Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Distance Preserving Graph Embedding

Bachelor Thesis

Dustin Brunner

brunnedu@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Karolis Martinkus
Prof. Dr. Roger Wattenhofer

August 31, 2021

Acknowledgements

First and foremost, I would like to thank Karolis Martinkus for supervising my thesis and always providing helpful tips. I would also like to thank Professor Wattenhofer and the Distributed Computing Group for providing the opportunity to do my thesis on such an interesting topic. I would also like to thank all the creators and contributors to the various datasets and packages that I was able to use. And last but not least, I'd like to thank my family and friends for their support during the six months of working on my thesis.

Abstract

Today graph data structures are everywhere, be it social graphs describing the relation between different accounts on social media platforms, road graphs describing the connections between different intersections in a city, or network graphs describing the connections between different servers. Being able to quickly query the shortest path distance between two nodes in such graphs can be of great use. However, using traditional graph traversing shortest path algorithms such as Dijkstra [1] or Floyd-Warshall [2] is slow on big graphs, and alternatively storing all shortest path information for lookup takes too much space.

Therefore, there have been various attempts at finding ways to speed up shortest path queries. Some of them include introducing shortcuts in the graph that preserve distance or computing labels for each node that encode the distance to selected landmark nodes. This paper explores other approaches that have not been explored in-depth yet. On the one hand, we investigate the distance preserving capabilities of established node embedding techniques using a simple multilayer perceptron. On the other hand, we propose our own graph neural network models for generating distance preserving embeddings and predicting shortest path distances. Furthermore, we investigate the relationships between certain parameters such as embedding size and prediction accuracy.

Contents

Acknowledgements i

Abstract ii

1 Introduction 1

2 Background 2

2.1 Task . . . 2

2.2 Exact Methods . . . 3

2.2.1 Traditional Methods . . . 3

2.2.2 Labeling Methods . . . 4

2.3 Approximate Methods . . . 5

2.3.1 Landmark Methods . . . 5

2.3.2 Embedding Methods . . . 6

3 Data 10

3.1 Graphs . . . 10

3.2 Preprocessing . . . 11

4 Models 13

4.1 Existing Node Embedding Techniques . . . 13

4.1.1 node2vec . . . 14

4.1.2 GraRep . . . 15

4.1.3 ProNE . . . 16

4.1.4 Comparison . . . 17

4.1.5 Evaluation Model . . . 18

4.2 Graph Neural Networks . . . 21

4.2.1 Message Passing . . . 21

4.2.2 Base Architecture . . . 21

4.2.3 CoordNet . . . 22
4.2.4 StackedCoordNet . . . 22
4.2.5 StackedEuclidCoordNet . . . 23
4.2.6 StackedAngleCoordNet . . . 23
4.3 Hyperbolic Graph Convolutional Network . . . 23
4.3.1 DPModel . . . 24

5 Experiments **26**
5.1 Training Settings . . . 26
5.2 Results . . . 27
5.2.1 Existing Node Embedding Techniques . . . 27
5.2.2 Graph Neural Networks . . . 28
5.2.3 Hyperbolic Graph Convolutional Network . . . 29
5.2.4 Landmark Training . . . 30
5.2.5 Impact of Embedding Dimensionality on Accuracy . . . 31
5.2.6 Error Distribution over Path Lengths . . . 32

6 Conclusion & Future Work **34**
6.1 Conclusion . . . 34
6.2 Future Work . . . 34

Bibliography **35**

Chapter 1

Introduction

Recommending a user the closest gas station to his location on a navigation app or making suggestions for new friends to add on a social platform are problems that need to be solved millions of times every second by Big Tech companies. These problems can be abstracted to shortest path distance queries on a graph. Therefore, in order to answer such a large number of queries as quickly as possible, it is important that we have very efficient and accurate algorithms to predict shortest paths.

For this reason, efforts were already made in the late 2000s to speed up shortest path queries on graphs. Some research groups worked on exact methods [3, 4], others on approximate methods [5, 6, 7, 8] and some used a rather rigorous mathematical approach [9, 10], while others used a rather algorithmic approach [11, 12, 13]. With the rapid increase in computing power and efficiency of computers in the early 2010s and the subsequent advent of machine learning, new approaches for solving the approximate version of the problem became possible. Nevertheless, to date, no more than a handful of papers using such a machine learning approach for answering approximate shortest path distance queries have been published. So there is still a lot of potential for improving such algorithms.

Therefore, this work focuses on investigating distance preserving embedding techniques that have not been studied before and allow for fast query speeds by processing the embeddings of the start and end nodes of a path. Further, we will be evaluating them on different datasets to learn more about the intricacies and challenges in solving this problem. In the second chapter, we first define the task and then analyze and explain the various traditional and newer techniques that have been proposed by others so far. Then we look at each investigated model in detail going over its architecture, hyperparameters and input format. After that we describe our datasets, the data processing, training and evaluation methods used and write about the results of our experiments. We finish with a conclusion of our findings.

Chapter 2

Background

In this chapter we first define some important terminology used throughout the thesis and describe the task we are trying to solve. In the last two sections, we will look at the various traditional and more recent shortest path distance computation methods related to our work.

2.1 Task

A *graph* $G = (V, E)$ is a non-linear data structure consisting of *vertices* V and *edges* E. Vertices are sometimes also referred to as *nodes*. Each edge $e = (v_1, v_2)$ can be described as the tuple of vertices v_1, v_2 it connects. Edges can either be *directed* or *undirected* and *weighted* or *unweighted*. Depending on the use case, weights can be used to express the cost of passing the edge, the length of the edge or the strength of the connection between its incident vertices.

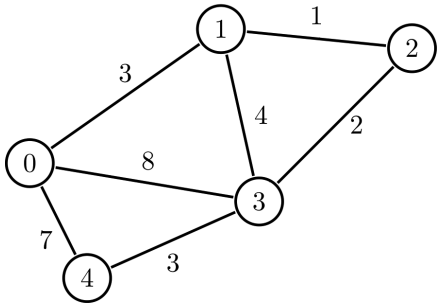


Figure 2.1: Example of an undirected, weighted graph.

Two vertices v_1, v_2 are *adjacent* if there exists an edge $e = (v_1, v_2)$. A *path* is a sequence of pairwise distinct adjacent vertices that begins at an initial vertex s and ends at a terminal vertex t. The *unweighted path length* is the number of edges on the path and the *weighted path length* is the sum of the weights of the edges on the path. Therefore the *shortest weighted path* from vertex 4 to vertex 1 in the example graph 2.1 would be the one traversing vertices $4-3-2-1$ and

its length would be 6. The goal of this thesis is to speed up the query times for approximating the lengths of such shortest weighted paths on road graphs while maintaining accuracy. For simplicity, we focus on undirected weighted graphs, while most of the methods discussed can also be applied to directed graphs if slightly modified.

2.2 Exact Methods

While exact methods are able to make error-free distance predictions this comes with the downside of them having a larger space or time complexity than approximate methods. Therefore they are mostly used for small graphs but are less useful for larger graphs.

2.2.1 Traditional Methods

Two of the oldest methods for computing shortest paths on graphs are *Dijkstra's algorithm* [1] and the *Floyd–Warshall algorithm* [2]. Although they were invented at the beginning of the information age, before the personal computer even existed, they are still widely used today.

Dijkstra

Dijkstra's algorithm is a single-source shortest path algorithm, meaning it computes for a single-source node the shortest paths to all other nodes. It works both on directed and undirected graphs but only with positive edge weights. Its time complexity depends on the data structure used for storing and querying partial solutions, but can be as low as $\mathcal{O}(m + n \log n)$, where m is the number of edges and n is the number of vertices in the graph. Its space complexity is $\mathcal{O}(n)$.

Floyd–Warshall

The Floyd–Warshall algorithm is an all-pairs shortest path algorithm, meaning it computes for all pairs of nodes the shortest path between them. It can handle both directed and undirected graphs and even negative edge weights as long as there are no negative cycles. Its time complexity is always $\mathcal{O}(n^3)$ and its space complexity is $\mathcal{O}(n^2)$.

2.2.2 Labeling Methods

To speed up query times over traditional methods many methods make use of distance labels. During preprocessing every node v_i gets a distance label $V_i = \{(v_j, dist(v_i, v_j) \mid v_j \in V_{labels}\}$ that is a set of tuples containing the distances from v_i to some other nodes V_{labels} that may differ for each node. The distance from v_i to v_j can then easily be calculated by inspecting their distance labels:

$$dist(v_i, v_j) = min\{dist(v_i, v) + dist(v_j, v) \mid v \in V_i \cap V_j\} \tag{2.1}$$

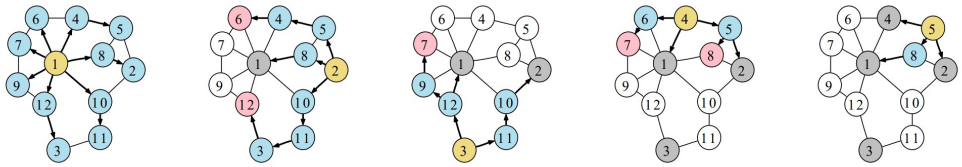
The difficulty with all labeling methods is to find the minimal set of nodes to which the distances need to be calculated and stored so that all shortest paths can be exactly computed. Finding that optimal set of nodes for a graph has been proven to be an NP-hard problem [14].

Multi-Hop Labeling

Multi-Hop Labeling [3] builds on the foundation of *2-Hop Labeling* [14], which computes for every node the distances to a set of hub nodes and stores them in the nodes distance label to ensure that the above calculation of $dist(v_i, v_j)$ indeed yields the correct distance. Instead of generating all 2-hop distance labels during preprocessing Multi-Hop Labeling generates only a small subset of 2-hop distance labels and generates the sufficient 2-hop labels needed for answering a query on-line. This can be achieved using a hierarchical approach where each node except the root node gets assigned a parent node and if two nodes do not share a common hub node in their distance labels the distance labels of their parent nodes get checked recursively until a common hub node is found. Therefore Multi-Hop Labeling is able to significantly reduce the preprocessing time and size of the stored distance labels.

