MASTER DEGREE THESIS

MASTER'S DEGREE IN COMMUNICATIONS AND COMPUTER NETWORKS ENGINEERING

Fast and scalable routing in large telecommunication networks

POLITECNICO DI TORINO

Author:
Youkabed Sadri
248219

Supervisor:
Dr. Andrea BIANCO

Co-supervisor:
Dr. Cristina ROTTONDI

JULY, 2020

A thesis submitted in fulfillment of the requirements for the

Master's degree in Communications and Computer Networks Engineering

Written in

Telecommunication Networks Group
Department of Electronics and Telecommunications
Politecnico di Torino

Abstract

Fast and scalable routing in large telecommunication networks

Computing the shortest path between nodes in a graph is the cornerstone of many graph algorithms and applications. In many real-world problems such as routing in the network, finding the shortest path is critical. It is also a constrained problem of optimization which has been studied in recent years.

Typically, Traditional methods such as Dijkstra or breadth-first-search (BFS) can deliver a good solution in most cases. However, such heuristic algorithms do not satisfy the scalability and with the problem scale increasing, these approaches are inefficient and can take considerable time. Hence, the methods must be found to allow scalable graph processing with significant speed. In this thesis, we proposed a machine learning based method using the message passing algorithm to find the shortest path between nodes in the large telecommunication networks. Also, we show that the suggested algorithm has linear runtime complexity which provides reasonable time for finding the shortest path in a very large network graph.

KEYWORDS: Shortest Path Problem, Machine Learning, Graph Embedding, Graph networks, Message Passing.

Acknowledgements

I would like to thank those who with their guidance and advice have patiently helped and guided me throughout all stages of this research. My deepest appreciation goes to my supervisors, Prof. Andrea Bianco and Dr. Cristina Rottondi, for their support and motivation. I am thankful for your helpful comments and advices during my work.

Last but not least, I would like to express my deepest gratitude to the most valuable person in my life. My husband, Neman, who have always been there to support me in my life. This work is dedicated to my family.

List of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Finding the shortest path between nodes in a graph is the cornerstone of many graph algorithms and applications. In many real-world problems such as routing in the network, finding the shortest path is a critical issue.

Shortest path computing is a fundamental issue in the routing of today's telecommunication networks. Routing is usually composed of two entities: a routing protocol and an algorithm for routing. The routing protocol takes hold of the network states and the available resources and distributes the information throughout the network. With this information the routing algorithm determines the shortest paths. A good routing algorithm can help manage bandwidth and delay for successful support of video and audio applications in real time. All routing algorithms are usually involved in computing a shortest path from source to destination along the least costly route.

Shortest path computation is a constrained problem of optimization which numerous authors studied. Typically, traditional methods such as Dijkstra or breadth-first-search (BFS) can deliver a good solution in most cases. However, such heuristic algorithms do not satisfy the scalability and with the problem scale increasing, these approaches are inefficient and can take considerable time.

This thesis presents a method based on Machine Learning for routing in large telecommunications networks. The proposed approach is using GNN which is a new class of neural networks optimized for graph-structured data working. GNNs implement a scheme that aggregates information about the neighborhood, recursively. In our work we calculate the node embedding using MPNN. So MPNN applies an iterative message-passing algorithm to propagate information between graph nodes. The nodes and edges of the input graph at the MPNN end are labeled. Finally, by minimizing the error considering the Dijkstra solution, we try to learn the neural network how to correctly label the graph elements on the network. Then,

using a post-processing approach we can guarantee to find the path. As the results show, the ML framework is scalable, meaning that the model achieves almost optimum and fair efficiency, twice as large as the training graphs. Also, the algorithm proposed is of linear temporal complexity. While the larger the network would be, the longer the computational time used by Dijkstra. So, in large networks the suggested algorithm has been quicker and provides better performance. It can be used for routing in large telecommunication networks and with good speed and accuracy obtain the shortest possible path.

1.1 Outline of thesis

The rest of the study is structured according to the following:

Chapter 2 provides an overview of relevant topics in literature, focusing on studies that use machine learning techniques to route and find the shortest path in a network graph and its associated complexities. It highlights similarities and discrepancies with respect to our work.

Chapter 3 reviews the main principles for understanding the theory of the methods and the models we use and the experiments are based on. We provide a simple description of ML with a summary of learning models types. Then, the definition of shortest path problem is introduced. The concept of graph neural network and the graph networks framework which we used in our work are presented. At the end, we provide the MPNN concept which our framework is based on.

Chapter 4 presents a detailed description of the framework created to conduct our experiments. The scalable enhancement and computational complexity of the Machine Learning framework is described.

In Chapter 5 we analyze and compare the numerical results obtained. First we offer a summary of the datasets and determine the hyper-parameter for our framework and then focus on the assessment of performance metrics in the various scenarios considered.

Chapter 6 concludes and summarizes the goals reached from the performance evaluation.

Chapter 2

Related works

This chapter presents the preliminary materials and overview of relevant topics in literature, focusing on studies that leverage machine learning techniques in routing and finding the shortest path in a network graph and their corresponding complexities. At the end of those descriptions, similarities and differences will be highlighted with respect to the work developed in this thesis.

2.1 Shortest path

In a variety of applications, finding the shortest distances between two nodes in the graph is a significant primitive function. In social networks for example, the shortest distance is used to measure the centrality of the proximity. For many real-world issues, such as network routing, its calculation is a critical issue [1].

Finding the shortest path is also a constrained problem of optimization which has been studied in recent years by many authors. Typically, it's solved with heuristic algorithms, like the popular Dijkstra algorithm, which can faster deliver a good solution in most cases. However, with the problem scale increasing, these approaches are inefficient and can consume considerable CPU time [2].

Such conventional methods for computing distance between nodes do not scale with graph size. Efficient Dijkstra implementations calculate the shortest paths for a node to others in $O(n \log n + m)$ time complexity for a graph with n nodes and m edges. Dijkstra's minor generalization, known as the A^* algorithm, uses heuristic methods in order to calculate the shortest path. In practice, the A^* runs at least as fast as the Dijkstra algorithm, but the complexity of run time is still $O(n \log n + m)$. This time complexity is tolerable for graphs with small size, but for a large graph with one million nodes computation of shortest path can take up to one minute [1]. Machine Learning, which widely exploits parallelism in learning models, can easily solve this issue.

2.1.1 Dijkstra algorithm

The Dijkstra's Shortest Path Algorithm is a popular approach to the problem of Shortest Paths, that is to find the shortest path from the initial vertex r to the other vertex in a directionally weighted graph with non-negative weights [3].

The Dijkstra Algorithm is defined as follows [4]:

"Initialize y, p;

Set $S = V$;

While (S $\neq \emptyset$) :

Choose $v \in S$ with $y(v)$ minimum;

Delete v form S;

Scan v."

Initialize means to set $y(v) = \infty$ and $y(r) = 0$, and $p(v) = null$ for every node v except for r. Here y is a set of $y(v)$ for each node v, indicates the size of the shortest path found so far from r to v; P is a sequence of the 'parent vertex' $p(v)$ of each node v, i.e. the node before v in the shortest r −to−v path has been found so far; S is a list of nodes not yet scanned, and V is the list of all the vertices in the graph;

Scanning a node u means checking that $y(u) + w\, y(v)$ for each edge $a = (u, v)$ with weight w, is "correct" and otherwise correcting it.

Correcting an edge $a = (u, v)$ means adjusting the value of $y(v)$ to $y(u) + w$ such that a becomes correct, and setting $p(v) = u$ in the process.

2.2 ML-based Methods for finding the shortest path in the network graph

Here we look at some ML-based methods for determining the shortest path between two nodes in a network graph. Some of these approaches employed supervised learning and the other us leverage reinforcement learning.

2.2.1. Shortest Path Distance using Deep learning

In [1] authors use Vector Embedding to approximate the shortest distances of paths in large graphs, through deep learning. A feedforward neural network, which is fed with embedding vectors, can estimate the shortest path distances with relatively small distortion error. The approach is illustrated as follow:

They consider $G = (V; E)$ as an undirected graph with n nodes and m edges and the graph is unweighted. Using Graph embedding techniques they create a real-valued embedding vector $\emptyset(v) \in R^d$ per node.

The aim of this research is to use a feedforward neural network to approximate the distance as $\hat{d}$, assuming that a node pair $u, v \in V$ with the true shortest path distance $d_{u,v}$ are given.

They define the estimated distance $\hat{d}$ as:

$$\hat{d} : \emptyset(u) \times \emptyset(v) \rightarrow R^+$$

This equation illustrates the mapping of a pair of vector embeddings in G to a real-valued shortest distance of path $d_{u,v}$.

For training the neural network, the training pairs need to be extracted from the entire G graph. At first, the authors select a small number of l nodes as their landmarks, $l \ll n$. They then calculate the true shortest distances between each landmark and all the remaining nodes by using BFS. It provides $l(n - l)$ pairs for training. By applying a binary operation, namely subtraction, concatenation, average and point-wise multiplication, over the vector embeddings of training pair $< \emptyset(v), \emptyset(u) >$, they produce a joint representation as input into the neural network.

Table 1 lists the definitions of the binary operations. Finally, training set vectors serve as inputs for a feedforward neural network. The neural network maps the vectors of the input to a real-valued distance.

Table 1. Selecting Binary operator correspond to the i_{th} part of $\emptyset$.

Operator	**Symbol**	**Definition**
Subtraction	$\ominus$	$\emptyset_i(u) - \emptyset_i(v)$
Concatenation	$\oplus$	$(\emptyset(u), \emptyset(v))$
Average	$\oslash$	$\frac{\emptyset_i(u) - \emptyset_i(v)}{2}$
Hadamard	$\odot$	$\emptyset_i(u) * \emptyset_i(v)$

The feedforward network includes an input layer, a hidden layer and an output layer. The number of neurons in the input layer depends on the binary operation over vector embeddings. For instance subtraction needs d neurons whereas concatenation needs $2d$ neurons. They use ReLU as activation function for the two first layers. Since the network performs a task of regression, the output layer is a single unit of softplus which is a smoother ReLU version over the range of $[0, \infty]$. for assessment they use Mean Squared Error (MSE), which measures the

average squares of difference between the estimator and what is estimated. Also, they leverage the Stochastic Gradient Descent for optimizer. In general, this optimizer is quick and efficient for large-scale learning.

Computational Complexity

The method which is proposed has a linear runtime complexity. Because first, they learn vector embeddings which takes $O(n)$ time for precomputation, n is the number of nodes in the graph [1]. Then they use the landmark scheme in order to reduce the number of shortest path calculations needed to determine the ground truth. They only have to measure a BFS tree for each landmark by choosing a low, constant number of landmarks. The resulting values provide the shortest distances from the rest of nodes to these landmarks and they are enough to compose the training set. $l(n - l)$ training pairs are created by l nodes as landmarks, where $l \ll n$. With respect to the time complexity of BFS on unweighted sparse graphs which is $O(n + m)$ time, it takes $O(l(n + m))$ time complexity. The benefit of using a graph embedding is that a feedforward neural network could respond to a distance query between two nodes u, v very fast independent of the graph size, so it takes $O(1)$ time. Therefore the proposed method for finding shortest path distances from a starting node u to all other nodes takes $O(n)$ time complexity.

2.2.2 Auto Computed Neural Network (ACNN) for Shortest Path Problem

In [5], a model of the neural network called Auto-wave-competed Neural Network (ACNN) has been proposed for the SP problem.

According to Figure 1. Neuron model of ACNN is composed of three components, which are the minimum selector, the auto-wave generator and the threshold updater.

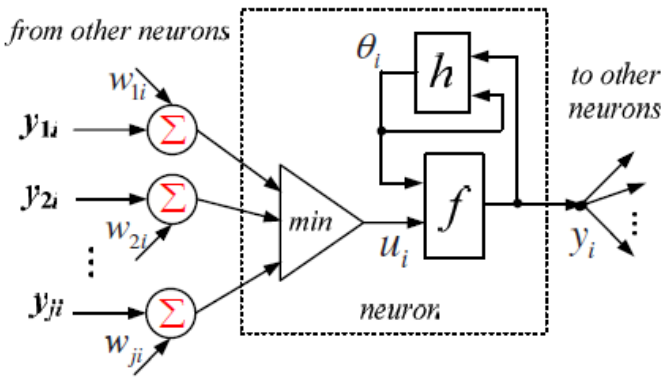


Figure 1. Neuron model of ACNN [5].

