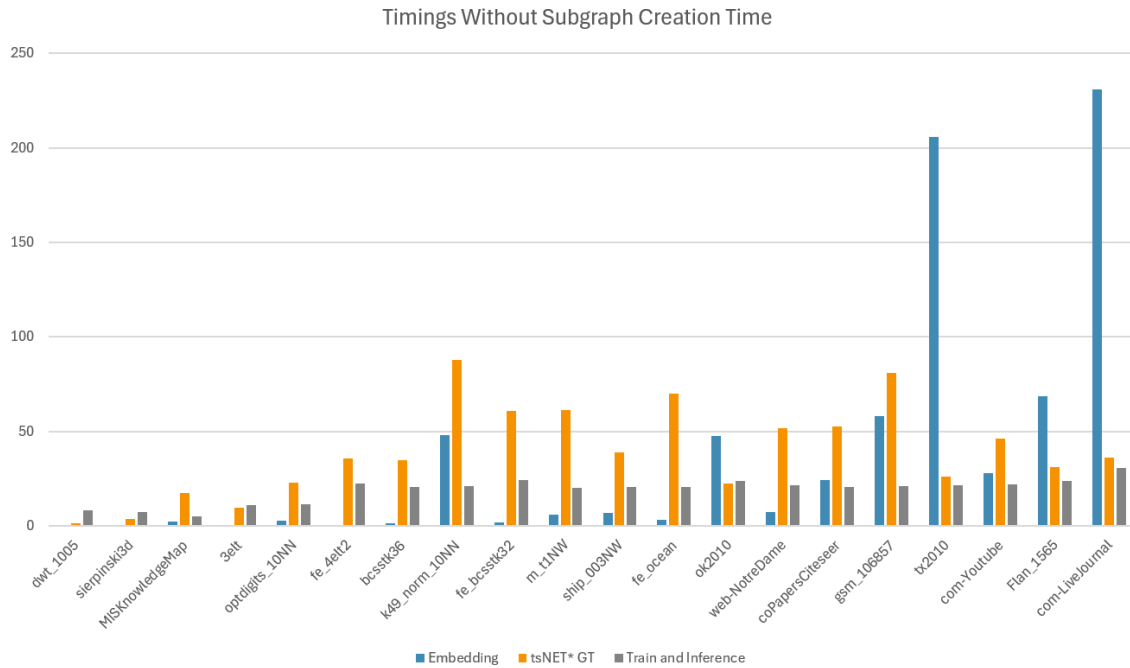


FIGURE 4.3: Chart with execution time per component per graph, with subgraph creation removed

The results for creating the ground truth using the approximate tsNET\* show different results however. While the overall trend might indicate the constant time complexity, it is heavily obscured by a significant amount of noise. Depending on the graph, creating the ground truth might take significantly longer than for other graphs. Embedding creation times also show inconsistent results. This is mainly the case for weighted graphs. Weighted graphs can not use BFS for distance calculations, and have to use Dijkstra instead. This causes significantly longer embedding creation times for the weighted graphs. This causes our method to take a different amount of time for graphs of the same size.

## 4.10 Results compared to other methods

Lastly, the results from our method will be compared against other methods from literature. The methods that are compared against are SFDP,  $fm^3$ , PMDS, DRGraph and tsNET\* (both our exact and approximate implementations). Any empty entry is either because of memory constraints or a very high runtime.

Graph	SFDP	$fm^3$	PMDS	DRGraph	tsNET* (exact)	tsNET* (Approx)	NNP-NET-c
dwt_1005							
sierpinski3d							
MIS-Knowledge-Map							
3elt							
optdigits_10NN							
fe_4elt2							
bcsstk36							
k49_norm_10NN							
fe_bcsstk32							
m_t1							
ship_003							
fe_ocean							

Graph	SFDP	$fm^3$	PMDS	DRGraph	tsNET* (exact)	tsNET* (Approx)	NNP-NET-c
ok2010							
web-NotreDame							
CoPapers-Citeseer							
gsm_106857							
tx2010							
com-Youtube							
Flan_1565							
com-LiveJournal							
asia_osm							