Pruned Landmark Labeling

Pruned Landmark Labeling [4] is a method created by Akiba et al. specifically for large-scale graphs. It makes use of pruned breadth-first searches and significantly speeds up the preprocessing by exploiting bitwise operations to perform up to 64 breadth-first searches in parallel. To prune the breadth-first searches the nodes are first ordered either randomly or based on some metric such as degree or closeness centrality to determine the sequence of root nodes of the searches. Then a breadth-first search is performed starting from the first root node and every node appends its distance from the first node to its distance label. In the following breadth-first searches starting from node s whenever a new node u is reached the current breadth-first search distance $d_{BFS}(s, u)$ is compared to the computed distance based on all labels created in previous searches $dist(s, u)$ and if $d_{BFS}(s, u) \geq dist(s, u)$ the search gets pruned as shown in Figure 2.2. By



(a) First BFS from vertex 1. We visited all the vertices.

(b) Second BFS from vertex 2. We did not add labels to five vertices.

(c) Third BFS from vertex 3. We only visited the lower half of the vertices.

(d) Fourth BFS from vertex 4. This time we only visited the higher half.

(e) Fifth BFS from vertex 5. The search space was even smaller.

Figure 2.2: Examples of pruned BFSs. Yellow vertices denote the roots, blue vertices denote those which we visited and labeled, red vertices denote those which we visited but pruned, and gray vertices denote those which are already used as roots. [4]

using pruned breadth-first searches this method significantly reduces label sizes and is, therefore, able to handle graphs up to two orders of magnitude larger than previous exact methods.

2.3 Approximate Methods

Compared to the various proposed improvements of exact methods approximate shortest path distance methods try to further reduce the space and query cost by sacrificing some prediction accuracy. Depending on the application, this sacrifice may be worthwhile if exact distances are not absolutely necessary or if we are dealing with very large graphs.

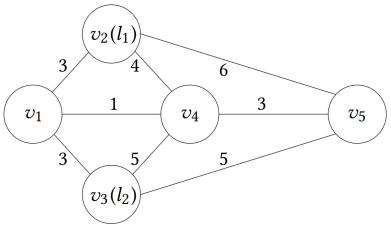
2.3.1 Landmark Methods

Most approximate shortest path distance methods use a landmark-based approach. Similar to the previously discussed labeling methods, landmark-based approaches work by selecting a subset of all nodes as landmark nodes $L \subset V$ with $|L| \ll |V|$. Each node then gets assigned a label consisting of its distances to these landmark nodes. When querying the distance of v_i to v_j the following approximation is returned:

$$dist(v_i, v_j) = min\{dist(v_i, l) + dist(v_j, l) \mid l \in L\} \tag{2.2}$$

In Figure 2.3 you can see an example landmark labeling of an undirected, weighted graph containing five nodes and two of which v_2, v_3 are landmarks.

Efforts at improving landmark-based methods focus mostly on improving the landmark selection as the prediction accuracy largely depends on it. But as selecting optimal landmarks is NP-hard [15] heuristic solutions need to be employed to select good landmarks. Potamias et al. showed that compared to randomly



(a) Road graph example

	$v_2(l_1)$	$v_3(l_2)$
v_1	3	3
v_2	0	6
v_3	6	0
v_4	4	4
v_5	6	5

(b) Landmark labeling

Figure 2.3: Landmark labeling example [7]

selecting landmarks using landmark selection heuristics, such as low closeness centrality, high betweenness centrality or high degree, can reduce the space cost by a factor of up to 250 while maintaining the same accuracy [15].

Definition 2.1 (Degree)**.** The *degree* of a vertex v in an unweighted, undirected graph $G = (V, E)$ is defined as the number of adjacent vertices v_i s.t. $(v, v_i) \in E$.

Definition 2.2 (Closeness Centrality)**.** The *closeness centrality* of a vertex v in a graph $G = (V, E)$ is defined as the average distance $\frac{1}{n} \sum_{v_i \in V} dist(v, v_i)$ of v to other vertices in the graph.

Definition 2.3 (Betweenness Centrality)**.** The *betweenness centrality* of a vertex v in a graph $G = (V, E)$ is defined as the proportion of shortest paths $\sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}$ between other vertices $s, t \in V$ lies on, where σ_{st} is the number of shortest paths between s and t and $\sigma_{st}(v)$ is the number of these paths that v lies on.

2.3.2 Embedding Methods

Another line of work in the category of approximate shortest path distance methods are embedding methods that embed every node in a d-dimensional latent space such as Euclidean space or hyperbolic space during preprocessing. While they also compute the one-to-all shortest path distances from a subset of nodes to all other nodes during the preprocessing, what separates them from the previously looked at landmark methods is the evaluation method. Instead of approximating the distance between two nodes v_i, v_j as described in equation 2.2, embedding methods use more efficient evaluation functions. They either directly calculate the L_p norm between the embeddings or they use a simple neural network to predict the distances based on the embeddings. Therefore they are able to increase query speeds even more than other approximate methods.

Orion

In the first of two papers focused on efficient node distance computation in unweighted social graphs Zhao et al. proposed a *graph coordinate system* named *Orion* [5]. A graph coordinate system maps nodes in high dimensional graphs to points in a d-dimensional Euclidean coordinate space. At query time, the associated coordinates to node v_i can then be used together with a Euclidean distance computation 2.3 to approximate, in constant time, the distance to every other node $v_j \in V$ in the graph.

$$dist(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} \tag{2.3}$$

To map the nodes to Euclidean coordinates during preprocessing Orion selects k landmark nodes from which the distances to all other nodes are computed using k breadth-first searches. Then a small number of landmarks are selected as initial landmarks whose pairwise distances are used in the first step to calibrate their positions using the Simplex Downhill algorithm [16]. The secondary landmarks then calibrate their positions using the initial landmarks as anchors to limit the cost of using the Simplex Downhill algorithm for the calibration of all landmark nodes as it runs in $\mathcal{O}(k^2 \cdot d)$ time. Finally, all other nodes are calibrated with their relative distance to the previously positioned landmarks.

Rigel

While Orion was successful at increasing query speeds significantly, the choice of using a Euclidean embedding space meant that social graphs, which are usually highly connected, could not be embedded without some distortion error. Because of this Zhao et al. proposed another graph coordinate system named *Rigel* [6] less than two years after they proposed Orion. Instead of using Euclidean space as embedding space, Rigel uses a d-dimensional hyperbolic coordinate space which allows for relatively low distortion error. Another improvement of Rigel over Orion is that the embedding process can be run on multiple machines in parallel which allows Rigel to be much more scalable than Orion. Rigel uses the Hyperboloid model of hyperbolic space as it has the advantage of a relatively simple point-to-point distance function 2.4 and the complexity evaluating this function is not dependent on the space curvature $c \leq 0$ which allows for it to be another tunable parameter.

$$dist(x, y) = arccosh\left(\sqrt{\left(1 + \sum_{i=1}^{d} x_i^2\right)\left(1 + \sum_{i=1}^{d} y_i^2\right)} - \sum_{i=1}^{d} x_i y_i\right) \cdot |c| \tag{2.4}$$

To embed the nodes in the hyperbolic space, Rigel uses a technique similar to Orion. It first fixes the coordinates of a small number of landmark nodes using

a global optimization algorithm and then positions all the remaining nodes so that their distances from the landmarks in the coordinate space match their real distances as closely as possible. Overall Rigel was a big improvement over Orion and was also deployed at different social network and gaming companies.

Shortest Path Distance Approximation using DL Techniques

In the paper "Shortest Path Distance Approximation using Deep Learning Techniques" [8] researchers from the University of Passau propose to feed a simple neural network with node2vec [11] or Poincare [17] embeddings to predict the shortest path distance of two nodes in social graphs. More specifically, they first learn the embeddings for each node. To approximate the distance between two nodes v_i, v_j they then combine their embeddings $emb(v_i), emb(v_j)$ using one of four binary operations. The operations are component-wise subtraction, average, multiplication or concatenating the embeddings of both nodes. These embedding combinations are then fed into a feedforward neural network consisting of a single hidden layer that outputs a real-valued prediction of the shortest path distance. Since the model is designed to predict distances in unweighted social graphs, the prediction is rounded to the nearest integer before being output. Although the model is quite simple, it achieves a significantly lower mean absolute error 2.4 on the social graph dataset than Rigel and Orion. Moreover, node2vec embeddings lead to better results than Poincare embeddings in all experiments.

Definition 2.4 (Mean Absolute Error)**.**

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

Vdist2vec

Vdist2vec is the name of a model recently proposed by Qi et al. in the paper "A Learning Based Approach to Predict Shortest-Path Distances" [7] that aims to predict shortest path distances on road networks efficiently and accurately. Instead of restricting the evaluation function to be a distance function in Euclidean space or any model of hyperbolic space, Vdist2vec uses a simple multilayer perceptron (MLP) to predict the shortest path distance given the embeddings of two vertices. In the same way, Vdist2vec restricts the generation of embeddings as little as possible by using one-hot encodings of vertices as input and generating the embeddings of each vertex in the first hidden layer which consists of k nodes. This results in k-dimensional embeddings of each vertex.

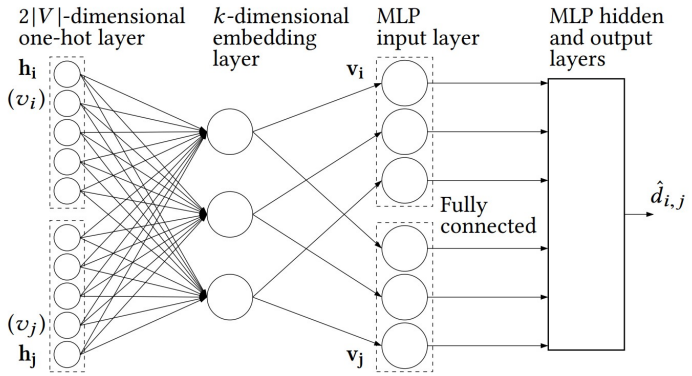


Figure 2.4: Base architecture of Vdist2vec [7]

In addition, Qi et al. propose two slightly improved variants of their base model named *Vdist2vec-L* and *Vdist2vec-S*. Compared to the base model that uses a classical mean square error 2.5 as loss function Vdist2vec-L uses a Huber loss 2.6 to shrink larger errors. Vdist2vec-S on the other hand splits the evaluation multilayer perceptron up into multiple multilayer perceptrons that each focus on a different distance range whose outputs are then summed to get the final distance prediction. As tested in experiments on multiple different graphs, Vdist2vec-S performed best out of the three proposed models and is currently the state-of-the-art model for shortest path distance prediction on road networks.