The following equations describe The ACNN neuron:

$$Z_i(t) = \{j \mid W_{ji} \neq \infty \ \& \ \ y_j(t-1) > 0\}$$ *Equation 1*

$$u_i(t) = \begin{cases} 0 & Z_i(t) = \emptyset \\ \min\limits_{j \in Z_i(t)} (y_j(t-1) + w_{ji}) & \text{otherwise} \end{cases}$$ *Equation 2*

$$y_i(t) = f[u_i(t), \theta_i(t-1)] = \begin{cases} u_i(t) & u_i(t) < \theta_i(t-1) \\ 0 & \text{otherwise} \end{cases}$$ *Equation 3*

$$\theta_i(t) = h[y_i(t), \theta_i(t-1)] = \begin{cases} \theta_i(t-1) & y_i(t) = 0 \\ y_i(t) & \text{otherwise} \end{cases}$$ *Equation 4*

Where i represents the index of neuron, t is the iteration number. $u_i(t)$ is the Internal activity, $\theta_i(t)$ is the threshold and $y_i(t)$ is the output of neuron i at time t. w_{ji}: The connection weight between neuron i and neuron j. $Z_i(t)$ is the set of neurons that fired to neuron i at time t and is reachable.

An ACNN isomorphic to weighted graph G should be constructed when applied to the SP problem; which means every node in the graph corresponds to a single network neuron i, and w_{ij} is weight of the edge(i, j) in the graph (see Figure 2).

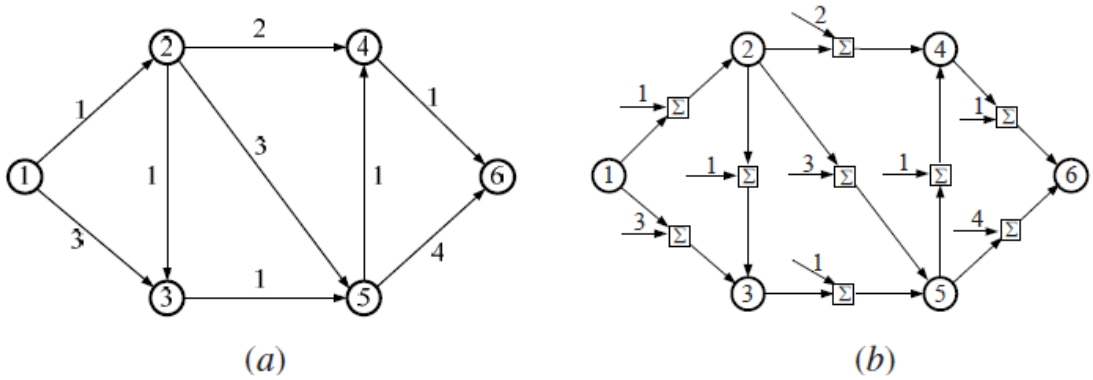


Figure 2. ACNN topology for SP problem. (a) A weighted digraph: it shows nodes and weighted edges. (b) The ACNN model for the graph's SP problem, the squares with "∑" inside are the summers on the corresponding links [5].

The neurons are initialized with infinite threshold and zero-internal-activity. Fire the source neuron to operate the network, and the firing will cause some propagating auto waves across the entire network. If the moving distance of an auto wave is the shortest, it will be decreased on the threshold $\theta_i(t)$ while going through the neuron i. The neurons are gradually decreasing their thresholds until the network stops.

Once the network ends, the threshold is equal to the distance from the source neuron to the neuron i of the shortest path.

Complexity

This algorithm's calculation focuses on the threshold update step. Suppose m is the average number of adjacent nodes, and n represents the number of nodes in the network the algorithm computes and compares $n(m+1)M$ paths in each algorithmic loop. So the computational complexity of the proposed algorithm is polynomial complexity which takes $O(n^2(m+1)M)$ time complexity.

2.2.3. Stochastic Shortest Path-based Q-learning (SSPQL)

Reinforcement learning (RL) has been commonly used as a method for autonomous robots by communicating with their environment to learn pairs of state-action. However, most RL approaches are slow in convergence when in practical applications an optimum policy is derived. In [6] in order to solve this problem, a stochastic shortest path-based Q-learning (SSPQL) is suggested, this method is combining a stochastic shortest path-finding method with Q-learning which is a popular model-free RL method.

Q-learning

The Q-learning algorithm is defined as a simple value iteration equation as follow:

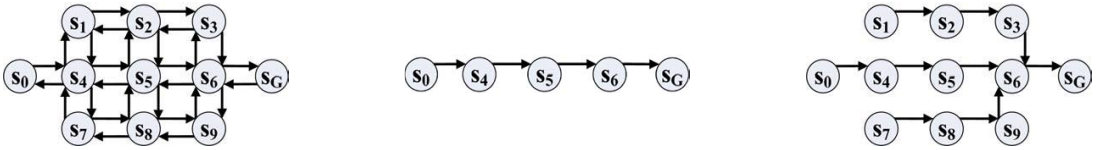
$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a' \epsilon A} Q(s',a') - Q(s,a) \right] \qquad \textit{Equation 5}$$

Where Q(s, a) denoted as an action-value function over an action a and state s. r is the reward value, α is the learning rate, and γ is the discount factor. In addition, A is a series of actions and s' is the state followed in a state s by an action a'.

Stochastic shortest path-finding method

In the SSPQL, to resolve model-free Q learning's slow learning speed, through using an internal state-transition model, the author obtains optimum local State-action pairs. Then by increasing the Q-value corresponding for each pair of state-actions, these optimal local State-action pairs are given a higher selection probability

Figure 3(a) shows a model of state-transition learnt from experience. Figure 3(b) represents the shortest single-pair path from the initial state, s_0, to the target state, s_G. A^* is a heuristic search method used for this type of problem. According to Figure 3(c), the shortest paths from all states to a single goal state are found in order to increase the probability of all relevant state actions in the model.



(a) An example of a state transition model. *(b) A shortest path from s_0 to s_G.* *(c) Shortest paths from all states to a goal state.*

Figure 3. An example of shortest path finding [6].

The definition of the expected cost of a state-transition from state s to s' is as:

$$EC(s,s') = weight(s,s')e^{-\lambda P(s'|s)} \qquad \textit{Equation 6}$$

Where $weight(s,s')$ illustrated true cost function of state-transition from s to s' and λ is a parameter which reflects the probability of a state-transition.

The set of edges that make up the SSP can be obtained after applying the Dijkstra's algorithm. To combine SSP with Q-learning, an action-value feature based on SSP was required, which is denoted as $QSSP(s,a)$:

$$Q_{SSP}(s,a) = \begin{cases} e^{-H(s,a)} & if\ s\ and\ s'\epsilon SSP, a = arg\max\limits_{a'\epsilon A} P(s'|s,a') \\ 0 & else \end{cases}$$

That $H(s,a)$ is the action entropy.

Integrating SSP-finding method with Q-learning

SSPQL algorithm learns and uses state-transition model simultaneously through combining the Q-learning method and the SSP method. Briefly, action-selection in SSPQL is based on a Linear Q-value combination, SSP-value, and the exploration bonus value:

$$a = \underset{a'\epsilon A}{\operatorname{argmax}} \left[W_Q Q(s',a') + W_{SSP_t} Q_{SSP(s',a')} + W_{EXP_t} Q_{EXP}(s',a') + \varepsilon \right] \qquad \textit{Equation 7}$$

$Q(s',a')$, $Q_{SSP}(s',a')$, and $Q_{EXP}(s',a')$ provide different characteristics with respect to the learning performance. Convergence to optimality can be ensured by using $Q(s',a')$. $Q_{SSP}(s',a')$ can increase the convergence speed. $Q_{EXP}(s',a')$ can push an agent to unexplored states. In [6], a simple counter-based exploration bonus is described as:

$$Q_{EXP}(s,a) = \frac{1}{(N(s,a)+1)^{\eta}} \qquad \textit{Equation 8}$$

Where $N(s,a)$ is the action frequency of a in state s, and η is the Q_{EXP}'s weighting parameter. As it gives higher priority values to actions that lead to unexplored states, and lower priority values to a regularly visited state, an exploration bonus reveals a property very different from other action-value functions.

Computational complexity

SSPQL's computational complexity is determined by considering the difficulty of finding shortest paths and estimating QSSP to achieve a target status. The most time-consuming process in SSP-finding method is the Dijkstra's algorithm; the computational complexity of this algorithm is given by $O(m + nlogn)$ with n nodes and m edges n when using the Fibonacci heap to implement the extract-min function as a priority queue. Therefore, in big-O notation, the complexity of the SSP finding method is $O(m + nlogn)$. the additional computational complexity of SSPQL is $O(SA + SlogS)$ for each episode; where S is the number of states, and A the number of actions available. This method's complexity is lower than many other model-based RL methods.

When integrating the two learning methods, both speed of learning and adaptability showed improvement compared with previous RL methods. Experimental results show that SSPQL's convergence speed exceeds both Q-learning and the sweeping method given priority.

2.3 Comparison to related work

Table 2 summarizes the different scenarios illustrated in the preceding subsections [7]:

Table 2. Comparing Ml_based algorithms for finding shortest path.

Authors	**ML model**	**Complexity**	**Approach**
Fatemeh Salehi et al [1]	Deep learning	O(n)	vector embedding
Jiyang Dong et al [5]	Artificial Neural Network	$O(n^2(m+1)M)$	auto-wave-competed neural network
Woo Young Kwon [6]	Reinforcement learning	$O(SA + SlogS)$	Q_Learning Stochastic shortest path
Us [2020]	Artificial Neural Network	O(n)	Graph neural network

In our thesis, we use a model which includes three components:

- An "Encoder" which independently encodes the edge, node features by using the neural networks.
- A "core" which performs N rounds of processing (message-passing) steps which is a well-known type of GNNs to propagate information between the nodes of the graph.
- A "Decoder" which independently encodes the edge, node attributes by using the neural networks.

The model is trained by supervised learning. Input graphs are procedurally generated, and output graphs have the same structure with the nodes and edges of the shortest path labeled.

After around 10000-15000 training iterations the model reaches near-perfect performance on graphs.

Then we focused on how to handle "wrong" solutions, i.e., solutions that are different from the shortest path (ground truth). Therefore we implemented a check mechanism to verify whether the solution provided by the model as output is a path. For those outputs which are paths do not reach the destination or that are not paths at all we do the following:

1- Build a graph using as link lengths (weights) the labels provided by the ML framework as output as follows: if the link is in the solution (label 1) then the link length is 0, if the link is not in the solution (label 0), then the link length is 1
2- Run the Dijkstra algorithm on such graph to find the shortest path between source and destination.

This way, all outputs are guaranteed to be paths connecting source and destination nodes. And it achieves a linear runtime complexity $O(n)$.

The above mentioned methods [5], [6] work well for small graphs, according the table we can see their complexity is polynomial or logarithmic.

The recent approach proposed in [1] utilize vector embeddings learnt by deep learning techniques to approximate the shortest paths distances in large graphs. The method achieves a linear runtime complexity and it can approximate distances with relatively low distortion error.

Chapter 3

Background

Machine-learning is a computer science sub field which is related to statistical computations; recently, ML techniques have been used in many application fields. It discusses the construction and analysis of algorithms in order to learn from and predict data. These methods work by constructing a model from inputs to predict or make decisions based on the data [8].

This chapter provides some information on ML, such as ML definitions, the algorithm categories which have been designed to address different problems. Then we define the shortest path problem and finally, the models and the framework which are used in our thesis are described.

3.1 Machine Learning

Machine learning was defined in 1959 by Arthur Samuel. He was the pioneer in ML and described it as [8]:

"Field of study that gives computers the ability to learn without being explicitly programmed."

Tom M. Mitchell offered a more formal description which was widely quoted:

"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

3.1.1 Types of Machine Learning tasks

Machine learning algorithms are usually divided into three groups, based on the type of the learning "signal" or "feedback" that a learning system can access [8]:

Supervised learning

The machine is provided by a "teacher" with example inputs and their desired outputs, and the goal is to learn a general rule that maps inputs to outputs.

Unsupervised learning

The learning algorithm is not given labels, so it is left alone to find structure in its data. Unsupervised learning may be a goal in itself (the discovery of secret patterns in the data).

Reinforcement learning

A program deals with a dynamic environment in which it has to accomplish a certain objective (such as driving a vehicle), without a teacher telling it explicitly whether or not it has come close to its goal.