TABLE 4.15: Layouts created by various methods on various graphs

Graph	Execution time per method						
	SFDP	$fm^3$	PMDS	DRGraph	tsNET* (exact)	tsNET* (Approx)	NNP-NET-c
dwt_1005	0.34	0.12	<b>0.02</b>	0.06	1.44	2.61	10.13
sierpinski3d	0.75	0.12	1.37	<b>0.11</b>	6.52	5.42	11.38
MISKnowledgeMap	1.23	0.76	1.20	<b>0.13</b>	40.42	13.71	24.27
3elt	1.71	0.20	<b>0.07</b>	0.27	110.70	18.92	21.14
optdigits_10NN	2.59	0.58	2.38	<b>0.30</b>	155.46	59.68	36.88
fe_4elt2	6.06	0.22	<b>0.15</b>	0.59	931.18	107.97	58.96
bcsstk36	10.89	2.28	<b>0.53</b>	1.49	-	288.91	56.96
k49_norm_10NN	37.28	<b>2.14</b>	37.13	2.66	-	1718.79	156.79
fe_bcsstk32	28.25	3.07	<b>2.47</b>	3.64	-	572.96	86.68
m_t1	77.61	14.22	<b>4.04</b>	8.21	-	1133.24	87.81
ship_003	94.44	14.24	<b>5.85</b>	9.61	-	1456.06	66.40
fe_ocean	117.74	<b>1.80</b>	2.28	14.39	-	-	94.24
ok2010	260.64	<b>4.13</b>	44.83	23.98	-	-	94.18
web-NotreDame	270.48	10.26	<b>6.13</b>	25.49	-	-	166.56
coPapersCiteseer	-	99.34	<b>23.69</b>	45.49	-	-	98.20
gsm_106857	-	57.28	<b>35.35</b>	55.86	-	-	160.41
tx2010	1165.55	<b>12.95</b>	200.21	83.18	-	-	254.17
com-Youtube	2209.56	-	<b>23.47</b>	132.47	-	-	1293.10
Flan_1565	-	-	<b>55.92</b>	171.44	-	-	124.12
com-LiveJournal	-	-	<b>220.14</b>	618.30	-	-	394.85
asia_osm	-	-	-	<b>1647.95</b>	-	-	-

TABLE 4.16: Execution time for creating a layout using a specific method for each of the graphs.

The reasons for the empty entries differ per method. SFDP failed because of memory constraints due to high edge counts.  $fm^3$  also suffered from high memory consumption. PMDS only failed on `asia_osm`, again due to high memory consumption. Exact tsNET\* had both to high memory consumption as well as execution times that would be too long. Approximate tsNET\* took more than an hour for larger graphs. NNP-NET-c, as it uses PMDS, also suffered from to high memory consumption on `asia_osm`.

NNP-NET-c needs significantly longer to create layouts for small graphs than other fast methods need, which can be seen in table 4.16. The execution time does not go up very fast however, and depends heavily on the specific graph. `k49_norm_10NN` for example takes longer than `coPapersCiteseer`, even though `coPapersCiteseer` has more than 10 times as many nodes, and is also slower than creating a layout for `Flan_1565`, which has about 40 times as many nodes. While NNP-NET-c is slower than DRGraph for most graphs, it overtakes DRGraph in performance once the graphs get large enough. NNP-NET-c was faster for both `Flan_1565` and `com-LiveJournal`.

Visually for the smaller sized graphs (table 4.15), the results from NNP-NET-c come very close to the ground truth results from the approximate tsNET\*. Approximate tsNET\* does exhibit weird behavior where all nodes seem to snap to a grid structure on the larger sized graphs that it can still handle. NNP-NET-c lucky does not copy this behavior and continues to output good looking graphs. Looking at the even larger graphs, NNP-NET-c creates results that differ from the other methods, but not necessarily in a bad way. Our results look good, although judging which layout better represents the input data is hard to do by just looking at the output.