Definition 2.5 (Mean Squared Error)**.**

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Definition 2.6 (Huber Loss)**.**

$$L_\delta(\hat{y}, y) = \begin{cases} \frac{1}{2}(\hat{y} - y)^2 & \text{for } |\hat{y} - y| \leq \delta, \\ \delta\,(|\hat{y} - y| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

Chapter 3

Data

In this chapter we write about the datasets used for evaluating our models and how we preprocess them for training.

3.1 Graphs

We use four different graph datasets that vary both in structure as well as in scale and that have also been used to evaluate Vdist2vec [7]. Their number of vertices ($|V|$), number of edges ($|E|$), average node degree, and diameter (d_{max}) are summarized in Table 3.1.

The Winterthur dataset is an OpenStreetMap [18] dataset we acquired using the OSMnx [19] Python package. The Surat and Dongguan datasets are both from a collection of urban road network data from the Complex and Sustainable Urban Networks (CSUN) laboratory [20]. And lastly, the New York dataset is one of the datasets from the 9th DIMACS Implementation Challenge [21]. All datasets contain weighted edges and unique map coordinates for each vertex. Furthermore, they are all undirected.

Graph Dataset	$\|V\|$	$\|E\|$	avg. deg	d_{max}
Winterthur, Switzerland **(WTHUR)**	1.6 K	2.2 K	2.73	13 km
Surat, India **(SRT)**	2.5 K	3.6 K	2.86	51 km
Dongguan, China **(DNG)**	7.7 K	10.5 K	2.75	97 km
New York, USA **(NY)**	264 K	365 K	2.76	160 km

Table 3.1: Datasets

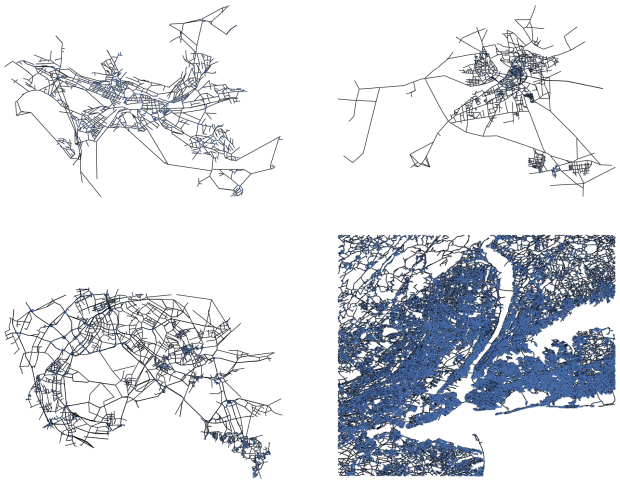


Figure 3.1: Graph visualizations (top, left to right: WTHUR, SRT bottom, left to right: DNG, NY)

3.2 Preprocessing

Clean Up

To clean up the raw datasets we process all graphs the same way using the NetworkX [22] Python package. This process consists of the following steps:

1. Read the graph from an edgelist or download it using OSMnx.
2. Assign provided coordinates to each vertex.
3. Convert graph to an undirected graph using NetworkX.
4. Extract largest connected component from graph.
5. Rename vertices to integers in the range $\{0, 1, ..., |V| - 1\}$

Shortest Path Data

To evaluate our models on these datasets we also calculate the ground truth of the shortest path distances during the preprocessing phase. Due to the large number of nodes in the New York graph and the resulting extremely large number of shortest paths, we are only able to calculate the all-pairs shortest path (APSP) data for Winterthur, Surat, Dongguan and Quanzhou. For the New York graph we calculate a fraction of the APSP data using 1000 randomly selected landmark nodes whose single-source shortest path data we combine. To calculate the single-source or all-pairs shortest path data we use the single-source Dijkstra and the

all-pairs Floyd-Warshall implementation of NetworkX, respectively. In Figures 3.3 and 3.2 you can see how both the weighted and unweighted shortest path lengths are distributed in the datasets.

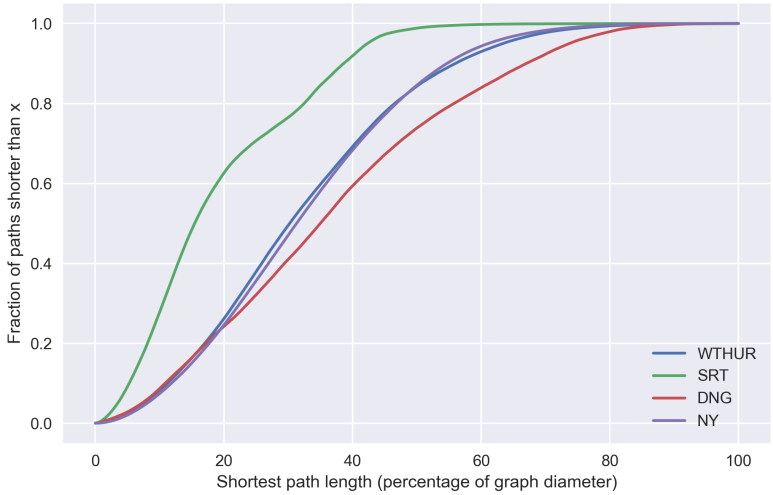


Figure 3.2: CDF plot of **weighted** shortest path length distributions

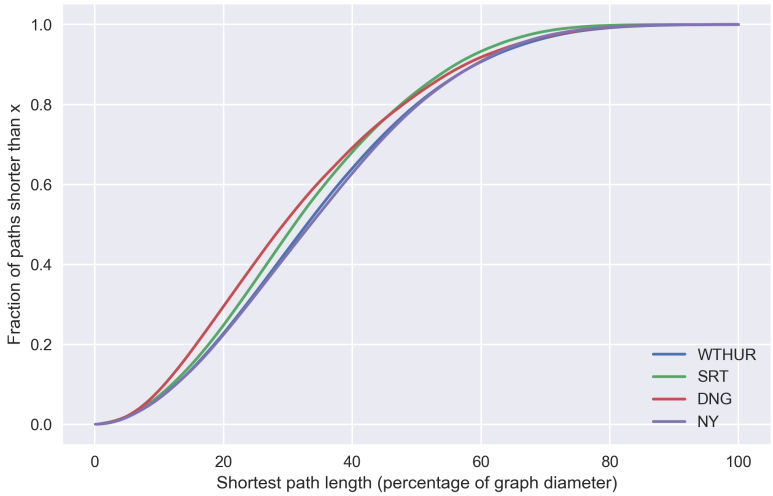


Figure 3.3: CDF plot of **unweighted** shortest path length distributions

Chapter 4

Models

In this chapter we describe the different types of models we investigate. The models can be divided into three broad categories: existing node embedding technique models, graph neural network models and hyperbolic graph convolutional network models. In the first category, we leverage various existing node embedding techniques [9, 11, 10] and examine how well they preserve distance if the node embeddings are fed into a simple multilayer perceptron. This is very similar to what researchers did in a paper we discussed earlier [8], but instead of evaluating the embeddings on social graphs, we evaluate them on road graphs, which have a much larger diameter. In the last two categories, we instead propose graph neural network models to generate distance preserving embeddings in Euclidean or hyperbolic space, that can efficiently be used to predict shortest path distances.

4.1 Existing Node Embedding Techniques

In total, we examine three different already existing node embedding techniques for their distance preserving capabilities: node2vec, GraRep and ProNE. To obtain the node embeddings these techniques produce, we use the respective implementations provided by the nodevectors Python package [23]. We will first explain how these techniques work, describe the differences between them, and then describe the model we use to evaluate the techniques.

4.1.1 node2vec

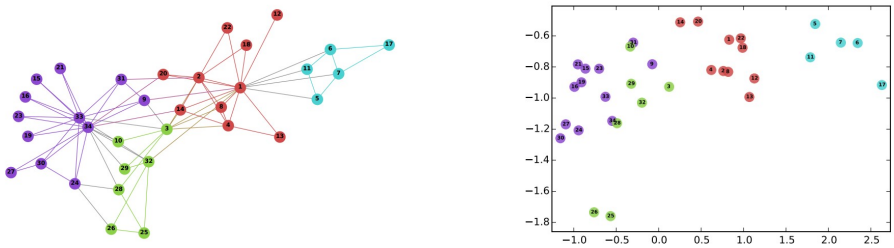
The node embedding technique *node2vec* [11] generates continuous feature representations for nodes in a graph. It is a modification of the previously published *DeepWalk* [12] method, which in turn generalizes techniques used in language models such as *word2vec*, like the *Skip-Gram model* [13], to graphs.

The goal of the Skip-Gram model is to find vector representations of words that capture their meaning so that words that are semantically close to each other are also close in the vector space. For that it uses the heuristic that the meaning of a word is determined by the words that occur frequently in its context. Or, to put it another way, when two words appear in a similar context, they usually have a similar meaning. DeepWalk builds on that approach by performing from each node a certain number of fixed-length random walks over the graph to build node contexts. These contexts can then be processed using the Skip-Gram model to generate node embeddings that preserve structural information in the graph. More precisely, DeepWalk optimizes the following objective function:

$$\min_{\Phi} -\log \Pr(\{v_{i-w}, \cdots, v_{i-1}, v_{i+1}, \cdots, v_{i+w}\} \mid \Phi(v_i)) \tag{4.1}$$

Where $\Phi : v \in V \mapsto \mathbb{R}^d$ is a mapping function that can be stored as a $|V| \times d$ matrix that maps each vertex to its d-dimensional vector representation and the vertices $\{v_{i-w}, \cdots, v_{i-1}, v_{i+1}, \cdots, v_{i+w}\}$ are the context of a given vertex v_i on a given random walk for a specific sliding window size w. Therefore the objective function tries to maximize the probability of a set of neighborhood vertices given the vector representation of a vertex v_i.