Another classifying of machine learning methods arises when the desired output of a learned machine system is considered [8]:

In classification

Inputs are categorized in some classes, and the learner should generate a model in order to assign inputs which are unseen to these classes. It is generally dealt with in a supervised manner. An example of classification is spam filtering where the email messages are and the classes are “spam” and “not spam”.

In regression

It is a supervised problem as well, the outputs are continuous instead of discrete.

In clustering

A collection of inputs shall be broken down into classes. The groups are not known in advance, unlike in classification; this is typically an unsupervised task.

Density estimation

It finds the inputs distribution in a multidimensional space.

Dimensionality reduction

It simplifies inputs by mapping them into a space of lesser dimensions. Topic modeling is a related issue, where a list of documents in the human language is given to a program and it is charged with figuring out which documents cover similar issues.

3.2 Shortest Path Problem

One of the well-studied topics in computer science, especially in graph theory, is the shortest-path problem. The ideal shortest path is one with minimal length from source to destination. Work in the shortest-path algorithms has surged due to the various and diverse applications of the problem. Some of these applications are network routing protocols, route planning, traffic management, social network path finding, computer games, and transport systems [9].

3.3 Graph neural network (GNN)

For some applications the information is expressed naturally by graphs. Traditional methods handle graphical data structures through a preprocessing step that converts the graphs into a set

of nodes. However, important topological information may be lost in this way, and the results obtained can rely heavily on the stage of preprocessing [10].

In the last few years, there has been an increase in interest in Graph Neural Network (GNN) approaches to graph learning representation. GNNs perform a scheme of recursive neighborhood aggregation of information (or message passing), in which every node aggregates attribute vectors of its neighbors to determine its new feature vector. A node is represented by its transformed feature vector after k iterations of aggregation. It collects the structural information within the node's k-hop neighborhood.

In the next sub sections, we present the graph networks framework which is proposed in [7].

3.3.1 Graph network (GN) block

GN framework defines a graph as a 3-tuple G= (u; V; E). The u represents a global feature; a node is denoted as v_i which is the node attribute and the V is the set of nodes V=$\{v_i\}_{i=1:N^v}$. edge attribute is denoted as e_k and The E=$\{(e_k, r_k, s_k)\}_{k=1:N^e}$ indicates the set of edges. Note that s_k and r_k denote the sender and receiver nodes for edge k, respectively. To be more precise, some terminology provided in [7] is presented:

Directed: it means edges of one-way, from the node \sender to the node \receiver.

Attribute: it is the properties which can be encoded as a vector.

Global Attribute: an attribute at a graph level.

Internal structure of a GN block

The GN block includes three functions for updating and three functions for aggregation which are denoted as $\emptyset$ and ρ [7].

Multi-Layer Perceptron (MLP) is used for implementation of Update Functions ($\emptyset$): (Noted below to show that these different functions have different parameters)

$$e'_k = \emptyset^e(e_k, v_{r_k}, v_{s_k}, u) = MLP_e([e_k, v_{r_k}, v_{s_k}, u]) \quad \textit{Equation 9}$$

$$v'_i = \emptyset^v(\bar{e}'_i, v_i, u) = MLP_v([\bar{e}'_i, v_i, u]) \quad \textit{Equation 10}$$

$$u' = \emptyset^u(\bar{e}', \bar{v}', u) = MLP_u(\bar{e}', \bar{v}', u) \quad \textit{Equation 11}$$

The Aggregation Functions (ρ) are implemented trough elementwise summation:

$$\bar{e}_k = \rho^{e\to v}(E'_i) = \sum_{\{k:r_k=i\}} e'_k \quad \textit{Equation 12}$$

$$\bar{e}' = \rho^{e\to u}(E') = \sum_i v'_i \quad \textit{Equation 13}$$

$$\bar{v}' = \rho^{v\to u}(V') = \sum_k \bar{e}'_k \qquad \textit{Equation 14}$$

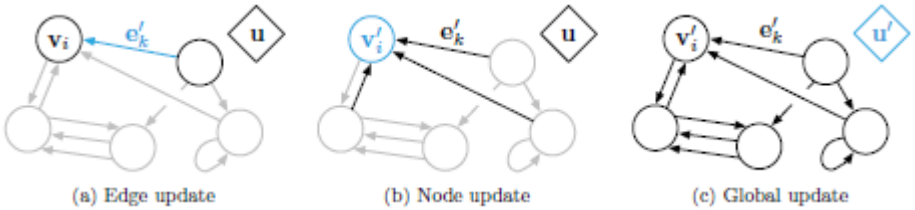
Where $E_i{}' = \left\{(\mathrm{e''}_k, r_k, s_k)\right\}_{k=1:N^e}$, $E' = \cup_i E'_i = \left\{(\mathrm{e'}_k, r_k, s_k)\right\}_{k=1:N^e}$ and $V' = \{v'_i\}_{i=1:N^v}$.

The $\emptyset^e$ is mapped in all edges to measure updates per edge, the $\emptyset^v$ is mapped to all nodes to determine updates per node, and the $\emptyset^u$ is used as a global update.

Each aggregation function takes a set as an input and reduces it to a single element representing aggregated information. These functions should be invariant to permutations of their inputs and variable number of arguments shall be allowed (e.g. summation, mean, maximum).

Computational steps within a GN block

Once a graph, G, is supplied as an input to a GN block, the computations proceed from the edge, to the node, to the global variables. Figure 4 provides a representation of which graph elements are involved in each of those calculations. Algorithm 1 indicates the steps of computation in the GN block.



(a) Edge update (b) Node update (c) Global update

Figure 4. Updates in the GN block. Blue represents the item being updated, and black indicates other elements involved in the updating [7].

Algorithm 1: computation steps in the block GN [7].

function GRAPHNETWORK(E, V, $\mathbf{u}$)
  for $k \in \{1 \dots N^e\}$ **do**
    $\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$ ▷ 1. Compute updated edge attributes
  end for
  for $i \in \{1 \dots N^n\}$ **do**
    let $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,\, k=1:N^e}$
    $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\to v}(E'_i)$ ▷ 2. Aggregate edge attributes per node
    $\mathbf{v}'_i \leftarrow \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$ ▷ 3. Compute updated node attributes
  end for
  let $V' = \{\mathbf{v}'\}_{i=1:N^v}$
  let $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\to u}(E')$ ▷ 4. Aggregate edge attributes globally
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\to u}(V')$ ▷ 5. Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$ ▷ 6. Compute updated global attribute
  return $(E', V', \mathbf{u}')$
end function

The following subsection describes the computation steps in the GN block:

1. $\emptyset^e$ with arguments $(\mathrm{e}_k, v_{r_k}, v_{s_k}, \mathrm{u})$ is applied per edge, and it returns e'_k. The resulting set of per-edge outputs for each node i is $E_i' = \{(\mathrm{e}'_k, r_k, s_k)\}_{r_k,=i, k=1:N^e}$. E' represents the set of all outputs per-edge which is $E' = U_i E'_i = \{(\mathrm{e}'_k, r_k, s_k)\}_{k=1:N^e}$.
2. $p^{e\to v}$ is applied to E'_i, it aggregates the edge features which is updated for edges that project to node i, and the output is $\bar{e}'_i$ used in the next node update stage.
3. $\emptyset^v$ is applied to every node i, in order to update node features, then it returns v'_i. The per-node outputs resulting set is: $V' = \{v'_i\}_{i=1:N^v}$.
4. $p^{e\to u}$ aggregates all edge updates, into $\bar{e}'$by applying to E', which will then be used in the global update of the next step.
5. $p^{v\to u}$ is applied to V', so aggregates all node attributes which is updated, into $\bar{v}'$, then it will be used in the next step's global update.
6. Finally $\emptyset^u$ is applied once to every graph, and computes an update the global features,u'.

Note, the order is not enforced strictly: the update functions can be reversed to go from global level update, to per-node, to per-edge updates.

3.3.2 Message-Passing Neural Network (MPNN)

MPNN are a well-known class of GNNs that use an iterative message-passing algorithm for the propagation of information between graph nodes.

Each node k receives messages from all its neighbor's nodes $N(k)$, in a message-passing step. Messages are created by a message function $m(.)$ which is applied to the node-pair's hidden state in the graph, then they are aggregated by an aggregation function such as an elementwise summation. Then, an update function $u(.)$ is implemented to provide a new hidden state for each node. Figure 5 and the following equations describe the Message-Passing method [11].

$$M_k^{t+1} = \sum_{i \in N(k)} m(h_k^t, h_i^t) \qquad \textit{Equation 15}$$

$$h_k^{t+1} = u(h_k^t, M_k^{t+1}) \qquad \textit{Equation 16}$$

Message functions $m(\cdot)$ and update function $u(\cdot)$ can be implemented by MLP. After several iterations, the update function outputs $u(.)$ are aggregated by an elementwise summation then transfers the result to a readout function $R(.)$. MLP is used for implementation of this function, as well.

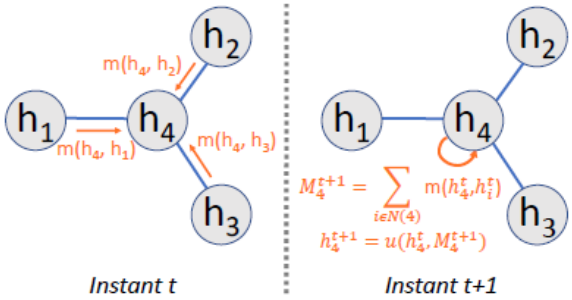


Figure 5. Message-passing for 4 nodes in the graph in tow steps of time [11].

MPNN can be transformed into GN formalism naturally [7]:

- The $m(.)$, message function plays the role of $\emptyset^e$ in the GN block, but doesn't use u as data.
- The aggregation function which is elementwise summation is used for the GN's $\rho^{e\to v}$ and $\rho^{e\to u}$.
- The $u(.)$, update function plays the role of $\emptyset^v$ in the GN block.
- The $R(.)$ readout function, which is implemented by MLP, plays the function of $\emptyset^u$ in the GN block, but does not use u or E' as input.

Chapter 4

Machine Learning Framework for Routing in Telecommunication Network

We provide a comprehensive overview of the Machine Learning Framework in this chapter that is used to do our experiments. We'll then describe the scalable enhancement of the Machine Learning framework and its computational complexity.

Finally, we give experiments design. The implementation of the ML framework is written in Python programming language, using libraries such as Networkx, Tensorflow.

4.1 Network Routing Workflow (Train Phase)

In our thesis we used the ML framework which is proposed in [12]. The framework is trained by a variety of graphs with a random number of nodes and edges. Then it is tested by samples of a test set which are independent of the training set.

The overall view of the ML framework illustrated in Figure 6.

As the training method is supervised learning we need target graphs and input graphs. Target graphs are used in computing error and optimization process**.** Input graphs are used in the training process. According to the graph definition, the graph is illustrated by the node's attributes and edge's attributes.

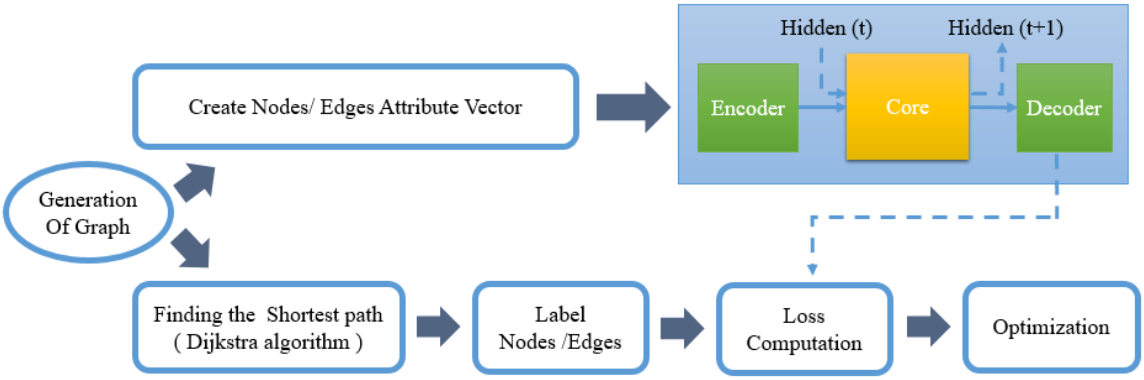


Figure 6. Workflow for network routing (Train Phase).