Comparing the neighborhood preservation of NNP-NET-c with DRGraph shown in table 4.17, NNP-NET tends to get similar but slight lower results. An exception to this are both `ok2010` and `tx2010`, which have significantly higher results for NNP-NET-c compared to DRGraph on both graphs. Both of these graphs are weighted, and originate from the same dataset, both of which likely are contributing factors to why NNP-NET-c performs so much better specifically on these two graphs. Another important comparison here is with PMDS, as NNP-NET-c uses PMDS as part of its pipeline. NNP-NET-c has a higher



Graph	Neighborhood preservation per graph						
	SFDP	$fm^3$	PMDS	DRGraph	tsNET* (exact)	tsNET* (Approx)	NNP-NET-c
dwt_1005	0.50	0.53	0.47	0.50	<b>0.62</b>	0.59	0.53
sierpinski3d	0.51	0.51	0.20	0.52	<b>0.55</b>	0.53	0.52
MISKnowledgeMap	0.17	0.16	0.13	0.33	0.40	<b>0.41</b>	0.24
3elt	0.62	0.65	0.36	0.63	<b>0.66</b>	0.63	0.63
optdigits_10NN	0.52	0.50	0.35	0.62	0.61	<b>0.63</b>	0.61
fe_4elt2	0.47	0.52	0.25	0.56	<b>0.60</b>	0.59	0.59
bcsstk36	0.45	0.41	0.30	0.49	-	<b>0.51</b>	0.44
k49_norm_10NN	0.048	0.045	0.034	0.12	-	<b>0.18</b>	0.093
fe_bcsstk32	0.32	0.38	0.21	<b>0.41</b>	-	0.24	0.37
m_t1	0.30	0.34	0.21	<b>0.37</b>	-	0.35	0.32
ship_003	0.21	0.21	0.17	<b>0.24</b>	-	0.20	<b>0.24</b>
fe_ocean	0.11	<b>0.12</b>	0.09	<b>0.12</b>	-	-	0.09
ok2010	<b>0.48</b>	0.46	0.27	0.32	-	-	0.44
web-NotreDame	0.38	0.31	0.33	<b>0.46</b>	-	-	0.39
coPapersCiteseer	-	0.079	0.058	<b>0.17</b>	-	-	0.091
gsm_106857	-	0.17	0.12	<b>0.24</b>	-	-	0.21
tx2010	<b>0.42</b>	0.39	0.26	0.21	-	-	0.36
com-Youtube	0.015	-	<b>0.092</b>	0.055	-	-	0.069
Flan_1565	-	-	0.09	0.20	-	-	<b>0.21</b>

TABLE 4.17: Neighborhood preservation per graph per method.

neighborhood preservation than PMDS for all graphs except com-Youtube. com-Youtube also had problems with creating  $G'$ , which possibly could be related in some way. Overall however, NNP-NET-c performs similar to DRGraph and significantly better than PMDS.

#### 4.10.1 Execution time comparison with DRGraph

Table 4.16 shows that the execution time of NNP-NET-c is lower than DRGraph for the two largest graphs. It is however hard to argue from those results that NNP-NET-c is expected to be faster than DRGraph for very large graphs, as the execution time of NNP-NET-c seems to be fairly random. These two results could be because of random chance. In this section we show a comparison with as much noise removed from our execution time results as possible. This is done in order to create a more meaningful execution time comparison between NNP-NET-c and DRGraph, showing our expected runtime for a certain size graph. There are three main sources of noise in our execution time results than can easily be removed:

- **Approximate tsNET\*:** The running time for creating the ground truth using approximate tsNET\* varies significantly between graphs of the same size. It took approximate tsNET\* somewhere between 20 to 90 seconds to create a layout for our subgraphs with size 10000. For this time comparison, we set the time needed for creating the ground truth to 50 seconds for all graphs that have at least 10000 nodes. 50 seconds was used, as it was exactly the average time needed to create the GT for graphs with at least 10000 nodes.
- **Weighted graphs:** Dijkstra has to be used when creating a layout for a weighted graph, were BFS can be used when it is not weighted. BFS is significantly faster then dijkstra. This results in significantly higher embedding creation times when using weights. To remove this noise, the weighted graphs were run again, but with the weights removed. The embedding creation times of these runs was then used. DR-Graph does not use the weights in the graphs, making this still a valid comparison.