As calculating $\Pr(v_j \mid \Phi(v_i))$ for some vertex $v_j \in V$ involves calculating the



(a) Input: Karate Graph (b) Output: Representation

Figure 4.1: 2-dimensional embeddings generated by DeepWalk on the Karate Graph [12]

normalization factor of a softmax, using a classic softmax would be very computationally expensive. Therefore the authors of DeepWalk propose the use of a modified softmax called the hierarchical softmax that lowers the computational cost of calculating $\Pr(v_j \mid \Phi(v_i))$ from $\mathcal{O}(|V|)$ to $\mathcal{O}(\log |V|)$.

While node2vec optimizes the same objective function as DeepWalk it introduces two hyperparameters p, q that enable biased random walks. The hyperparameters allow the random walk to interpolate between a pure breadth-first-search that would best model homophily and a pure depth-first-search that would best model the structural equivalence. Assuming we just traversed the edge (t, v) from node t to node v and are currently residing at node v, the unnormalized transition probability $\alpha_{pq}(t, x)$ from node v to node x in an unweighted graph is defined as formula 4.2.

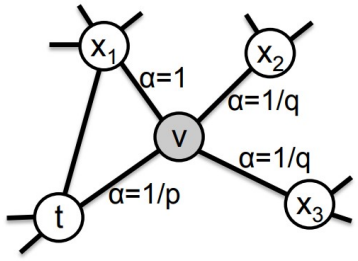


Figure 4.2: Illustration of the random walk procedure in node2vec [11]

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases} \tag{4.2}$$

Where d_{tx} denotes the shortest path distance between nodes t and x. Intuitively, the *return parameter*, p, controls the probability of backtracking to the previously visited node, and thus setting p to a high value encourages the path to explore the graph, while a low value keeps the path "local". On the other hand, the *in-out parameter*, q, allows the random walk to differentiate between "inward" and "outward" nodes. Choosing $q > 1$ makes the walk biased towards close nodes to t and therefore approximate breadth-first-search behavior, while choosing $q < 1$ makes the walk inclined to visit nodes further away from t and therefore approximate depth-first-search behavior. The added flexibility in exploring neighborhoods allows node2vec to learn richer node representations than DeepWalk. In order to speed up the training process, node2vec approximates the softmax activation function by negative sampling.

4.1.2 GraRep

GraRep [9] is a model for learning vector representations of vertices in a graph that preserve global structural information. Unlike node2vec, GraRep takes a more analytical approach and does not perform random walks. Instead, it reformulates the problem of defining the k-step relationship between two nodes to a matrix factorization task.

The k-step transition probability matrix A^k is defined as the k-th power of the 1-step transition probability matrix A that is defined as

$$A = D^{-1}S \tag{4.3}$$

where S is the adjacency matrix of the graph and the diagonal matrix D is the degree matrix defined as

$$D_{i,j} = \begin{cases} \sum_p S_{i,p}, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases} \tag{4.4}$$

This leads to entry $A^k_{i,j}$ exactly referring to the transition probability from node v_i to node v_j where the transition consists of exactly k steps. GraRep first calculates the different transition probability matrices $A^1, A^2, \ldots, A^K$ up to a fixed hyperparameter K called the *order*. GraRep then calculates the log probability matrices $Y^k_{i,j}$ for $k \in \{1, 2, \ldots, K\}$ defined as

$$Y^k_{i,j} = W^k_{i,j} \cdot C^k_{i,j} = \log\left(\frac{A^k_{i,j}}{\sum_t A^k_{t,j}}\right) - \log(\beta) \tag{4.5}$$

where $\beta = \lambda / |V|$ and λ is a hyperparameter indicating the number of negative samples. In order to reduce noise, each Y^k is modified into a matrix X^k, where $X^k_{i,j} = \max(0, Y^k_{i,j})$. GraRep then calculates the singular value decomposition (SVD) of each $X^k = U^k \Sigma^k (V^k)^T$ and approximates $W^k = U^k_d (\Sigma^k_d)^{\frac{1}{2}}$ using the d largest eigenvalues and the corresponding eigenvectors, where d is the embedding dimensionality hyperparameter. This process is also called truncated singular value decomposition (tSVD). The i-th row of matrix $W^k \in \mathbb{R}^{|V| \times d}$ can then be seen as the vector representation of the i-th vertex capturing k-step relational information. Finally, the different k-step representations W^k are either concatenated, summed up or merged in some other way to obtain the graph representation matrix W.

A key advantage of GraRep is that the different k-step relational information is not projected into the same subspace as in node2vec, and instead can be preserved in distinct subspaces.

4.1.3 ProNE

ProNE [10] is the most recently published of the three embedding techniques and aims to increase efficiency and scalability over the other methods while maintaining or even slightly improving the performance. The embedding generation is based on a two-step process. In the first step the embeddings are initialized efficiently by formulating the task as a sparse matrix factorization, similar to GraRep. In the second step the embeddings are then enhanced by propagating

them in a spectrally modulated space. The second step can also be applied to other embedding techniques to improve their performance for various downstream tasks.

More precisely, in the first step they use the set of edges E as node–context pair set $\mathcal{D} = E$ and define the occurence probability of context v_j given node v_i as

$$\hat{p}_{i,j} = \sigma(r_i^T c_j) \tag{4.6}$$

where σ is the sigmoid function and $r_i, c_i \in \mathbb{R}^d$ are the embedding and context vectors of node v_i, respectively. They then define the objective function to be minimized as

$$L = -\sum_{(i,j)\in\mathcal{D}} \left[p_{i,j} \ln \sigma(r_i^T c_j) + \tau P_{\mathcal{D},j} \ln \sigma(-r_i^T c_j)\right] \tag{4.7}$$

where $p_{i,j} = A_{i,j}/D_{i,i}$, with A being the adjacency matrix and D being the diagonal degree matrix. The second part of the objective function aims to avoid the trivial solution $r_i = c_j$ by negative sampling, where τ is the negative sample ratio and $P_{\mathcal{D},j}$ are the negative samples associated with context node v_j. They then reformulate the problem of minimizing the objective function into a matrix factorization task. To solve the matrix factorization task, they use a truncated singular value decomposition (tSVD) as used by GraRep to approximate W^k. However, since the matrix to be factorized is sparse, a randomized tSVD can be used to speed up this process.

In the second step they then propagate the initialized d-dimensional embeddings $R_d \in \mathbb{R}^{|V|\times d}$ in a spectrally modulated space. This is to allow the embeddings to capture global structural information, since in the first step only edges are used as node-context pairs and therefore only local structural information is initially captured. Formally, given the initial embeddings R_d, the following propagation rule is used:

$$R_d \leftarrow D^{-1}A(I_n - \tilde{L})R_d \tag{4.8}$$

where D is the degree matrix, I_n is the identity matrix, $\tilde{L}$ is the Laplacian filter and combined, $D^{-1}A(I_n - \tilde{L})$ is the modulated network of G.

4.1.4 Comparison

To better understand the differences between the embeddings generated by these three techniques, Figures 4.4 and 4.5 on the following pages illustrate the task our model must solve if we are using addition or difference as binary operation. Every plot is a graph-technique pair, where each point is either the addition $emb(v_i) + emb(v_j)$ or difference $|emb(v_i) - emb(v_j)|$ of the embeddings of

nodes v_i, v_j and the points color represents the normalized shortest path distance $dist(v_i, v_j)/d_{max}$. Even though these visualizations are limited to 2-dimensional embeddings, they do allow for some interesting insights. Two of them are:

- The structure of the embeddings varies greatly. While the embeddings produced by node2vec are in some kind of bowl shape, the embeddings generated by a low order GraRep are spread along two perpendicular axis and increasing the order causes them to spread further. In contrast, the spectral propagation of ProNE generates a ball of radius 2.
- Some techniques seem to preserve shortest path distances better than others. While node2vec and ProNE do not seem to preserve much structure, GraRep seems to be quite structure-preserving, especially at a higher order.

4.1.5 Evaluation Model

Regardless of the technique used to create the embeddings, all models have the same basic neural network architecture as shown in Figure 4.3. For a pair of nodes v_i, v_j we first apply one of three binary operations to their embeddings $emb(v_i), emb(v_j) \in \mathbb{R}^d$. We either concatenate them $(emb(v_i), emb(v_j))$ resulting in a $2d$-dimensional input, add them $emb(v_i)+emb(v_j)$ or we take their difference $|emb(v_i) - emb(v_j)|$ which both result in a d-dimensional input. Therefore, depending on what binary operation we use to combine the embeddings, the input layer of our network has size d or $2d$. We fix the size of the hidden layer to 128 as making it independent of the embedding size makes it easier to compare the performances of different embedding sizes. We use a rectified linear unit (ReLU) as activation function in the hidden layer. In the output layer we use a softplus activation function as the predicted distances should never be negative.