4.1.1 Graph Generation

To generate random graphs we used the method proposed in [12]. According to Figure 7, two graphs with the same number and location of nodes are combined in order to generate fully connected graph. The graphs are geographic threshold graphs, but with a minimum spanning tree algorithm added edges to ensure that all nodes are connected.

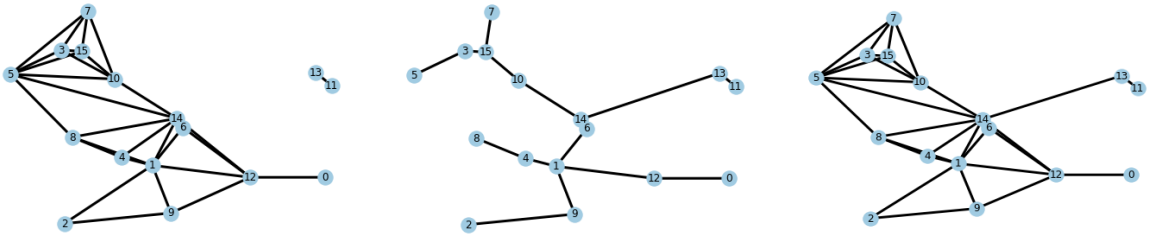


Figure 7. Proposed algorithm generated graph in [12]. (right) Geographic graph with separate nodes, (middle) minimum spanning tree, and (left) graph combined.

4.1.2 Input graph feature vector

Node's attributes in the input graphs consist of a vector of five elements which are as following:

- Node position: [x coordinate, y coordinate]
- Weight
- Start node
- End node

Weight is an exponential random value that specifies if there is a connection between the nodes or not.

Start and End are binary values that indicate if the node is source or destination node of the path or not.

Edge's attribute in the input graph is a vector of one element which is:

- Distance

It shows the distance between two edge-belonging nodes.

4.1.3 Target graph feature vector

The Dijkstra algorithm is used to mark the nodes and edges of the shortest path problem. In Dijkstra algorithm distance is used as weight to find the shortest path. It marks each node and edge as bellow:

The nodes and edge which are in the Dijkstra's shortest path marks to [0, 1]. We call them nodes of solution (T), and edges of solutions (T). The nodes and edge which are not in the Dijkstra's shortest path labels to [1, 0]. We name these nodes and edges non-solution (F).
Figure 8 shows an example graph which are labeled.

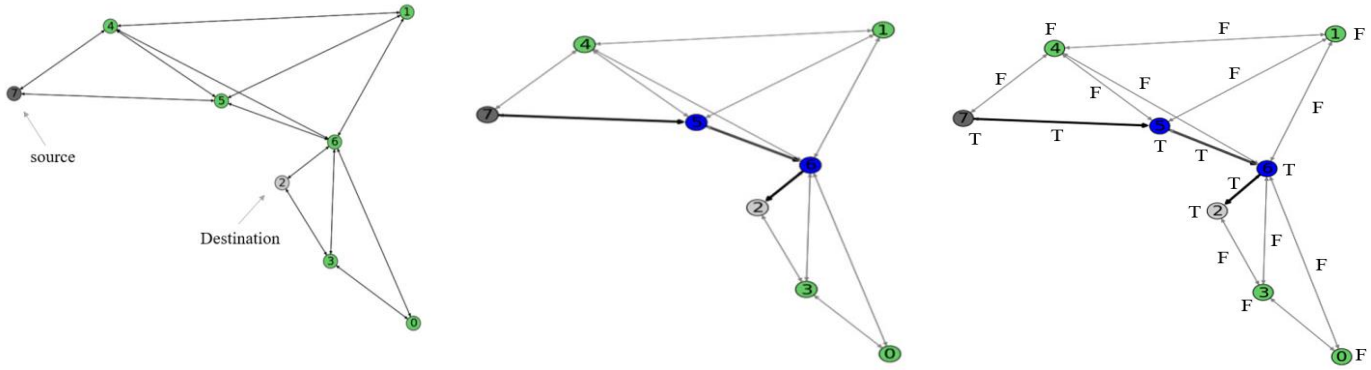


Figure 8. (Right graph) The original graph, (Middle graph) the routed path graph, (Left graph) the labeled graph.

The Table 3 shows the labels of each node in the example graph. These labels used as feature vectors.

Table 3. node labels in the example graph

Nodes ID	Label	Attribute vector
0	False	[1,0]
1	False	[1,0]
2	True	[0,1]
3	False	[1,0]
4	False	[1,0]
5	True	[0,1]
6	True	[0,1]
7	True	[0,1]

4.1.4 Training Model

According to the Figure 9 the model which we used includes 3 parts [12]:

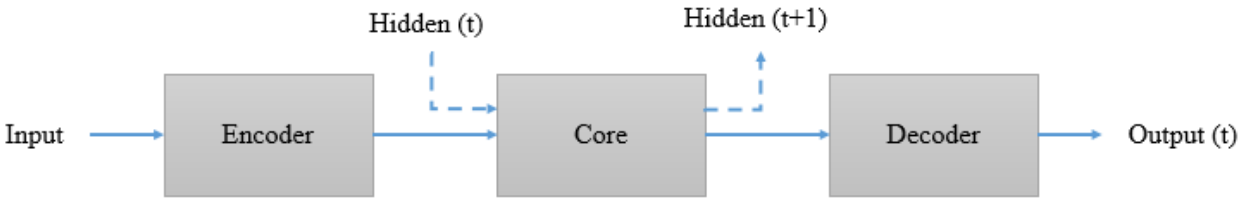


Figure 9. Workflow for Training Model.

- Encoder: It encodes the edge vector and node vector independently. That means 2 separate MLPs independently map edge and node vectors to a 32-elements vector (see Figure 10).

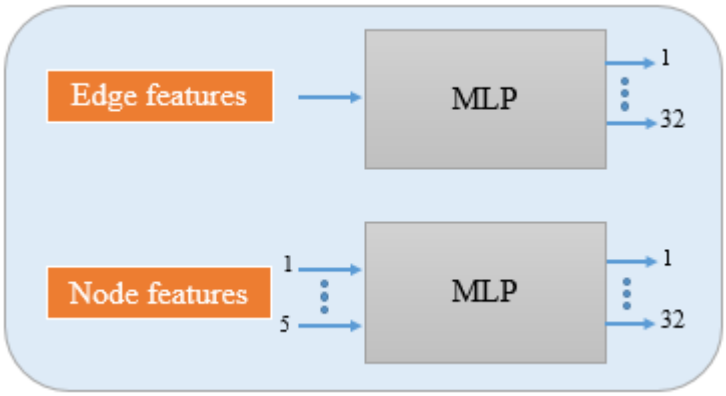


Figure 10. Encoder scheme.

- Core: It executes massage passing steps N times. The core input is a concatenation of the core's prior output and the encoder output. This section will be illustrated in detail.
- Decoder: It decodes the edge vector and the node vector independently on each message-passing step. So, similarly to the encoder we have 2 separate MLPs that independently map edge and node vectors to a 32-elements vector (see Figure 11).

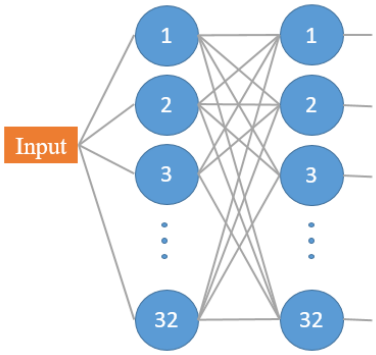


Figure 11. A fully connected MLP with two layers and 32 neurons in each layers

4.1.5 Core

The Core section is the Graph Network block which is proposed in [7]. It is consist of 3 blocks (see Figure 12):

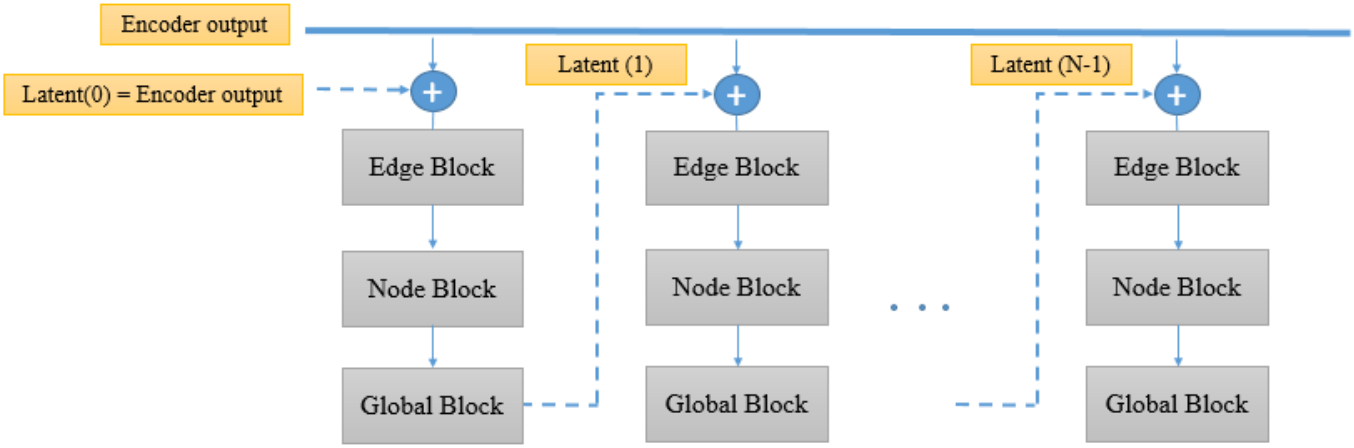


Figure 12. Core scheme.

- Edge Block: in this block features of each edge will be updated using MLP. The input of MLP is a concatenation of the preceding edge features, and the adjacent node features which called the sender node and receiver node.
- Node Block: the block updates every node features using a MLP. At first, it aggregated the features of adjacent edges. Then a concatenation of aggregated adjacent edge and previous node feature used as the input to MLP.
- Global Block: this block updates global features. Using the MLP a concatenation of aggregated edge features and aggregated node features will be updated.

4.1.6 ML output

Nodes and edges are labeled in the input graph at the MPNN end. As Figure 13 shows the two element vectors with different values are represented and associated to label. By interpreting the labels we classify them and create 2 output classes. These classes are defined by the following equation:

$$class_{out_i} = argmax\left(Label_{MPNN_i}\right) \qquad \textit{Equation 17}$$

Nodes ID	Node Labels at the MPNN Output	ArgMax (Label)
0	[6.0417661 -2.23854092]	0
1	[3.35104672 -2.1593072]	0
2	[-4.89505307 3.83677993]	1
3	[5.43695683 -2.35722765]	0
4	[0.72520728 -0.88029439]	0
5	[-0.90006829 0.30690665]	1
6	[-4.14749179 3.46277]	1
7	[-4.68965861 3.7333144]	1

Figure 13. Example of node labeling.

4.1.7 Loss Computation and optimization

Softmax Cross-Entropy is used as loss function. Softmax function $S(.)$ takes a vector with C-dimension, z as input. It returns a real vector with C-dimension, y which is between 0 and 1.

$$y_c = S(z)_c = \frac{e^{Z_c}}{\sum_{d=1}^{c} e^{Z_d}} \qquad forc = 1 \cdots C \qquad \text{Equation 18}$$

The softmax function can be used in the output layer of a neural network, which is represented graphically by a layer with C neurons.

Using the ADAM optimizer from Tensorflow, the optimization method for the training step was performed to minimize Softmax cross-entropy as the classification loss function. The extension of the stochastic gradient descent is the Adam optimization algorithm. Since it quickly achieves good results, it is a commonly adopted in deep learning algorithms.

Our ML system executes 15,000 iterations of training steps with an adaptive learning rate of 0.02 starting rate (in each step 32 training graphs are used as batch size input). Also, it performs 10 rounds of message passing. The satisfactory generalization results will be obtained after a number of trial and error experiments, by checking the classification error in the train, test and validation sets.