- **Subgraph pivot fallback:** The coarsening subgraph method was not able to successfully reduce all graphs down to the required size. In these cases, the pivot subgraph method was used as a fallback. The pivot fallback is significantly slower however, increasing the running time on those graphs. The subgraph time was set to 1 second for all graphs that had this happen, as all other graphs were able to create the subgraph in less than a second time.

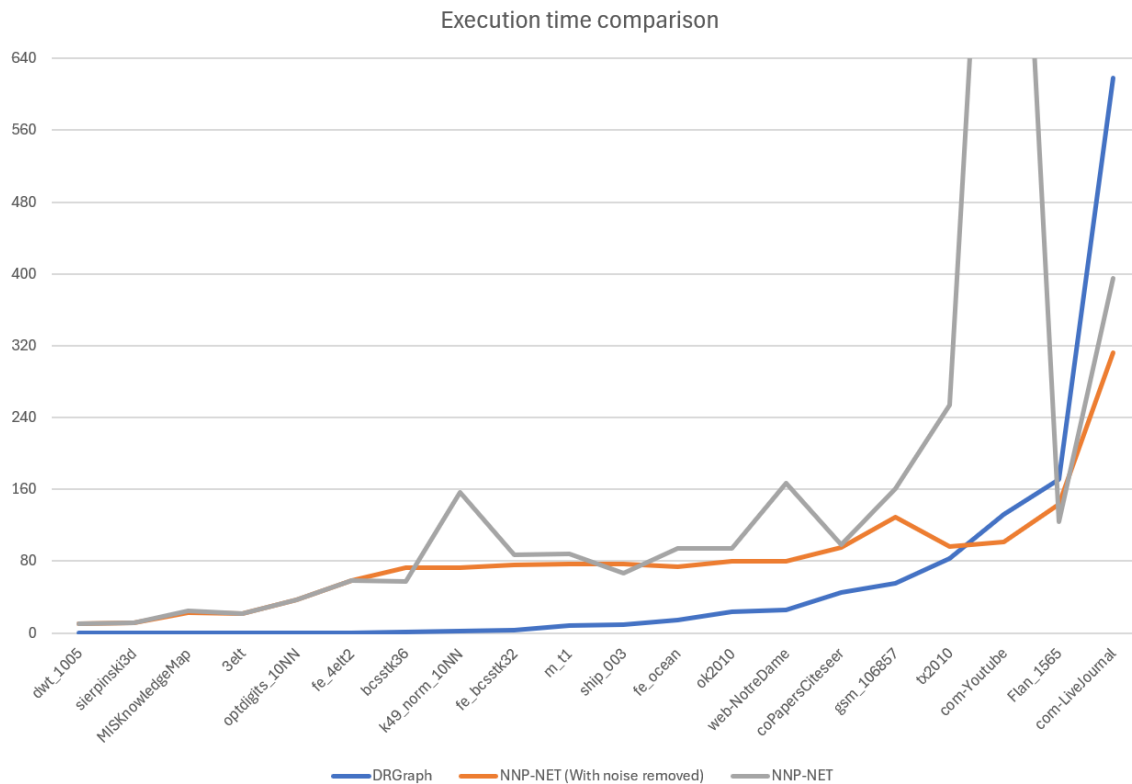


FIGURE 4.4: Chart comparing the execution time of DRGraph with NNP-NET-c, both the actual obtained results as well as the noise reduced results

The graphs are order from lowest node count to highest node count. From this, the expectation is that the execution time of each graph is higher than the time needed for the graphs smaller than itself. From figure 4.4 it becomes apparent however that this is not the case for the NNP-NET-c results. Removing the noise does create a more consistent line however. Only gsm\_106857 has a result that is significantly higher than expected, which resulted from a longer embedding creation time. This graph shows quite clearly that the NNP-NET-c execution time increases quite rapidly for graphs under the subgraph size of 10000. After this point however, the execution time stays almost constant up until coPapersCiteseer with about half a million nodes. This slow growth causes DRGraph to be slower from com-Youtube onwards when comparing with our noise reduced results.



## Chapter 5

# Discussion

A few criteria for our method were laid out at the start of this thesis. Lets take a closer look at each of these criteria and whether NNP-NET-c fulfills them.