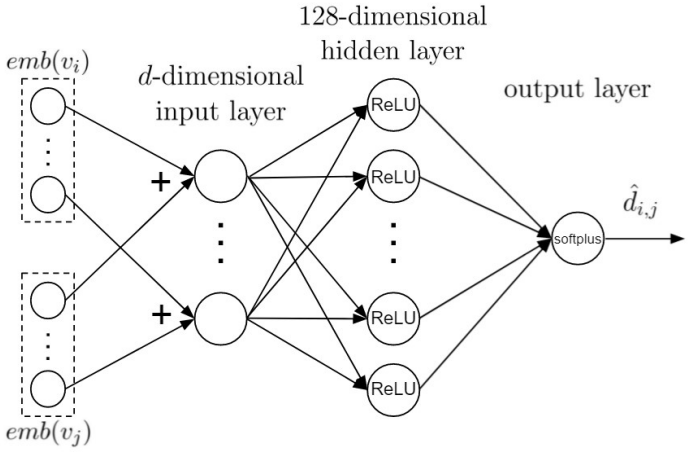


Figure 4.3: Example of model architecture using addition as binary operation

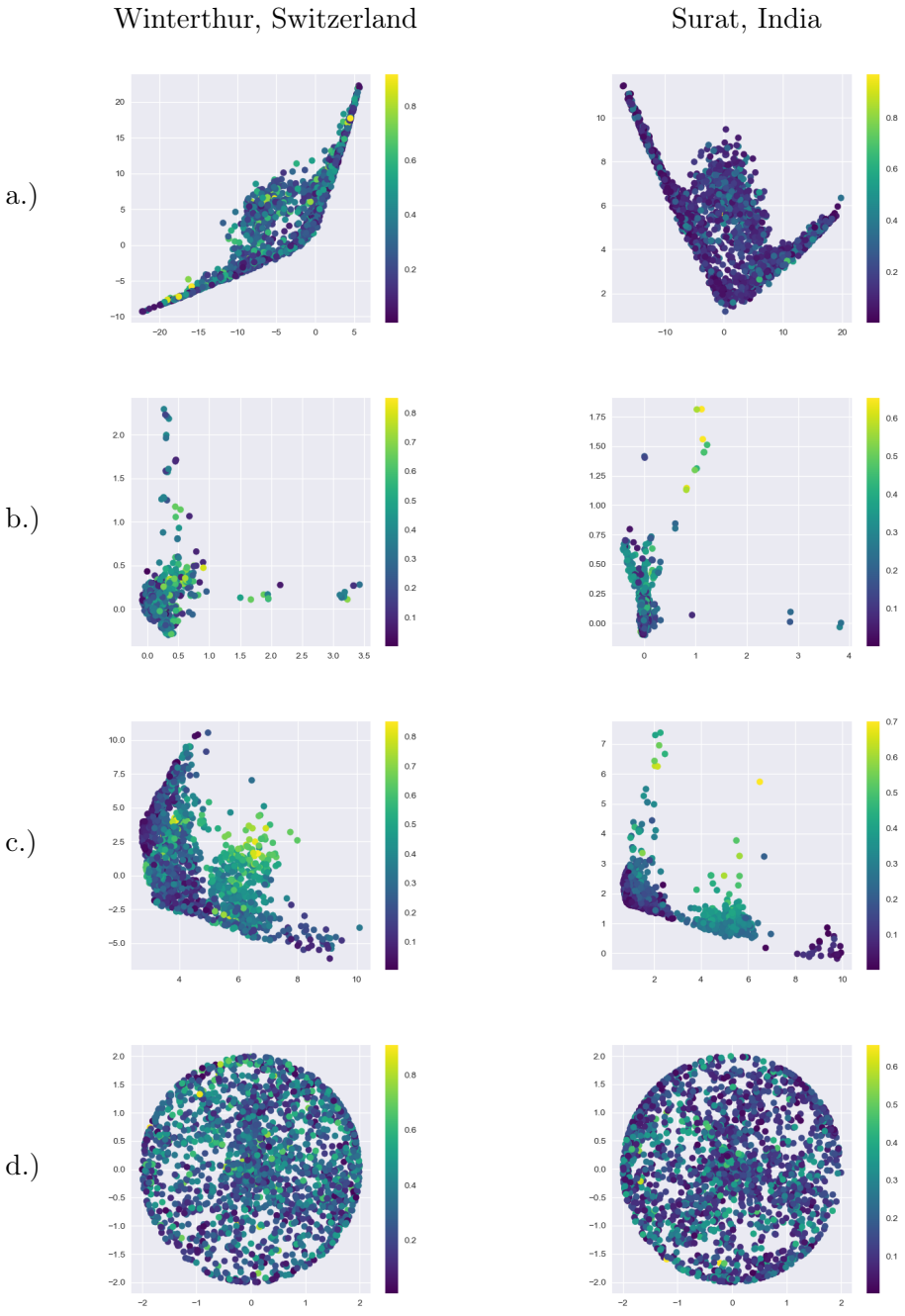


Figure 4.4: Illustration of the task the model has to perform depending on the technique we use to generate the embeddings. The coordinates of each dot are the **sum** of the embedding vectors of two nodes v_i, v_j and the dot's color represents the normalized shortest path distance between v_i and v_j. a.) node2vec (walklength=80, windowsize=10, p=1, q=1) b.) GraRep (order=10) c.) GraRep (order=100) d.) ProNE.

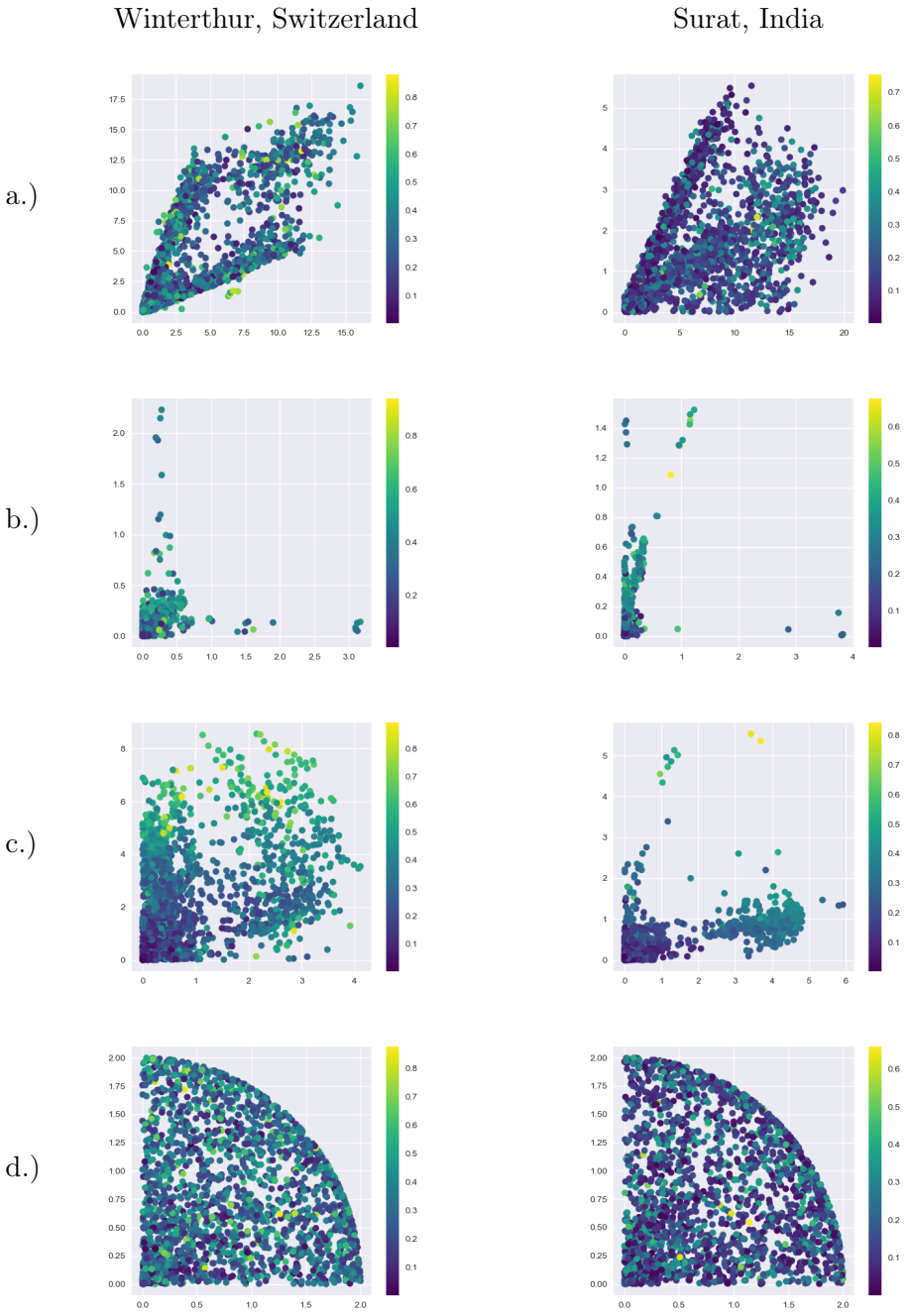


Figure 4.5: Illustration of the task the model has to perform depending on the technique we use to generate the embeddings. The coordinates of each dot are the **difference** of the embedding vectors of two nodes v_i, v_j and the dot's color represents the normalized shortest path distance between v_i and v_j. a.) node2vec (walklength=80, windowsize=10, p=1, q=1) b.) GraRep (order=10) c.) GraRep (order=100) d.) ProNE.

4.2 Graph Neural Networks

As in the previous section, we will first look at the theoretical background of graph neural networks and then describe the specific graph neural networks we used for our shortest path distance prediction task.

4.2.1 Message Passing

Graph neural networks (GNNs) are a class of neural networks that has recently become popular for processing data represented by graph data structures. Graph neural networks are used for various tasks such as node classification, graph classification and link prediction. We will focus on general message passing graph neural networks as almost all popular graph neural networks are of this type.

Given a graph G and for each node v_i an initial representation vector h_i^0, the message passing graph neural network performs T message passing rounds, consisting of propagation and aggregation steps, which can be described as

$$h_i^t = q_t \left(h_i^{t-1}, \bigcup_{\forall j : v_j \xrightarrow{k} v_i} f_t(h_i^{t-1}, k, h_j^{t-1}) \right) \tag{4.9}$$

where q_t is the update function and f_t is the message building function, both must be differentiable. The outputs of f_t, the messages, are aggregated over all j such that the edge $(v_j, v_i) \in E$ and has attributes k. For aggregation, any permutation invariant function can be used, such as max, sum or mean. The initial node representation vectors can be anything from embeddings generated using previously discussed techniques to node attributes, or just randomly initialized vectors. After performing T message passing rounds in which all node representation vectors are updated in parallel, each node v_i is assigned a final representation vector h_i^T. Therefore T is also called the depth or number of message passing layers of the graph neural network. These final representation vectors can then be fed into further downstream tasks or they can be extracted and used as embeddings.

4.2.2 Base Architecture

To build our graph neural networks we use the message passing base class provided by the PyTorch Geometric library for PyTorch [24]. We propose several graph neural networks for end-to-end shortest path distance prediction on road graphs that are all based on the following architecture:

Starting with the 2-dimensional coordinate data of each node as initial embeddings, they are first passed through a single, fully connected linear layer with

input size 2 and output size d. The output of the first hidden layer is then fed into l consecutive message passing layers whose output dimension is d. The kind of message passing layer used differs slightly for the different models. In general, we use a rectified linear unit as activation function after message passing layers, except for the last message passing layer where we do not use any activation function to allow for negative embedding coordinates as well. Either only the output of the last message passing layer or a combination of all outputs of the individual message passing layers then represent the final embedding. Depending on the model, the embeddings are additionally normalized before being used for distance evaluation. To predict the distance between two nodes v_i, v_j we either pass a combination of their embeddings to a multilayer perceptron or directly calculate the L_p-norm of $|emb(v_i) - emb(v_j)|$.