4.2 Network Routing Workflow (Test Phase)

When the ML framework has learned and the datasets are created, into the trained network will be fed by a graph during the test process, and the decoder will output the labels for all nodes and edges in that network. We must ensure that the solution is a connected and constitutes a

valid path, because the nodes and edges are labeled independently. Figure 14 shows the workflow of the test phase

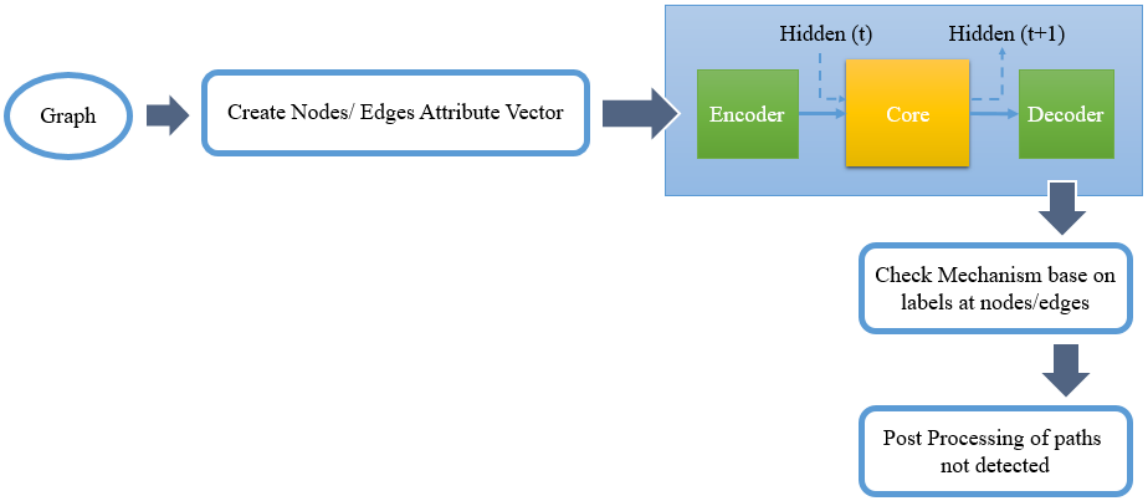


Figure 14. Workflow for network routing (Test Phase).

4.2.1 Check mechanism

We may have extra nodes or edges which are identified as solutions in the node and edge labels (see Figure 15), or even sometimes we may have a path which is not continuous or is disconnected. So we are proposing an independent check mechanism over the path based on node and edge labels to check if the solution is a path or not. In the following we'll describe the mechanism's pseudocode.

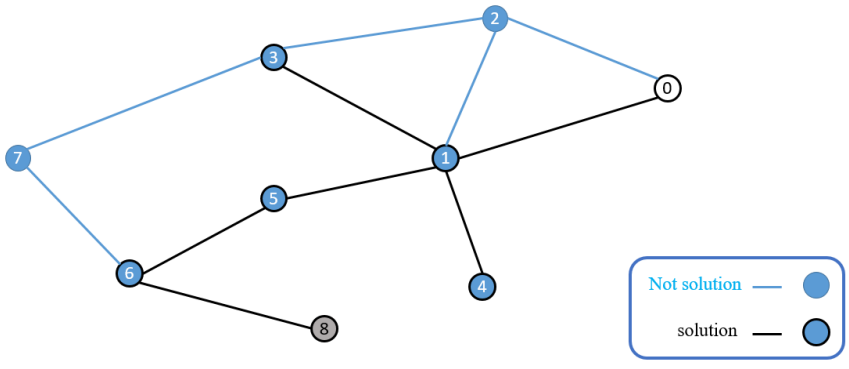


Figure 15. Nodes 3 and 4 are labeled incorrectly, But the path has been obtained.

$G = (V, E)$ represents a graph in which V is the set of graph nodes and E is the set of edges. As we know, the aim of the shortest path problem is to find a path between the source (v_{src}) and destination (v_{dst}) nodes. In the check mechanism we start from v_{src} and traverse over the nodes which are marked as the solution, then determine the number of possible outgoing nodes at each step. If the number of outgoing nodes is zero and the current node is not the destination node, the current node is assumed to be a dead-end node and removed from the search domain. When the number of outgoing nodes is equal to 1, implies that it only has one option to pass, so the next step is to choose that node. If the number of outgoing nodes is more than one, it

means that there is a branch, so the node is added to the Branch list and one of the available nodes is chosen as the next node. The Branch list helps the algorithm in order to go back in selecting the right way in case of a dead end or wrong path selected.

The algorithm only stops when the route reaches the destination node or discontinuity is detected.

$G = (V, E)$
$Branch\ list = []$
$Current\ node = v_{src}$
$While\ path = not\ Found$
    $Num\ nodes, node\ id = OutgoingNodes(G, Current\ node, node\ Labels)$
    $If\ (Num\ nodes = 1)$
        $Next\ node = node\ id$
    $Else\ if (Num\ nodes = 0)$
        $Delete(Current\ position)$
        $Next\ node = Previous\ branch$
        If (branch list = empty)
            break
    $Else\ if (Num\ nodes > 1)$
        $Branch\_list += current\_node$
        $Next\ node = random\_node\ from\ available\ nodes$
    $If (Next\ node = v_{des})$
        $Path = Found$
    Current node = $Next\ node$
end

The same is achieved for the edge marks, but the algorithm crosses the edges of the solution to get the v_{dst} . Detected paths using both methods will then combine to obtain a path with greater certainty.

Figure 16 (a) shows an example which illustrates the path is detected by tracking node labels, while by following edge labels the path will not be achieved. Similarly Figure 16 (b) indicates an example that the path is detected by tracking edge labels, while by following node labels it cannot reach the destination.

$G = (V, E)$
$Branch\ list = []$
$Current\ node = v_{src}$
$While\ path = not\ Found$

$Num\ edges, node\ id = OutgoingEdges(G, Current\ node, edge\ Labels)$

$If\ (Num\ edges = 1)$

$Next\ node = node\ id$

$Else\ if (Num\ edges = 0)$

$Delete(Current\ position)$

$Next\ node = Previous\ branch$

If (branch list = empty)

break

$Else\ if (Num\ edges > 1)$

$Branch\_list \mathrel{+}= current\_node$

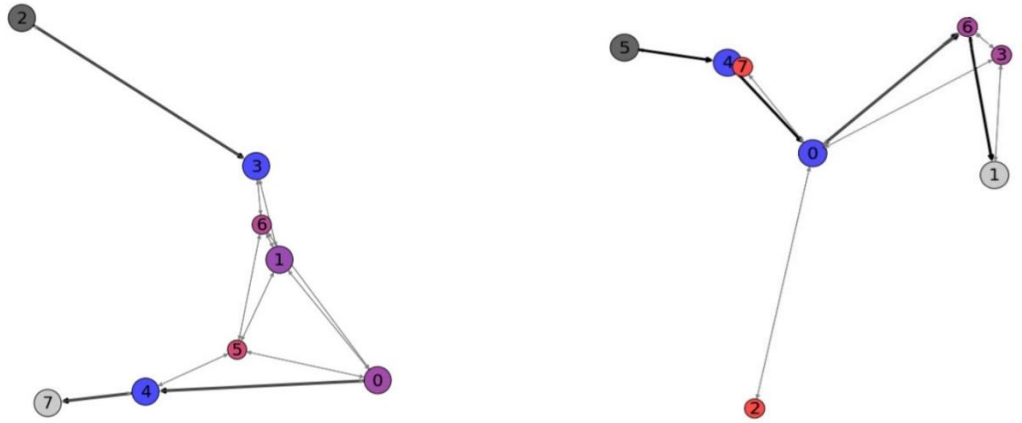
$Next\ node = random\_edge\ from\ available\ edges$

$If (Next\ node = v_{des})$

$Path = Found$

Current node = $Next\ node$

end



(a) (b)

Figure 16. detecting path with node labels or edge lables. (large nodes are solution labeled nodes, thicker edges are solution labeled edges)

4.2.2 Path recovery

According to the Figure 17 after applying the check mechanism the path can be set in the most graphs, but for some cases the search mechanism algorithms cannot find a path, so we proposed the Path Recovery algorithm to find a path.

1. It Builds a graph using the labels given by ML framework as output then the link length consider as weights in this way:
    - If the link is in the solution (label 1) then the link length is 0.
    - If the link is not in the solution (label 0), then the link length is 1.

2. It Runs the Dijkstra algorithm on the created graph in order to find the shortest path based on new weights between source and destination.

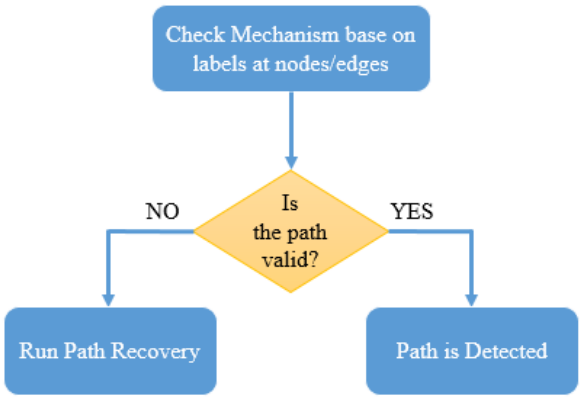


Figure 17. path recovery scheme.

4.3 Performance Assessment

To evaluate the performance of the model which has used, we used the metrics involved in the measurement of the accuracy of the routing (node and edge classification problem) [12], these metrics are defined as bellow:

- **STR**: training fraction examples solved correctly

This metric measures the "float" fraction of training graphs which are correctly solved the path and defined as bellow:

$$STR = \frac{Number\ of\ correctly\ solved\ examples\ in\ training\ dataset}{Number\ of\ total\ examples\ in\ training\ dataset} \qquad \textit{Equation 19}$$

- **SGE**: test fraction examples solved correctly

This metric measures the "float" fraction of generalization graphs which are correctly solved the path and defined as:

$$SGE = \frac{Number\ of\ correctly\ solved\ examples\ in\ test\ dataset}{Number\ of\ total\ examples\ in\ test\ dataset} \qquad \textit{Equation 20}$$

- **CTR**: training fraction of nodes/edges labeled correctly

This metric measures the "float" fraction of training graphs which are correctly labeled the Nodes/Edges and defined as bellow:

$$CTR = \frac{Number\ of\ correctly\ labeled\ nodes/edges\ in\ training\ dataset}{Total\ Number\ of\ nodes/edges\ in\ training\ dataset} \qquad \textit{Equation 21}$$

❖ **CGE**: test fraction of nodes/edges labeled correctly

This metric measures the "float" fraction of generalization graphs which are correctly labeled the Nodes/Edges and defined as:

$$CGE = \frac{Number\ of\ correctly\ labeled\ nodes/edges\ in\ test\ dataset}{Total\ Number\ of\ nodes/edges\ in\ test\ dataset} \qquad \textit{Equation 22}$$

4.4 Complexity analysis

Here, we analyze the framework time complexity which it includes a large amount of the algorithm's computational time. So we will compute the complexity in Encoder, Core, and Decoder.

The main operations in Neural Network consist of matrix multiplication in each layer. We know, for $A_{i\times j} * B_{j\times k}$, the Time Complexity of matrix multiplication is $O\ (i \times j \times k)$ [13].

As we mentioned, the neural network which we used in the implementation of the algorithm, has 2 layers and each layer has 32 number of neurons.

So we do as follow for computing the complexity of node and edge embedding:

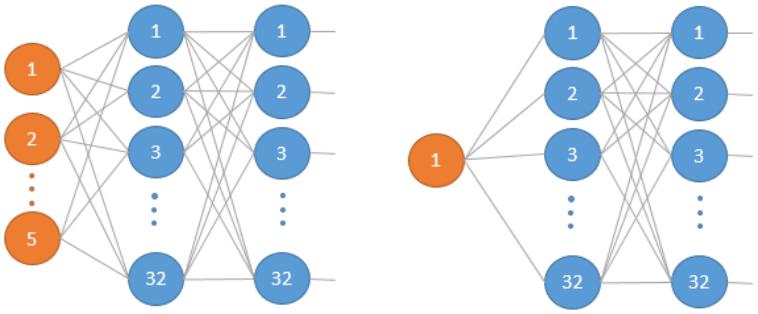


Figure 18. Neural network scheme.

As there are 3 layers, 2 weight matrixes is needed. We'll denote them by $W_{32\times5}$, $W_{32\times32}$, for node embedding and $W_{32\times1}$, $W_{32\times32}$ for edge embedding.