- **Scalability:** Our method needs to have a linear time complexity and be able to layout very large graphs in a reasonable amount of time. NNP-NET-c has succeeded on this front, and performed even better than we aimed for. In the introduction, we established that our main focus was to get the layout time to linear, while the training time was allowed to be slower. The final pipeline is linear in both of these aspects however, making it possible to generate layouts from start to finish in linear time. NNP-NET-c does take significantly more time than other linear graph layout methods for smaller graphs, but becomes more competitive for larger graphs, eventually being faster than DRGraph on very large graphs. The actual time needed to create a layout is heavily dependent on the graph however, where some graphs could take significantly longer than other graphs with a similar size.
- **Layout quality:** The layouts generated by our method should be of similar quality to other methods and to the original tsNET\* results. On smaller graphs, where we use the entire graph as ground truth, the results are extremely close to the approximate tsNET\* results, which we use as the ground truth. The layouts generated for larger graphs using our method visually look good when comparing with other methods.
- **Robustness:** Our method should be able to generate a layout for any given graph. In the testing performed, it only failed on one graph, asia\_osm, which was because of to high memory requirements. A result should be able to be obtained using a system with more memory. NNP-NET-c did have more trouble with certain graphs than others. Most notably is com-Youtube, that took significantly longer than other graphs as the coarsening subgraph method was not able to create a  $G'$  that was small enough. It was still able to create a layout however, as it should be able to create a layout for any given graph.
- **Ease of use:** While our method has a lot of parameters that could be changed (perplexity, target  $G'$  size, training epochs, embedding size  $n$ , approximate tsNET\*'s  $\theta$  and PMDS pivot count), all of these were set to the same constant set value for all tests. These values gave good results for all tests performed. While these values can be tinkered with to possibly obtain better results, it is not needed in order to get good results from this method.
- **Edge weights:** Edge weights are used in NNP-NET-c's pipeline. The results show that the edge weights have a noticeable impact on the end results, meaning that they are being used to generate a more correct representation of the input data.

Our method largely fulfilled the requirements that were laid out at the start of this thesis. It is also gets competitive results with other methods, making it a viable alternative to those methods.

The core of NNP-NET is to use NNP to generate a layout for a given graph. In order to accomplish this, multiple different components are needed to get NNP working in a graph context. An embedding needs to be created for each point in the graph. Furthermore, a ground truth needs to be generated for NNP to learn from. Learning from the entire ground truth is pointless however, as in that case we already have a good layout, so a way is needed to choose specific points and generate a ground truth from that. We will take a closer look at each of these aspects and reflect on the choices made and whether alternatives could be used.

## 5.1 Embedding method

Two different embedding methods were considered in this paper: Using pivot points as the embedding, or using a high dimensional PMDS layout. The results showed that the PMDS embedding was able to get results closer to the original ground truth layout. The PMDS embedding did take significantly more time however. NNP-NET-c's execution time is largely dominated by the creation of the PMDS embedding when creating a layout for larger graphs, as evident by the results from section 4.9. Using the pivot embedding instead of the PMDS embedding might be a worth while tradeoff. While it would reduce the quality of the layout somewhat, it would reduce the execution time significantly. How significantly is hard to estimate from the tests performed, as the pivot embedding has only been tested on smaller graphs, which do not necessarily give a good indication of how well the embedding would perform on larger graphs. This might be an interesting approach to revisit however, as it could allow NNP-NET-c to have a lower execution time compared to other methods earlier.

More experimentation can also be done with the PMDS embedding, or more precisely, with the embedding using a high-dimensional graph layout. The idea of this embedding as explained in 3.1.2, any graph layout that fulfills the needed criteria could be used to generate the embedding. In our testing, only PMDS was tested for this purpose. There might however be a different graph layout algorithm that better fits our requirements and would perform better, either in layout quality or execution time, compared to using PMDS for the embedding.

There always is the possibility to use a completely different method for creating the embedding. The embedding method can freely be swapped out for any method as long as the new embedding method fits all the necessary requirements. Swapping out the embedding method can be done without having requiring any changes to the rest of the pipeline, as all the components are completely separate from each other.