4.2.3 CoordNet

CoordNet (CN) uses a slight modification of the edge convolutional operator [25] for message passing, which can be described as

$$h_i^t = \sum_{j:(v_i,v_j)\in E} f_t\left(h_i^{t-1} \parallel h_j^{t-1} - h_i^{t-1} \parallel w((v_i, v_j))\right) \tag{4.10}$$

where $\parallel$ is the concatenation operator, $w((v_i, v_j))$ is the normalized weight of edge (v_i, v_j) and f_t is a simple two-layer multilayer perceptron. The first layer of f_t MLP has input size $2d$, output size d and uses a ReLU activation function, while the second layer has input and output size d without any activation function. To aggregate the messages a simple summation is used. The output of the last message passing layer is normalized so that all embedding vectors have an L_2-norm equal to 1. To predict the shortest path distance between two nodes their respective embedding vectors are concatenated and passed to a two-layer MLP. The first layer of the MLP has input size $2d$, output size d and uses a ReLU activation function. The second layer has input size d and output size 1 as the output of this layer represents the predicted distance.

4.2.4 StackedCoordNet

StackedCoordNet (SCN) is a small modification of CoordNet. It uses the same modified edge convolutional operator as described in 4.10 and the same MLP to predict the distances based on the generated embeddings. It differs from CoordNet in that it uses the outputs of all message passing layers (before the ReLU activation function) and the input to the first message passing layer to generate embeddings. More precisely, it sums all outputs and the input of the first layer and normalizes them so that every embedding vector has an L_2-norm of 1. The reason for this modification is that it is harder for the model to learn

when only the output of the last layer is used, and we have found that models with fewer layers predict short distances more accurately, while models with more layers predict long distances more accurately.

4.2.5 StackedEuclidCoordNet

StackedEuclidCoordNet (SECN) is another slight modification to StackedCoordNet. It is exactly the same as StackedCoordNet except that it does not use an MLP for the final distance prediction and instead directly calculates the Euclidean distance between $emb(v_i)$ and $emb(v_j)$ to predict the shortest path distance between v_i and v_j. We found that although an MLP generally leads to a lower training error, it is prone to overfitting. Moreover, calculating the Euclidean distance between two embedding vectors is even more efficient than sending them through a two-layer MLP.

4.2.6 StackedAngleCoordNet

StackedAngleCoordNet (SACN) is also a slight modification of StackedCoordNet. It is identical to StackedEuclidCoordNet, but instead of calculating the Euclidean distance between two embeddings, the angle between them is calculated in radians to estimate their shortest path distance.

4.3 Hyperbolic Graph Convolutional Network

The hyperbolic graph convolutional network (HGCN) [26] is a modification of classic graph neural networks proposed by Chami et al. in a paper of the same name. Classic graph neural networks embed nodes in Euclidean space, which has been shown to cause large distortions when embedding real-world graphs with a hierarchical structure. Hyperbolic geometry offers a great advantage over Euclidean geometry, as it allows to embed hierarchical graphs with much less distortion.

However, transferring the concept of graph neural networks to hyperbolic space presents certain challenges: (1) Converting Euclidean input node features into useful inputs for the HGCN. (2) Performing the aggregation step and linear transformations in hyperbolic space. (3) Choosing the best hyperbolic space curvature for every layer.

The HGCN overcomes these challenges in the following way: (1) It transforms input node features from Euclidean space into hyperbolic space using an exponential map. (2) The aggregation step, as well as linear transformations, are performed in a Euclidean tangent space as shown in 4.6. More precisely, the hyperbolic points are first projected into a Euclidean tangent space using a logarithmic map, then the aggregation or linear transformation is performed, and finally the points are projected back into hyperbolic space using an exponential map. (3) At each layer, the feature transformations can be performed in hyperbolic space with individual curvature. One has the option of either manually selecting the individual curvatures or having them trained. The HGCN uses the hyperboloid model of hyperbolic space for its simplicity and numerical stability.

Definition 4.1 (Hyperboloid model)**.** The *hyperboloid model* is a model of d-dimensional hyperbolic space, in which points are represented as points on the forward sheet of a two-sheeted $(d+1)$-dimensional hyperboloid. We denote $\mathbb{H}^{d,K}$ as the hyperboloid manifold of dimension d with curvature $c = -1/K$ $(K > 0)$.

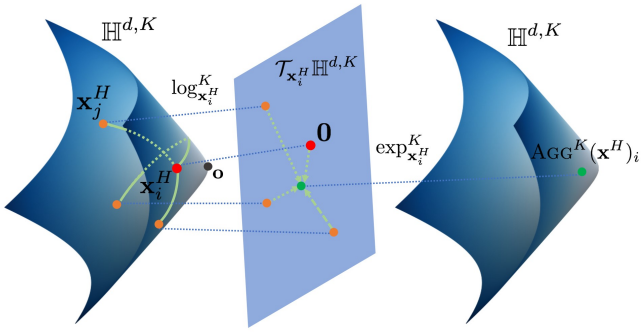


Figure 4.6: The HGCN aggregation step first maps messages into a Euclidean tangent space, performs the aggregation in the tangent space, and then maps them back to hyperbolic space. [26]

4.3.1 DPModel

To take advantage of the HGCN's ability to embed real graphs with low distortion for our task, we build a model that uses the HGCN to generate hyperbolic node embeddings and then directly calculates the hyperbolic distance between the embeddings of two nodes to predict their shortest path distance. This model is called *DPModel*, which stands for distance prediction model. The DPModel uses the coordinates of the nodes as initial embeddings and passes them to a l-layer HGCN. The HGCNs implementation is available on the research team's GitHub[1]. In each layer of the HGCN the activation function, as well as the dimensionality

[1] https://github.com/HazyResearch/hgcn

and curvature of the output space, can be specified. We use a rectified linear unit as activation function in all layers except for the final layer where we use no activation function. In addition, we set the output dimensionality of all layers to a certain value d and either use the same curvature c for all layers or have the curvatures trained. The HGCN outputs for each node v_i an embedding $emb(v_i) \in \mathbb{H}^{d,K}$ represented as coordinates in the hyperboloid model of hyperbolic space. Finally, to predict the shortest path distance between v_i and v_j a simple hyperboloid distance computation is performed as defined in 4.3.

Definition 4.2 (Minkowski inner product)**.** Let $\langle \cdot, \cdot \rangle_{\mathcal{L}} : \mathbb{R}^{d+1} \times \mathbb{R}^{d+1} \to \mathbb{R}$ denote the Minkowski inner product:

$$\langle x, y \rangle_{\mathcal{L}} = -x_0 y_0 + x_1 y_1 + \ldots + x_d y_d$$

Definition 4.3 (Distance in Hyperboloid Model)**.** The distance between two points (x, y) in $\mathbb{H}^{d,K}$ can be computed as:

$$d_{\mathcal{L}}^{K}(x, y) = \operatorname{arcosh}(-\langle x, y \rangle_{\mathcal{L}}) \cdot \sqrt{K}$$

Chapter 5

Experiments

In this chapter, we describe the experiments we performed and the settings we used. We also write about the conclusions we draw from the results of the experiments and reason about why the experiments turned out the way they did. We first describe the general training settings and then specify them in the individual subsections.

5.1 Training Settings

During preprocessing, we normalized the coordinates (x_i, y_i) of each node v_i in the following way:

$$x_i' = \frac{x_i - \min(x)}{\max(x) - \min(x)} \qquad y_i' = \frac{y_i - \min(y)}{\max(y) - \min(y)}$$

We also divided the weight of each edge by the weight of the edge with the maximum weight. We did the same with the shortest path distances on which we trained the model, we divided each distance by $d_{\max}$. To train our models, we used the mean relative error (MRE) as loss function. We prefer the mean relative error to the mean absolute error because it gives more weight to the prediction errors on short paths. We believe this is desirable because accurately predicting short distances is more difficult than predicting long distances. However, we use the mean absolute error 2.4 (divided by the diameter of the respective graph) together with the mean relative error 5.1 as evaluation metrics. Unless otherwise specified, the Adam or Riemannian Adam (HGCN) [26] optimization algorithm was used for training with a learning rate of 0.001 and the batch size was 512. In general, we used $l = 3$ message passing layers in our models as we found that to be a good compromise between efficiency and accuracy.

Definition 5.1 (Mean Relative Error)**.**

$$MRE = \frac{1}{n} \sum_{i=1}^{n} \frac{|\hat{y}_i - y_i|}{y_i}$$

5.2 Results

In this section we present the results of our experiments. We first review the general results obtained with the various existing node embedding techniques and our neural network models and then address some aspects in detail.

5.2.1 Existing Node Embedding Techniques

To compare the existing node embedding techniques we trained the evaluation model for 20 epochs on randomly selected 90% of the APSP data and tested on the other 10% of the APSP data. Although the embedding techniques are capable of using edge weights, we did not use them because we did not find large differences between the performances with edge weights and without edge weights. We used the default settings for node2vec, i.e., we made 10 walks per node of length 80, used a window size of 10, and set p and q to 1. GraRep10 and GraRep100 are GraRep embeddings with orders 10 and 100 respectively. We used 128-dimensional embeddings for all tests.

		WTHUR		SRT	
		MRE	MAE	MRE	MAE
node2vec	Addition	0.2440	0.0616	0.3063	0.0497
	Difference	0.4036	0.1195	0.9999	0.1735
	Concatenation	0.1858	0.0486	0.2022	0.0340
GraRep10	Addition	0.0922	0.0199	0.0847	0.0109
	Difference	0.0859	0.0214	0.0795	0.0107
	Concatenation	0.0748	0.0181	0.0613	0.0086
GraRep100	Addition	0.0748	0.0133	0.0694	0.0071
	Difference	**0.0571**	0.0128	0.0565	0.0073
	Concatenation	0.0573	**0.0120**	**0.0502**	**0.0058**
ProNE	Addition	0.0804	0.0174	0.0946	0.0121
	Difference	0.2222	0.0623	0.3079	0.0544
	Concatenation	0.0660	0.0156	0.0674	0.0096

Table 5.1: Results

From the results, we can conclude that node2vec does not have a competitive distance preserving capability compared to GraRep and ProNE. We can also see that the order 100 GraRep performs best and that the accuracy changes only slightly when using difference or concatenation as a binary operator, which corresponds to our observations from the task visualizations in Figures 4.4 and 4.5. However, ProNE generates embeddings even quicker than the order 10 GraRep and is almost as distance preserving as the order 100 GraRep, as long as we do not use it with the subtraction binary operation.