4.4.1 Node Embedding Complexity

In general, to propagate from layer i to layer j, we first do following multiplication:

$$Z_{j1} = W_{ji} \times X_{i1}$$

The time complexity of this operation is $O(j * i)$. Then the activation function is applied:

$$X_{j1} = f(Z_{j1})$$

Since the operation is element_wise the complexity would be $O(j \times 1)$. Therefore total complexity in propagating between 2 layers is:

$$O(j \times i + j) = O\big(j \times (i + 1)\big) = O(j \times i)$$

According to the Figure 18, each node represented as a five_element vector in the framework, so the time complexity of the node embedding for each node, is computed as following:

propagation	Weight matrix	Time Complexity
First	$W_{32\times5}$	$O(32\times5)$
Second	$W_{32\times32}$	$O(32\times32)$

$$C\ (Node\ Embedding) = \ O\ (32\times5) + (32\times32) = O(32\times32)$$

4.4.2 Edge Embedding Complexity

The time complexity of the edge embedding for each edge is determined as below, According to the Figure 10.

propagation	Weight matrix	Time Complexity
First	$W_{32\times1}$	$O(32\times1)$
Second	$W_{32\times32}$	$O(32\times32)$

$$C\ (Edge\ Embedding) = \ O\ (32\times1) + (32\times32) = O(32\times32)$$

4.4.3 Encoder / Decoder Complexity

The encoder performs edge embedding for each edge and also node embedding for each node in the graph. Suppose we have N nodes and E edges in each graph, so the encoder time complexity will be:

$$C(Encoder) = \ N * \ C(Node\ Embedding\ ) + E * C(Edge\ Embedding)$$

$$C(Encoder) = \mathrm{O}\ ((N+E) * \ (32\times32))$$

Note that Decoder complexity is the same as encoder complexity.

$$C(Decoder) = \mathrm{O}\ ((N+E) * \ (32\times32))$$

4.4.4 Core Complexity

According to Figure 12, the core carries out N rounds of node block, edge block and global block. In each block features will be updated by following MLP.

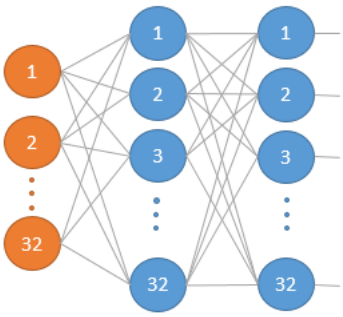


Figure 19. Neural network.

- Node block complexity:

  According to the Figure 19, the time complexity of feature updating for each node is $O(32 \times 32)$. So, the complexity of node block for N number of nodes would be $O(N \times 32 \times 32)$.

- Edge block complexity:

  Edge features updates using the MLP in Figure 12, it has $O(32 \times 32)$ for each edge. Se the edge block complexity for E number of edges is $O(E \times 32 \times 32)$.

- Global block complexity:

  Same as node block and edge block the complexity of global block is $O(32 \times 32)$.

The time complexity of core with respect to the performing 10 rounds of mentioned blocks in our implementation, would be:

$$C(Core) = N_{steps} * \big(C(Node\ block\ ) + C(Edge\ block) + C(Global\ block\ )\big)$$

$$C(Core) = \mathrm{O}\ (10 * (N + E + 1) * \ (32 \times 32)))$$

4.4.5 ML framework Complexity

The ML framework time complexity is summation of Encoder complexity, Decoder complexity and Core complexity:

$$C(ML\ framework) = \ C(Encoder\ ) + C(Core) + C(Decoder\ )$$

$$C(ML\ framework) = O\big((N + E) \times (12 \times 32 \times 32)\big) = O(C \times (E + N))$$

It shows the ML framework has a linear Run Time Complexity. So by this method finding the shortest path for the large networks has reasonable time complexity.

$$C(ML\ framework) = O(E + N)$$

Chapter 5

Numerical Assessment

This chapter offers a detailed description of the experiments performed in this work and shows the numerical evaluation for each.

We first explain how datasets are generated to perform experiments and provide statistical characteristics of the data. Then we present the environment and conditions of the simulations, and how they are applied.

Finally, the simulation results of the presented method are compared with the results of the benchmark and these results will be analyzed.

5.1 dataset Generation

In order to generate the graphs for our dataset, we consider two graphs with the same number and location of nodes that are combined to create a fully connected graph. These graphs are geographic threshold graphs and through a minimum spanning tree algorithm edges are added to ensure all nodes are connected to each other. Therefore the combined graph would be a fully connected graph for sure [12].

Using this method, we can create a different number of batch size graphs as our input set for the training and test phase.

5.2 Determination of hyper-parameters

In our framework we have different parameters and variables. These parameters are:

- Number of Nodes per training graph
- Number of Nodes per test graph
- Number of Message Passing Steps
- Number of Iterations
- Batch Size (Training/Test)
- Learning rate
- Adaptive Learning Rate

- ❖ Number of Layers
- ❖ Number of Neurons
- ❖ Time (Hour)

To find the best configuration for our framework we changed the parameters and evaluated the results. At each step, we change only one parameter so that the variation in the performance the ML framework can be measured by applying that parameter change. In all of these experiments, the number of fixed graph nodes is assumed, also the batch size and the number of iterations are fixed:

- Number of nodes per training graph sampled uniformly from the range (8, 16).
- Number of nodes per test graph sampled uniformly from the range (17, 33).
- Number of Iterations: 1000.
- Batch Size (Training/Test): 32/100.

To measure the effectiveness of the parameters we use the SGE factor which measured the fraction of test graphs which are correctly solved the path.

5.2.1 Change the Learning Rate

We fixed all other parameters in our framework and just changed the value of the learning rate. Table 4 shows the results, as we can see by setting the lowest Learning Rate, we can get the best result. So for the following tests, we have used fixed Learning Rate 0.001.

Table 4: Change the Learning Rate

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	1e-1	2	16	1.04h	0.8333
10000	10	1e-2	2	16	0.79h	0.9397
10000	10	1e-3	2	16	0.67h	0.9433

5.2.2 Change the Message Passing Steps

To find a reasonable number of massage passing steps for the ML framework, we just changed the number of massage passing steps. As Table 5 shows, by setting the highest number of massage passing steps we can get the best result but it takes more time. So for the following tests, we have used a fixed number of message passing steps equal to10.

Table 5: Change the message passing steps

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	8	1e-3	2	16	0.58h	0.9233
10000	10	1e-3	2	16	0.67h	0.9433
10000	12	1e-3	2	16	0.73h	0.9437
10000	14	1e-3	2	16	0.79h	0.9509

5.2.3 Change the Number of Neurons

As Table 6 shows by setting the highest number of neurons in each layer we can get the best result but it takes more time to train. But the improvement obtained by setting the number of neurons 64 in comparison with the number of neurons 32 is negligible moreover, it took much more time. So it seems training with 32 neurons is more reasonable.

Table 6: Change the number of neurons

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	1e-3	2	16	0.67h	0.9433
10000	10	1e-3	2	32	0.9h	0.9570
10000	10	1e-3	2	64	1.7h	0.9627

Again, we set number of layers 3 and we performed previous experiments, similar results were achieved:

Table 7: Change the number of neurons in 3 layers

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	1e-3	3	16	0.78h	0.9085
10000	10	1e-3	3	32	0.99h	0.9310
10000	10	1e-3	3	64	1.89h	0.9528

5.2.4 Change the Number of Layers

Here we analyze the impact of changing the number of layers. Table 8 shows with the same number of neurons if we have less number of layers the result is better and also it takes less time. So it seems training with 32 neurons and 2 layers is more reasonable.

Table 8: Change the number of layers

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	1e-3	2	16	0.67h	0.9433
10000	10	1e-3	3	16	0.78h	0.9085
10000	10	1e-3	2	32	0.9h	0.9570
10000	10	1e-3	3	32	0.99h	0.9310
10000	10	1e-3	2	64	1.7h	0.9627
10000	10	1e-3	3	64	1.89h	0.9528

5.2.5 Using an Adaptive Learning Rate

We repeated the previous experiments, according to following tables we found out that the results improved by using an Adaptive Learning Rate than using a Fixed Learning Rate. Also, training with 2 layers is better than 3 layers with the same number of neurons.

Table 9: Fixed and Adaptive LR with 2 - 3 hidden layers and 16 neurons.

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	Fix	2	16	0.67h	0.9433
10000	10	Adaptive	2	16	0.64h	0.9588
10000	10	Fix	3	16	0.78h	0.9085
10000	10	Adaptive	3	16	0.68h	0.9245

Table 10: Fixed and Adaptive LR with 2 - 3 hidden layers and 32 neurons.

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	Fix	2	32	0.9h	0.9570
10000	10	Adaptive	2	32	0.93h	0.9840
10000	10	Fix	3	32	0.99h	0.9310
10000	10	Adaptive	3	32	1h	0.9632

Table 11: Fixed and Adaptive LR with 2 - 3 hidden layers and 64 neurons.

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	Fix	2	64	1.7h	0.9627
10000	10	Adaptive	2	64	1.6h	0.9707
10000	10	Fix	3	64	1.89h	0.9528
10000	10	Adaptive	3	64	1.78h	0.9573

So far, we found the results improved by using adoptive LR and training with 2 Layers. Table 12 indicates training with 32 neurons is more reasonable whit respect to the training time.

Table 12: using adoptive LR and change the number of neurons.

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons	Time (Hours)	SGE
10000	10	Adaptive	2	16	0.64h	0.9588
10000	10	Adaptive	2	32	0.93h	0.9840
10000	10	Adaptive	2	64	1.6h	0.9707

Finally, we have chosen the following structure for the ML framework:

Table 13: Final Structure for ML framework.

Iterations	Message Passing steps	Learning Rate	Number of Layers	Number of Neurons
10000	10	Adaptive	2	32

5.3 Results

The ML framework is trained with network topologies of different sizes and it is tested to evaluate the scalability and the capability of our approach to generalize. The training dataset includes 8~200-node network topologies and the testing dataset includes network topologies of the 16~400-nodes. The experiments have been performed, using of the cluster of the High Performance Computing (HPC) [14] center of the Politecnico di Torino, running python scripts. The following are the parameters and results of the experiments performed:

5.3.1 Experiment (1)

Training phase

I. In this experiment, first we considered the same number of nodes for training and validation graphs, in order to see how well is the ML framework for unseen test graphs having the same size of the training graphs.
   - Number of nodes per training graph sampled uniformly from the range (8, 16).
   - Training batch size: 32.
   - Number of nodes per validation graph sampled uniformly from the range (8, 16).
   - Validation batch size: 100.
   - Figure 20 represents the training graphs statistics.

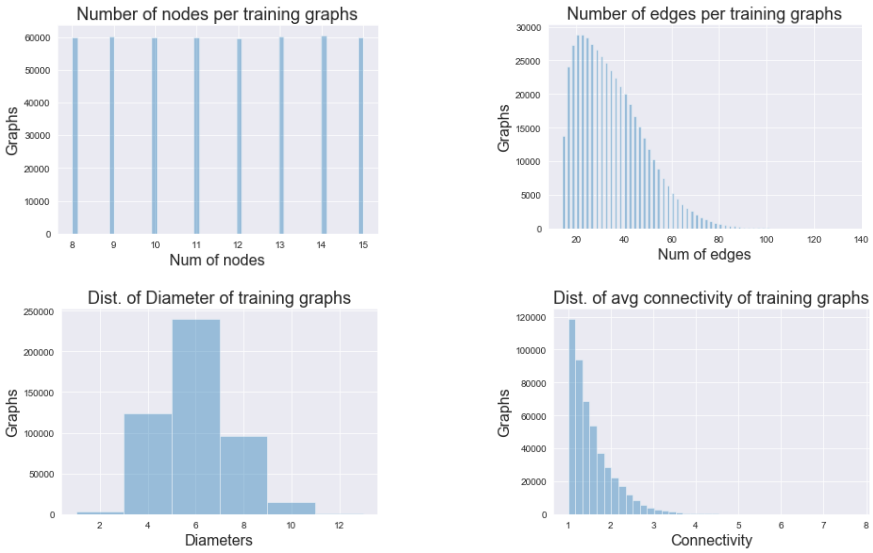


Figure 20. Testing graphs statistics.

Figure 21 shows the accuracy of the ML framework in terms of the CTR metric: as we can see the network was able to correctly label the vast majority of the edges and nodes for both training datasets and validation datasets. The accuracy is very close to 1.

Also, in Figure 22 we measure the STR metric, and it indicates that it can correctly identify the shortest paths with 98% accuracy.