## 5.2 Ground Truth

One of the goals of this thesis was to accelerate tsNET(\*) in order to use it on large large graphs. This made it a logical choice to use tsNET\* to generate the ground truth that NNP learns from. As shown in section 4.8 however, other graph layout algorithms can be used to generate the ground truth, which also produces good results. In order to effectively use a graph layout algorithm to generate the ground truth used by NNP-NET it needs to be able to generate a layout of  $G'$  in a reasonable amount of time. This is why the choice was made to use the approximate tsNET\* implementation instead of an exact tsNET\* implementation. Generating a graph layout using approximate tsNET\* for a  $G'$  of size 10000

still takes somewhere between 20 to 90 seconds depending on the graph, as shown in section 4.9. This time cost causes our method to be significantly slower than other method for smaller graphs. Once the graphs get bigger, this problem is alleviated as this time cost does not increase with the input size. A different faster method could be chosen in order to minimize this time cost.

On the other hand, using the exact version of tsNET\* might improve the quality of the results. This would however likely require that less than 10000 points are chosen to generate  $G'$ , as the execution time would increase significantly by using exact tsNET\*. Reducing the number of points in  $G'$  would however reduce the quality of the results, as shown in section 4.4, so it would have to be tested whether this would have an overall positive or negative impact on the quality of the results.

### 5.3 Subgraph method

There were two subgraph methods proposed and tested in order to create the subgraph  $G'$  from the full graph  $G$ : Using pivot points to choose the points, and coarsening the entire graph to get a smaller representation of the original  $G$ . The coarsening method was chosen as it had a significantly lower execution time and was therefore able to generate larger subgraphs than the pivot method. Using this method does come with its downsides however. Coarsening the graph can distort the between points in a graph. This is because edges get deleted in the process of coarsening the graph, which reduces the number of possible paths that can be taken, including paths that are the shortest distance between two points. The impact that this loss of information has on the end result has not been directly measured in this thesis, nor how large the error in the distances between points is because of the coarsening process. Using the pivot method for this purpose would not have this issue, although the execution time would go up significantly, which is not desirable.

The other problem with the coarsening method is that it is not always able to reach the target number of nodes for  $G'$ . While falling back to the pivot method allows NNP-NET-c to still produce results, even for the graphs that suffer from this issue, it does come with a significant time cost. Ideally, the coarsening would be changed in a way that ensures that it will always be able to reach the target number of points. This has already been tried by changing the order the nodes are collapsed in to go from the node with the lowest degree to the node with the highest degree. While this helped, it did not get rid of the issue completely. More research would have to be put into this method in order to make sure it can reduce all graphs to the required size within a reasonable amount of time.

## Chapter 6

# Conclusion

This thesis has presented a new graph layout method, NNP-NET-c, that can create layouts for graphs in the style of tsNET\*, but for far larger graphs than tsNET\* can do. While our method is not competitive on small graphs (as it gets the same results as tsNET\*, while taking longer), it gets good results on larger graphs that tsNET\* can not handle. It is able to generate graphs in the style of tsNET\* for graphs that are far larger than what tsNET\* can handle. This thesis has also shown that tsNET\* and PMDS can be extended to use edge weights by simply incorporating them in the shortest path algorithm as edge length. NNP-NET-c also takes the edge lengths into account because they are used in the ground truth used for our algorithm.

Keeping all this mind, we can reflect back on the original goal of this thesis. The goal of this thesis was to create a faster version of tsNET that would be able to create graph layouts for very large graphs while also taking edge weights into account. The focus would be on making sure that the inference time would be very low, while not focussing on getting the training time to a reasonable level. In the end however, both the inference and training time were low enough to outperform DRGraph on very large graphs. We've also shown that NNP-NET-c is able to successfully take edge weights into account in a meaningful way. We were able to exceed our original requirements for our method.