5.2.2 Graph Neural Networks

To evaluate the performances of our proposed graph neural network models we trained them for 100 epochs on randomly selected 10% of the APSP data and tested them on the other 90% of the APSP data. We did separate tests with 32-dimensional and 128-dimensional embeddings.

	WTHUR		SRT		DNG	
	MRE	MAE	MRE	MAE	MRE	MAE
CN	0.0480	0.0091	0.0435	**0.0041**	0.0702	0.0151
SCN	**0.0460**	**0.0085**	**0.0396**	**0.0041**	0.0693	**0.0123**
SECN	0.0557	0.0120	0.0492	0.0065	0.0533	0.0128
SACN	0.0545	0.0115	0.0477	0.0062	**0.0509**	0.0124

Table 5.2: Results with 32-dimensional embeddings

	WTHUR		SRT		DNG	
	MRE	MAE	MRE	MAE	MRE	MAE
CN	0.0309	0.0053	0.0378	0.0038	0.0529	0.0115
SCN	**0.0281**	**0.0047**	**0.0309**	**0.0034**	0.0518	**0.0100**
SECN	0.0486	0.0106	0.0450	0.0062	0.0537	0.0137
SACN	0.0474	0.0101	0.0436	0.0060	**0.0485**	0.0117

Table 5.3: Results with 128-dimensional embeddings

An interesting finding from the test results is that the StackedCoordNet performs best on the smaller graphs, but is beaten by the StackedAngleCoordNet on the larger DNG graph, which could be an indicator of the better generalization ability of StackedAngleCoordNet. Another finding is that while all models are able to take advantage of the larger embedding size, the models that use an MLP as evaluation method are able to take even more advantage of it than the other models. Overall, SECN and SACN perform similarly, which is not surprising given their architectural similarities, but SACN generally outperforms SECN. Similarily, SCN persistently performs better than CN, which confirms the benefit of a stacked architecture in terms of prediction accuracy.

5.2.3 Hyperbolic Graph Convolutional Network

To test the performance of our proposed DPModel, we performed multiple tests on the WTHUR and SRT graphs. Since the HGCN used to generate the embeddings is quite inefficient due to the large number of mapping operations between the tangent spaces and hyperbolic space it must perform, we chose to train 50 epochs with a bigger batch size of 4096 on randomly selected 10% of the APSP data. As before we then tested the trained model on the other 90% of the APSP data. We performed tests with 32-dimensional and 128-dimensional embeddings and also used $l = 1$ or $l = 3$ message passing layers in the HGCN. We fixed the curvatures of the hyperbolic embeddings to $c = 1$ since training them proved to be numerically unstable and did not yield consistent results.

		WTHUR		SRT	
		MRE	MAE	MRE	MAE
DPModel	$l = 1$	0.1077	0.0266	0.1063	0.0166
	$l = 3$	**0.0978**	**0.0233**	**0.0779**	**0.0125**

Table 5.4: Results with 32-dimensional embeddings

		WTHUR		SRT	
		MRE	MAE	MRE	MAE
DPModel	$l = 1$	0.1091	0.0266	0.1271	0.0213
	$l = 3$	**0.1004**	**0.0242**	**0.0665**	**0.0081**

Table 5.5: Results with 128-dimensional embeddings

From the test results, it can be concluded that although the DPModel with $l = 3$ performs better than with $l = 1$, the performance of the 3-layer DPModel is still not comparable to that of the graph neural network models. A reason for the relatively poor performance of DPModel on these road graphs could be that the road graphs are not hyperbolic enough. Gromov's hyperbolicity is a measure of how hyperbolic a graph is. A lower value means that the graph is more hyperbolic, e.g. a tree graph has a hyperbolicity of 1. However, the hyperbolicity of the WTHUR graph is only 17 while the SRT graph is even less hyperbolic with a value of 20. Even though the DPModel is relatively inaccurate compared to the other method it also has almost an order of magnitude fewer parameters than SACN and SECN.

5.2.4 Landmark Training

Since computing the all-pairs shortest path (APSP) data, or even 10% of it, is often not efficiently possible for large graphs, we investigate how our proposed neural graph network models perform when only a very small amount of training data is available. We randomly selected $k \ll |V|$ landmark vertices and calculated the single-source shortest path (SSSP) data from these landmarks to all other vertices using exact shortest path distance methods. We then trained our models for $\lceil 400/k \rceil$ or, in the case of NY, $\lceil 100/k \rceil$ epochs on the obtained SSSP data. We tested the trained models on the APSP data or, in the case of NY, on the SSSP data from 1000 landmarks, which were completely disjoint of the landmark sets used to train the models. We tested the models with 32-dimensional embeddings as well as with 128-dimensional embeddings.

		WTHUR		SRT		NY	
		MRE	MAE	MRE	MAE	MRE	MAE
CN	k = 5	0.3976	0.0699	0.3586	0.0484	0.2096	0.0630
	k = 10	0.3225	0.0695	0.1478	0.0198	0.1166	0.0340
	k = 50	0.2173	0.0486	0.3382	0.0358	0.0517	0.0146
	k = 100	0.2231	0.0496	0.2964	0.0407	0.0399	0.0104
SCN	k = 5	0.3151	0.0571	0.4600	0.0633	0.2094	0.0636
	k = 10	0.1837	0.0412	0.1562	0.0203	0.0917	0.0249
	k = 50	0.2226	0.0508	0.1400	0.0178	0.0512	0.0139
	k = 100	0.2106	0.0497	0.3009	0.0374	0.0381	0.0095
SECN	k = 5	0.1269	0.0242	0.0948	0.0118	0.0608	0.0156
	k = 10	0.0936	0.0209	0.0812	0.0104	0.0618	0.0147
	k = 50	0.0897	0.0212	0.0814	0.0118	0.0392	0.0104
	k = 100	0.0891	0.0226	0.0851	0.0125	0.0369	0.0100
SACN	k = 5	0.1361	0.0261	0.0946	0.0117	0.0606	0.0153
	k = 10	0.0922	**0.0199**	**0.0772**	**0.0100**	0.0582	0.0134
	k = 50	**0.0877**	0.0207	0.0777	0.0105	0.0384	0.0103
	k = 100	0.0900	0.0230	0.0833	0.0121	**0.0358**	**0.0094**

Table 5.6: Results with 32-dimensional embeddings

While CoordNet and StackedCoordNet performed better in the previous tests where 10% of the APSP data was used for training they are beaten quite clearly by StackedEuclidCoordNet and StackedAngleCoordNet in these tests. This demonstrates the generalizability of the SECN and SACN. Although the models in these tests do not appear to be very accurate, with mean relative errors usually exceeding 5%, it is important to note that the time spent training them is orders of magnitude lower than in the previous tests. It may seem counterintuitive that the models perform better when trained on fewer landmarks, but this could also be because we trained the models for fewer epochs the more landmarks we used.

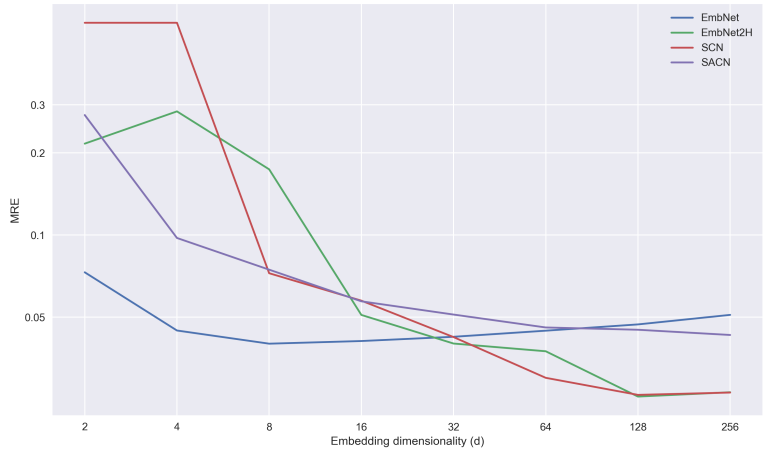
		WTHUR		SRT		NY	
		MRE	MAE	MRE	MAE	MRE	MAE
CN	k = 5	0.2862	0.0603	0.3831	0.0521	0.2132	0.0637
	k = 10	0.2296	0.0523	0.1360	0.0201	0.1468	0.0397
	k = 50	0.2177	0.0515	0.0201	0.0343	0.0518	0.0148
	k = 100	0.2216	0.0508	0.2404	0.0320	**0.0289**	**0.0067**
SCN	k = 5	0.2934	0.0604	0.3798	0.0553	0.2064	0.0619
	k = 10	0.1815	0.0427	0.1372	0.0192	0.1135	0.0337
	k = 50	0.2081	0.0441	0.1773	0.0237	0.0483	0.0132
	k = 100	0.1841	0.0404	0.1903	0.0242	0.0291	0.0070
SECN	k = 5	0.1189	0.0216	0.0990	0.0117	0.0652	0.0156
	k = 10	0.0827	0.0176	0.0694	**0.0084**	0.0684	0.0153
	k = 50	0.0798	0.0185	0.0767	0.0099	0.0383	0.0106
	k = 100	0.0790	0.0179	0.0802	0.0118	0.0357	0.0097
SACN	k = 5	0.1156	0.0206	0.0987	0.0117	0.0624	0.0149
	k = 10	0.0814	**0.0167**	**0.0677**	**0.0084**	0.0636	0.0141
	k = 50	0.0784	0.0178	0.0802	0.0106	0.0356	0.0096
	k = 100	**0.0756**	0.0170	0.0800	0.0117	0.0340	0.0090