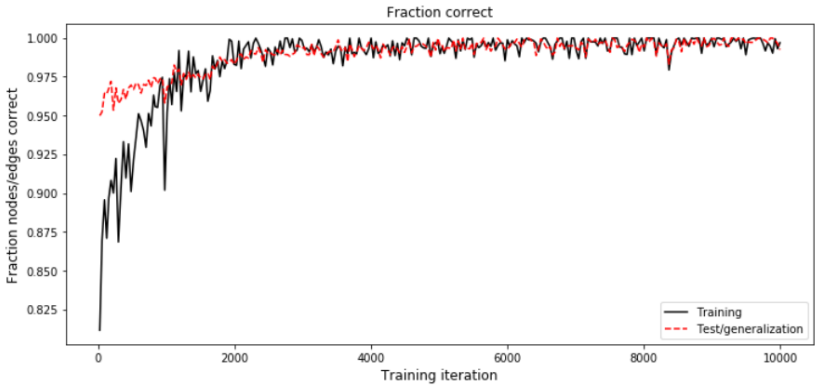


Figure 21. Performance (CTR) of the ML framework for experiment (1-I).

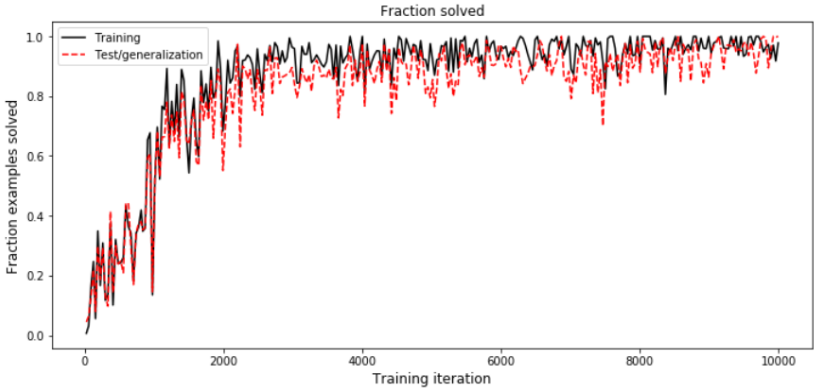


Figure 22. Performance (STR) of the ML framework for experiment (1-I).

II. In the next step, we considered validation graphs with the twice number of nodes of the training graphs to evaluate the generalizability of the model.
   - Number of nodes per training graph sampled uniformly from the range (8, 16).
   - Number of nodes per validation graph sampled uniformly from the range (16, 32).

The CTR metric is shown in Figure 23, it reaches almost the perfect performance in nodes and edges labeling.

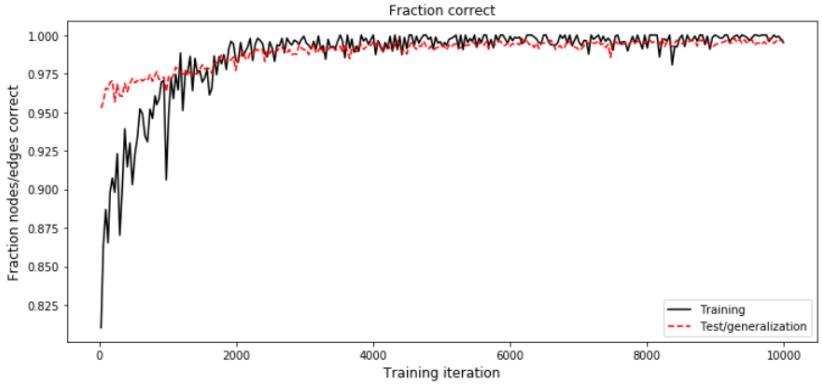


Figure 23. Performance (CTR) of ML framework for experiment (1-II)

According to Figure 24, the model achieves high accuracy for graphs with a scale twice greater than the training graphs after 1000 iterations. It achieves STR of 0.94.

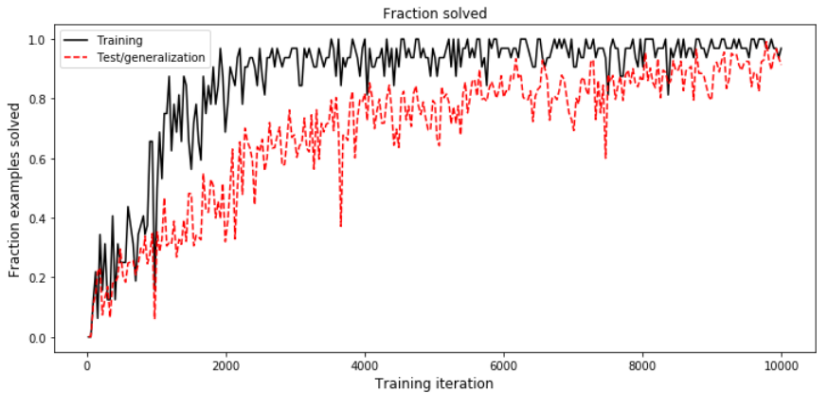


Figure 24. Performance (STR) of ML framework for experiment (1-II)

Test Phase

Finally once the model is trained we tested it with 1000 samples of graphs with nodes in the range (16-32). Figure 25 represents the testing graphs statistics.

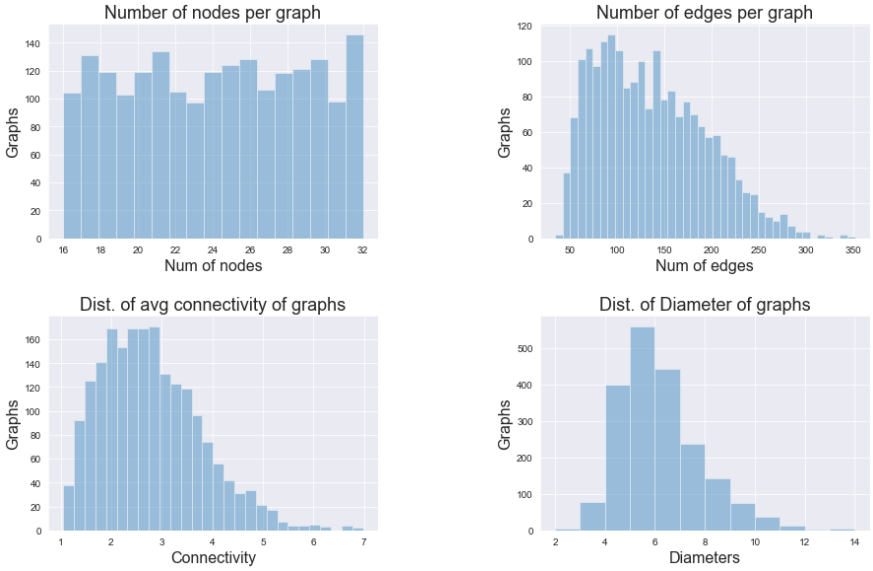


Figure 25. Testing graphs statistics.

The model can correctly solve the path with 0.9 accuracy so the shortest path is found. For the remaining graphs we use the check mechanism. In the check mechanism:

1- In some graphs the path can be detected by tracking the node labels.
2- In some graphs the path can be detected by tracking the edge labels.
3- In some graphs the path is not detected so it is obtained using the path recovery algorithm.

Therefore the algorithm can guarantee to find the route.

5.3.2 Experiment (2)

Training phase

I. Using the same number of nodes for training and validation graphs.

Table 14. Specifications for experiment 2-I

training graph size	validation graph size	Training batch size	Validation batch size
(40-50)	(40-50)	32	100

According to CTR metric in Figure 26 we can say, the model achieves results very close to 1 in nodes and edges labeling for both the training datasets and the validation datasets. Similarity to the previous experiment, the STR metric also closely approaches 1 when identifying the shortest path (see Figure 27).

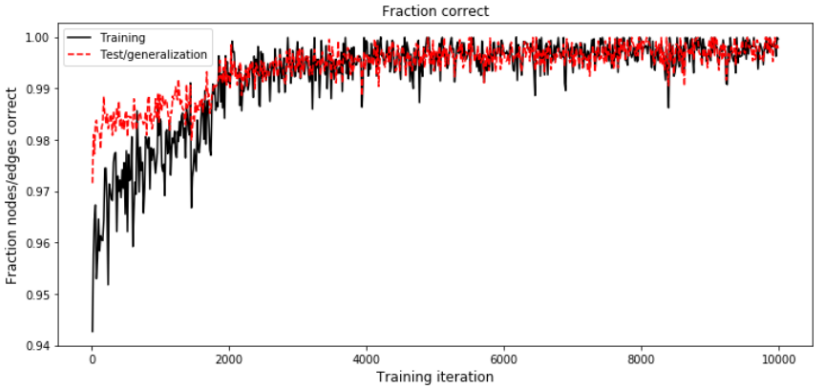


Figure 26. Performance (CTR) of ML framework for experiment (2-I)

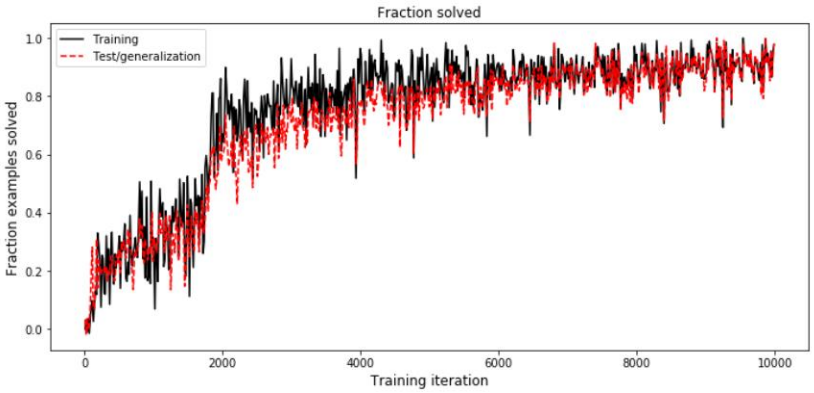


Figure 27.Performance (STR) of ML framework for experiment (2-I)

II. Using Validation graphs with the training graphs having twice the number of nodes to evaluate generalizability of model.

Table 15. Specifications for experiment 2-II

training graph size	validation graph size	Training batch size	Validation batch size
(40-50)	(80-100)	32	100

Figure 28 shows the model achieves almost the optimal value for graphs with a scale twice greater than the training graphs. Also, STR reaches 0.93 (Figure 29).

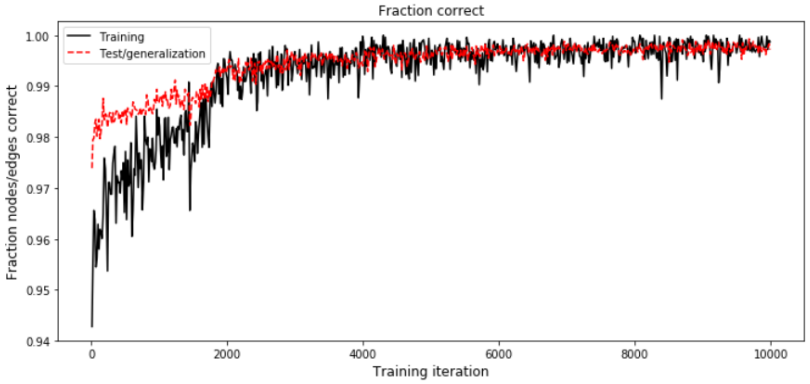


Figure 28. Performance (CTR) of ML framework for experiment (2-II)

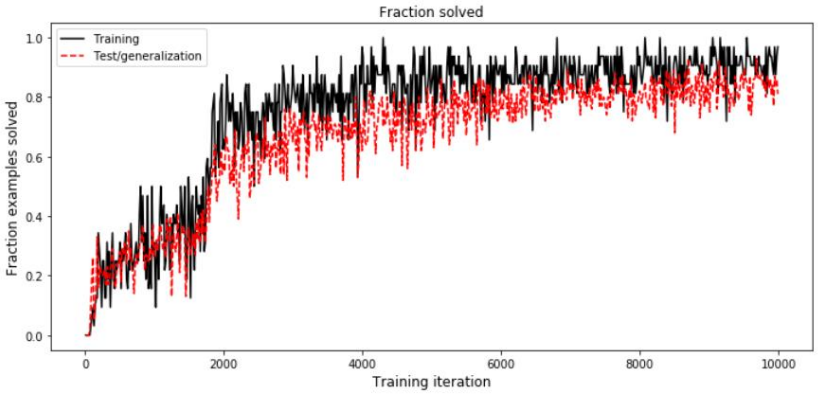


Figure 29.Performance (STR) of ML framework for experiment (2-II)

Test Phase

Once the model has been trained we tested it with 1000 graph samples with nodes in the range (80-90).