NNP-NET-c is created from multiple different components that each can be swapped out for a different algorithm that fulfills the same requirements, without having to change anything about the rest of the pipeline. Future work can build on this method by changing these components individually in order to either get better quality results, or to reduce the execution time. The main improvements would include a more robust subgraph method for creating  $G'$ . While the current method works well on most graphs, it struggles with some others. Changing out the graph layout method that is used to create the embedding could also have a large positive impact. We were unable to run NNP-NET-c on the largest graph in our set, as the memory requirements of PMDS were too high for the limit amount of RAM in the test system. PMDS also ended up as the bottleneck for the larger graphs. A faster method than PMDS would increase performance significantly for very large graphs. Furthermore, these results were all generated using tsNET\* as the ground truth. NNP-NET-c can use any method as a ground truth method. It would be interesting to see whether a different ground truth method could yield better results in this pipeline. Lastly, NNP-NET-c does contain a good number of parameters. The results have shown that they can all be set to a default value while generating competitive results compared to other state of the art methods. For a lot of these parameters however, no in depth testing was done to determine what a good default value should be. Tweaking some of these values has the potential to improve the results even more.

## Chapter 7

# Acknowledgements

Aside from my supervisors, Alex Telea and Tamara Mtsentlintze, I also want to thank Simon van Wageningen, who also present at meetings in order to brainstorm ideas for our method, and who also provided very useful feedback on both intermediate results as well as the final thesis. He has been a large help for this thesis. There are two more people I want to mention here. While both have not contributed to the thesis, I do want to thank Romy Jeskens and Thomas Vos for their continued support.

# Bibliography

- Ahmed, Reyan, Felice De Luca, Sabin Devkota, Stephen Kobourov, and Mingwei Li (2020). "Graph Drawing via Gradient Descent, (GD)<sup>2</sup>". In: *Graph Drawing and Network Visualization: 28th International Symposium, GD 2020, Vancouver, BC, Canada, September 16–18, 2020, Revised Selected Papers 28*, pp. 3–17.
- (2022). "Multicriteria Scalable Graph Drawing via Stochastic Gradient Descent, (SGD)<sup>2</sup>". In: *IEEE Transactions on Visualization and Computer Graphics* 28.6, pp. 2388–2399.
- Ayesha, Shaeela, Muhammad Kashif Hanif, and Ramzan Talib (2020). "Overview and comparative study of dimensionality reduction techniques for high dimensional data". In: *Information Fusion* 59, pp. 44–58.
- Brandes, Ulrik and Christian Pich (2007). "Eigensolver Methods for Progressive Multidimensional Scaling of Large Data". In: *Graph Drawing*. Ed. by Michael Kaufmann and Dorothea Wagner. Berlin, Heidelberg, pp. 42–53. ISBN: 978-3-540-70904-6.
- Chimani, Markus, Carsten Gutwenger, Michael Jünger, Gunnar W Klau, Karsten Klein, and Petra Mutzel (2013). "The Open Graph Drawing Framework (OGDF)". In: *Handbook of graph drawing and visualization* 2011, pp. 543–569.
- Davis, Timothy A and Yifan Hu (2011). "The University of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1, pp. 1–25.
- Espadoto, Mateus, Nina Sumiko Tomita Hirata, and Alexandru C Telea (2020). "Deep learning multidimensional projections". In: *Information Visualization* 19.3, pp. 247–269.
- Gansner, Emden R., Yifan Hu, and Stephen North (2013). "A Maxent-Stress Model for Graph Layout". In: *IEEE Transactions on Visualization and Computer Graphics* 19.6, pp. 927–940.
- Gibson, Helen, Joe Faith, and Paul Vickers (2013). "A survey of two-dimensional graph layout techniques for information visualisation". In: *Information visualization* 12.3-4, pp. 324–357.
- Giovannangeli, Loann, Frederic Lalanne, David Auber, Romain Giot, and Romain Bourqui (2021). "Deep Neural Network for DrawiNg Networks, (DNN)<sup>2</sup>". In: *Graph Drawing and Network Visualization: 29th International Symposium, GD 2021, Tübingen, Germany, September 14–17, 2021, Revised Selected Papers 29*, pp. 375–390.
- Gori, M., G. Monfardini, and F. Scarselli (2005). "A new model for learning in graph domains". In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005*. Vol. 2, 729–734 vol. 2.
- Hachul, Stefan (2005). "A Potential-Field-Based Multilevel Algorithm for Drawing Large Graphs". PhD thesis. Universität zu Köln.
- Harel, David and Yehuda Koren (2002). "Graph Drawing by High-Dimensional Embedding". In: *Graph Drawing*. Ed. by Michael T. Goodrich and Stephen G. Kobourov. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 207–219. ISBN: 978-3-540-36151-0.
- Hu, Yifan (2005). "Efficient, high-quality force-directed graph drawing". In: *Mathematica journal* 10.1, pp. 37–71.
- Joia, Paulo, Danilo Coimbra, Jose A Cuminato, Fernando V Paulovich, and Luis G Nonato (2011). "Local affine multidimensional projection". In: *IEEE Transactions on Visualization and Computer Graphics* 17.12, pp. 2563–2571.