Table 5.7: Results with 128-dimensional embeddings

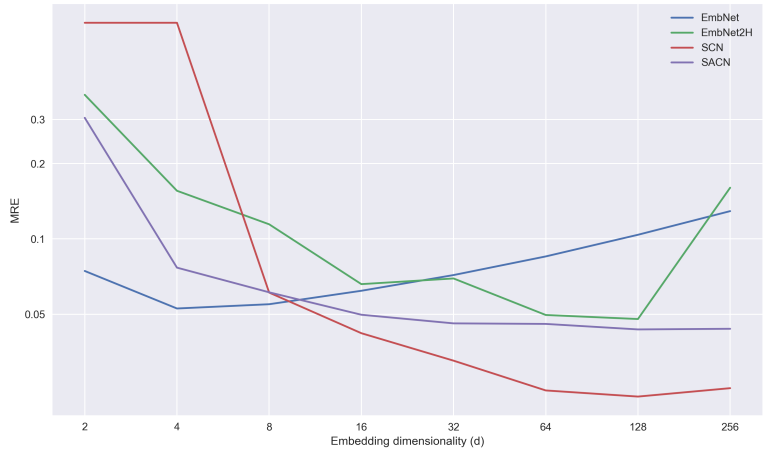
5.2.5 Impact of Embedding Dimensionality on Accuracy

To examine the impact of embedding dimensionality on the prediction accuracy, we plot the mean relative error of four different models on the WTHUR and SRT graphs as a function of the embedding sizes used. EmbNet and EmbNet2H are based on the following approach: we train a single fully connected linear layer that takes one-hot node representations as input and outputs the d-dimensional embeddings of the corresponding nodes. Therefore, we do not restrict the embedding generation at all and let the model learn the best embedding for each node with parameters that are completely independent of the embeddings of other nodes. EmbNet then uses a simple Euclidean distance calculation between the embeddings of the nodes to predict the distance, while EmbNet2H passes the concatenated embeddings of two nodes to a two-layer MLP identical to the one used in CoordNet and StackedCoordNet.

The results in Figure 5.1 are somewhat unexpected, as in some cases the loss increases again the larger the embeddings become. Since this is more noticeable for EmbNet and EmbNet2H, we suspect that this is due to the difficulty of the Adam optimizer to efficiently use the additional degrees of freedom. From a dimensionality of $d = 2$ to $d = 8$ the prediction accuracy increases most significantly. From $d = 8$ to $d = 64$ the accuracy generally continues to increase, while from $d = 64$ to $d = 256$ it mostly stagnates or even decreases again. Therefore, a dimensionality of between $d = 16$ and $d = 64$ seems to be the best compromise between prediction accuracy and space requirements on these graphs.



(a) WTHUR

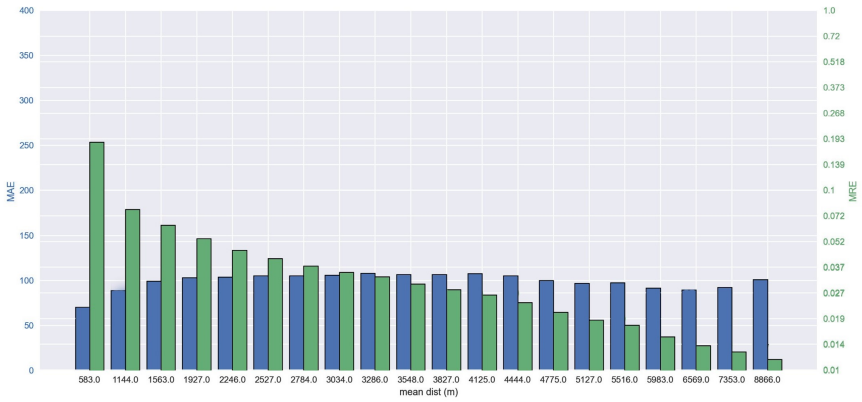


(b) SRT

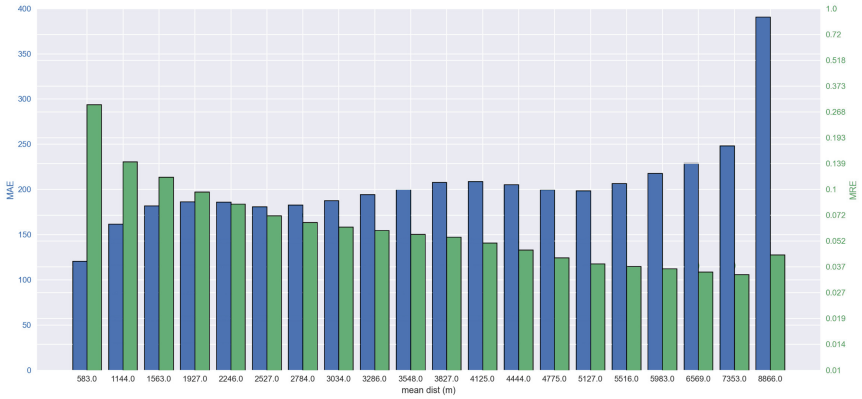
Figure 5.1: Impact of embedding dimensionality on MRE

5.2.6 Error Distribution over Path Lengths

To better investigate which path lengths are the most difficult to accurately predict, we plot the mean relative error and the mean absolute error as a function of the path length. We visualize the error distributions of SCN and DPModel on the APSP data of the WTHUR graph after training them on randomly selected 90% of the APSP data. More precisely, we split the APSP data into 20 sets ordered by path length and calculate the MRE and MAE on these sets.



(a) StackedCoordNet



(b) DPModel

Figure 5.2: MRE and MAE distribution over the path lengths

From the diagrams we can conclude that short paths are significantly more difficult to predict accurately. Even if the models are trained only on the short paths, the errors do not decrease significantly. An explanation for that could be that it is not possible to embed the complete bipartite graph $K_{1,3}$, also referred to as the "claw" graph, isometrically in any dimension. And since road graphs generally have many claw graphs as induced subgraphs this could be the reason why the models have difficulty accurately predicting short path lengths.

Chapter 6

Conclusion & Future Work

Our work on this thesis and the experiments we performed allow us to draw several conclusions. Furthermore, there are some ways in which our work could be expanded.

6.1 Conclusion

With respect to existing embedding techniques, we found that node2vec embeddings are significantly less distance preserving compared to GraRep and ProNE embeddings. We also found that while high-order GraRep embeddings appear to be the most distance preserving embeddings of the three techniques, ProNE embeddings come close to their distance preserving capabilities and are far more efficient. Regarding our four proposed graph neural network models, we were able to demonstrate that graph neural networks can be used to generate distance preserving embeddings with competitive accuracy. While the graph neural network models using an MLP as evaluation method performed better with relatively large amounts of training data, the graph neural networks calculating the angle or Euclidean distance for evaluation demonstrated their superior generalizability when only very little training data was used. Our hyperbolic graph convolutional network model did not yield the improvements we had hoped for, which we attribute to the lacking hyperbolicity of road graphs.

6.2 Future Work

Our work could be extended by finding ways to develop a graph neural network model that is capable of transferring knowledge learned from one or more graphs to other unseen graphs, i.e., without retraining. We tried to transfer our models to new graphs but weren't able to get below 10% mean relative error. To obtain transferable graph neural network models, one could improve on our normalization and regularization techniques.

Bibliography

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Numerische mathematik*, vol. 1, no. 1. Springer, 1959, pp. 269–271.

[2] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[3] L. Chang, J. X. Yu, L. Qin, H. Cheng, and M. Qiao, "The exact distance to destination in undirected world," *The VLDB Journal*, vol. 21, no. 6, pp. 869–888, 2012.

[4] T. Akiba, Y. Iwata, and Y. Yoshida, "Fast exact shortest-path distance queries on large networks by pruned landmark labeling," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 349–360.

[5] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao, "Orion: Shortest path estimation for large social graphs," in *Proceedings of the 3rd Wonference on Online Social Networks*, ser. WOSN'10. USA: USENIX Association, 2010, p. 9.

[6] X. Zhao, A. Sala, H. Zheng, and B. Y. Zhao, "Efficient shortest paths on massive social graphs," in *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 2011, pp. 77–86.

[7] J. Qi, W. Wang, R. Zhang, and Z. Zhao, "A learning based approach to predict shortest-path distances," in *EDBT*, 2020.

[8] F. S. Rizi, J. Schloetterer, and M. Granitzer, "Shortest path distance approximation using deep learning techniques," in *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 2018, pp. 1007–1014.

[9] S. Cao, W. Lu, and Q. Xu, "Grarep: Learning graph representations with global structural information," in *Proceedings of the 24th ACM international on conference on information and knowledge management*, 2015, pp. 891–900.

[10] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding, "Prone: Fast and scalable network representation learning." in *IJCAI*, vol. 19, 2019, pp. 4278–4284.

[11] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.

[12] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[13] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[14] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," *SIAM Journal on Computing*, vol. 32, no. 5, pp. 1338–1355, 2003.

[15] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, "Fast shortest path distance estimation in large networks," in *Proceedings of the 18th ACM conference on Information and knowledge management*, 2009, pp. 867–876.

[16] Y. Huang and W. McColl, "An improved simplex method for function minimization," in *1996 IEEE International Conference on Systems, Man and Cybernetics. Information Intelligence and Systems (Cat. No.96CH35929)*, vol. 3, 1996, pp. 1702–1705 vol.3.

[17] M. Nickel and D. Kiela, "Poincaré embeddings for learning hierarchical representations," vol. 30, 2017, pp. 6338–6347.

[18] OpenStreetMap contributors, "Planet dump retrieved from https://planet.osm.org ," https://www.openstreetmap.org, 2017.

[19] G. Boeing, "Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks," *Computers, Environment and Urban Systems*, vol. 65, pp. 126–139, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0198971516303970

[20] U. Road Networks, "Urban road network data," Jan 2016. [Online]. Available: https://figshare.com/articles/dataset/Urban_Road_Network_Data/2061897/1

[21] C. Demetrescu, A. Goldberg, and D. Johnson, "The shortest path problem : ninth dimacs implementation challenge," 2009.

[22] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," in *Proceedings of the 7th Python in Science Conference*, G. Varoquaux, T. Vaught, and J. Millman, Eds., Pasadena, CA USA, 2008, pp. 11 – 15.

[23] M. Ranger, "nodevectors," https://github.com/VHRanger/nodevectors, 2021.

[24] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[25] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *Acm Transactions On Graphics (tog)*, vol. 38, no. 5, pp. 1–12, 2019.

[26] I. Chami, Z. Ying, C. Ré, and J. Leskovec, "Hyperbolic graph convolutional neural networks," vol. 32, 2019, pp. 4868–4879.