- Kruiger, Johannes F, Paulo E Rauber, Rafael Messias Martins, Andreas Kerren, Stephen Kobourov, and Alexandru C Telea (2017). "Graph Layouts by t-SNE". In: *Computer graphics forum*. Vol. 36. 3, pp. 283–294.
- Leow, Yao Yang, Thomas Laurent, and Xavier Bresson (2019). "GraphTSNE: A Visualization Technique for Graph-Structured Data". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*.
- Maaten, Laurens Van der and Geoffrey Hinton (2008). "Visualizing data using t-SNE." In: *Journal of machine learning research* 9.11.
- Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean (2013). "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems*. Ed. by C.J. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger. Vol. 26.
- Paulovich, Fernando V., Luis G. Nonato, Rosane Minghim, and Haim Levkowitz (2008). "Least Square Projection: A Fast High-Precision Multidimensional Projection Technique and Its Application to Document Mapping". In: *IEEE Transactions on Visualization and Computer Graphics* 14.3, pp. 564–575.
- Purchase, HELEN C. (2002). "Metrics for Graph Drawing Aesthetics". In: *Journal of Visual Languages Computing* 13.5, pp. 501–516. ISSN: 1045-926X.
- Silva, Vin and Joshua Tenenbaum (2002). "Global Versus Local Methods in Nonlinear Dimensionality Reduction". In: *Advances in Neural Information Processing Systems*. Ed. by S. Becker, S. Thrun, and K. Obermayer. Vol. 15.
- Tamassia, Roberto (2013). *Handbook of graph drawing and visualization*. CRC press.
- Torgerson, Warren S (1952). "Multidimensional scaling: I. Theory and method". In: *Psychometrika* 17.4, pp. 401–419.
- Van Der Maaten, Laurens (2014). "Accelerating t-SNE using tree-based algorithms". In: *The journal of machine learning research* 15.1, pp. 3221–3245.
- Venna, Jarkko and Samuel Kaski (2001). "Neighborhood preservation in nonlinear projection methods: An experimental study". In: *International Conference on Artificial Neural Networks*, pp. 485–491.
- Wang, Xiaoqi, Kevin Yen, Yifan Hu, and Han-Wei Shen (2021). "DeepGD: A Deep Learning Framework for Graph Drawing Using GNN". In: *IEEE Computer Graphics and Applications* 41.5, pp. 32–44.
- Wang, Yong, Zhihua Jin, Qianwen Wang, Weiwei Cui, Tengfei Ma, and Huamin Qu (2020). "DeepDrawing: A Deep Learning Approach to Graph Drawing". In: *IEEE Transactions on Visualization and Computer Graphics* 26.1, pp. 676–686.
- Wu, Zonghan, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu (2021). "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1, pp. 4–24.
- Zheng, Jonathan X., Samraat Pawar, and Dan F. M. Goodman (2019). "Graph Drawing by Stochastic Gradient Descent". In: *IEEE Transactions on Visualization and Computer Graphics* 25.9, pp. 2738–2748.
- Zhou, Jie, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun (2020). "Graph neural networks: A review of methods and applications". In: *AI Open* 1, pp. 57–81. ISSN: 2666-6510.
- Zhu, Minfeng, Wei Chen, Yuanzhe Hu, Yuxuan Hou, Liangjun Liu, and Kaiyuan Zhang (2020). "DRGraph: An efficient graph layout algorithm for large-scale graphs by dimensionality reduction". In: *IEEE Transactions on Visualization and Computer Graphics* 27.2, pp. 1666–1676.