Same as before, the model performs near perfectly and acceptable results with twice the size of the training graphs (0.93 accuracy) can be obtained. After applying check mechanism and path recovery, the algorithm can guarantee to always find a feasible route.

5.3.3 Experiment (3)

Training phase

I. Using the same number of nodes to graphs for training and validation.

Table 16. Specifications for experiment 3-I

training graph size	validation graph size	Training batch size	Validation batch size
(150-200)	(150-200)	32	100

Similar to previous experiments Figure 30, Figure 31 shows that the model performs optimally for the same size training and validation datasets.

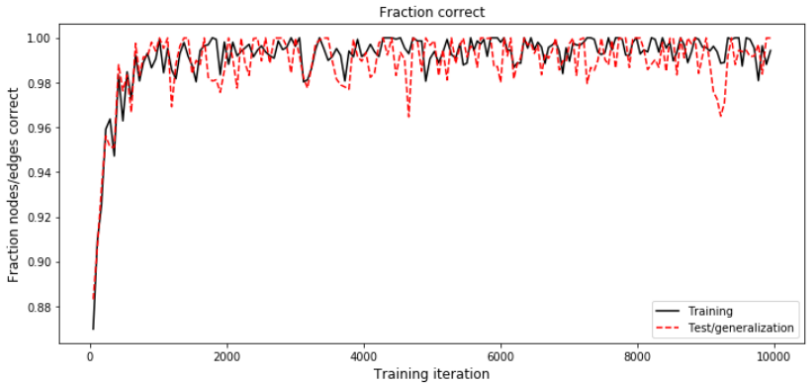


Figure 30. Performance (CTR) of ML framework for experiment (3-I)

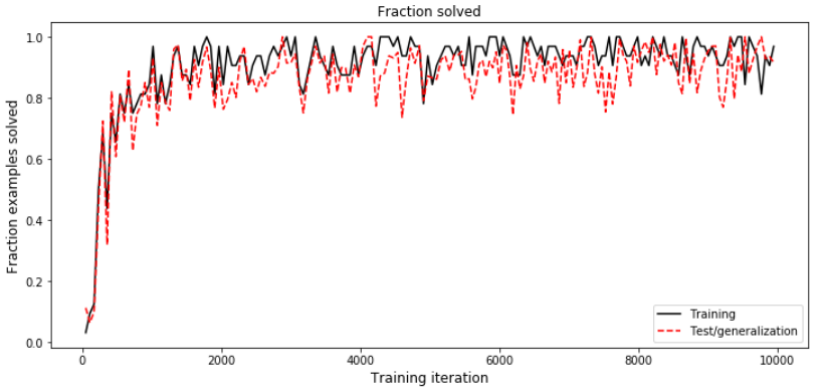


Figure 31. Performance (STR) of ML framework for experiment (3-I)

II. Using Validation graphs with the twice number of nodes of the training graphs to evaluate the generalizability of the model.

Table 17. Specifications for experiment 2-II

training graph size	validation graph size	Training batch size	Validation batch size
(150-200)	(300-400)	32	100

Figure 32 shows that the model can label nodes/edges perfectly for graphs twice as large as the training graphs. With respect to the STR in Figure 33 we can say the model obtains acceptable results with twice the size of the training graphs, it can solve the route correctly with STR=0.85.

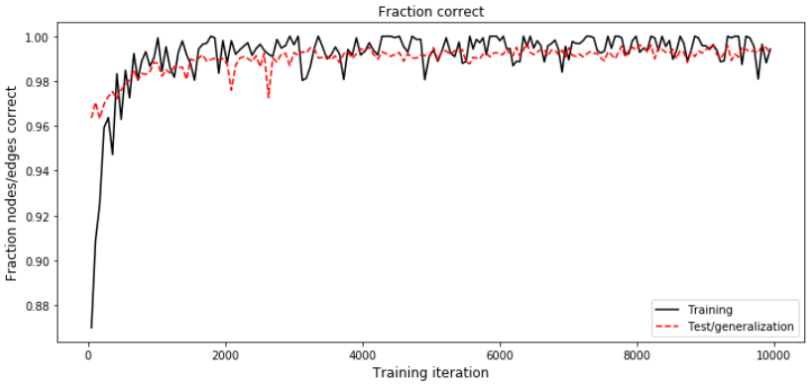


Figure 32. Performance (CTR) of ML framework for experiment (3-II)

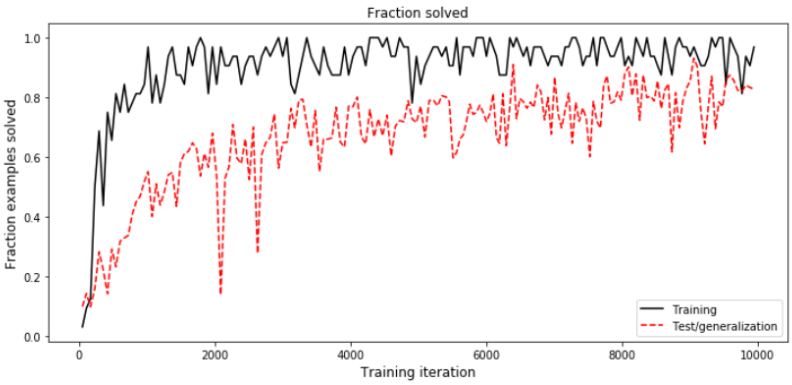


Figure 33. Performance (STR) of ML framework for experiment (3-II)

Test Phase

After training of the model we tested it with 1000 samples of graphs with nodes in the range (350-400).

Just like the previous experiments our model will guarantee that the route will be found in all graph sizes.

5.3.4 Complexity Evaluation

We have performed experiments to test the scalability of the algorithm so far, and the algorithm is scalable, as observed. In this section, we look at the algorithm in terms of time and compare it with the popular Dijkstra algorithm which can quickly provide the shortest path.

As we discussed in the section 4.4, the proposed algorithm has a linear time complexity. Figure 34 is proof of this claim. As can be seen, the blue curve indicates the time needed to perform calculations for graphs of different sizes in the range (10-500). This curve coincides with the curve theoretically obtained.

As we know, the computational complexity of the Dijkstra algorithm is $O(\,|E| + |V|\;log\;|V|)$. Figure 35 shows the correspondence between the time taken to implement Dijkstra and the theory.

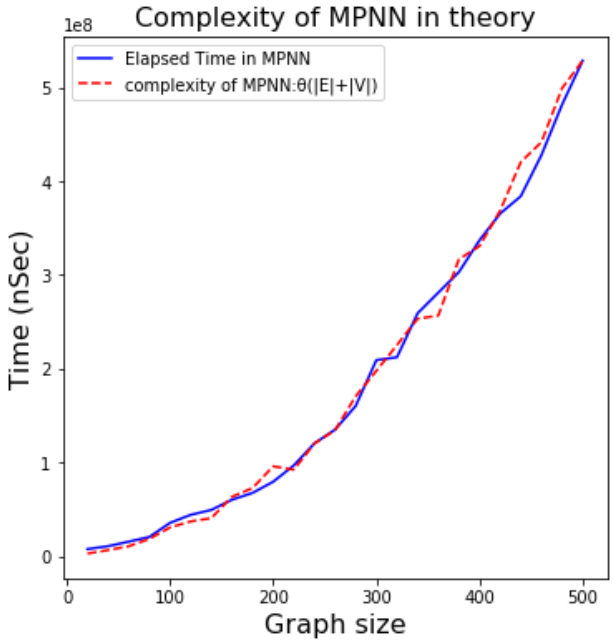


Figure 34. Comparison of algorithm's runtime complexity presented in theory and practice.

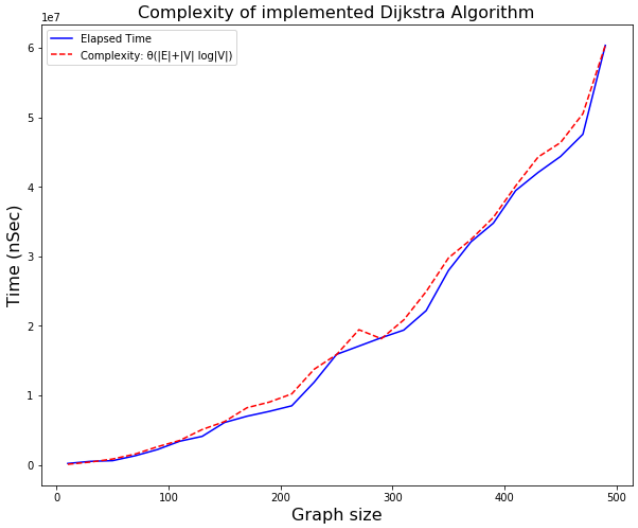


Figure 35. Comparison of Dijkstra's runtime complexity in theory and practice.

In Figure 36 a comparison of the proposed algorithm with the Dijkstra algorithm is given. For small networks the Dijkstra algorithm works well and quickly. But the larger the network, the longer it takes to find the shortest path, while our algorithm for larger networks has acceptable and fast runtime complexity.

Therefore our algorithm finds the path faster for very large networks than Dijkstra.

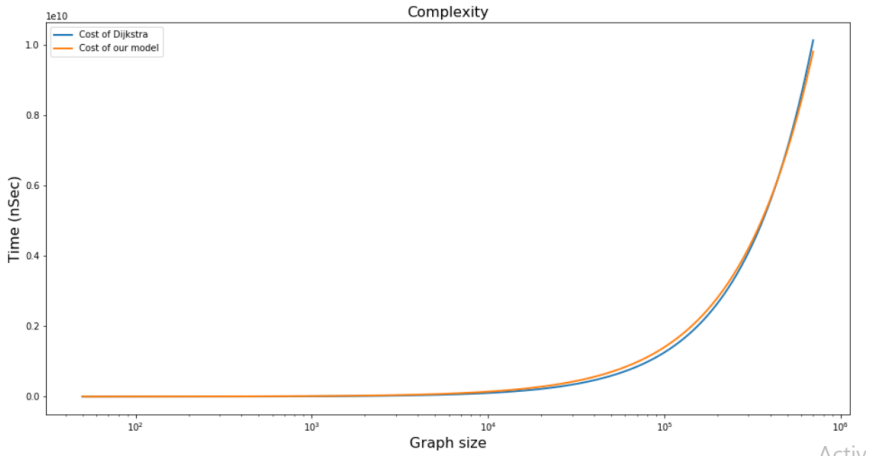


Figure 36. Comparison of the proposed algorithm complexity with the Dijkstra algorithm complexity.

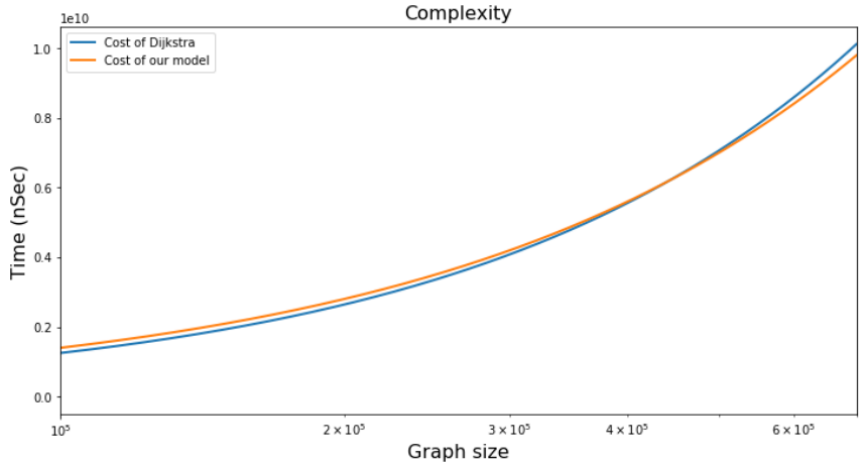


Figure 37.To see the intersection of the two graphs and the better performance of the proposed algorithm, we zoom Figure 36.

Chapter 6

Conclusion

In this study, a method based on Machine Learning for routing in large telecommunication networks was proposed. The proposed method uses GNN which is a new family of neural networks designed to work with graph-structured data. GNNs execute a scheme that aggregates neighborhood information, recursively, in which every node aggregates its neighbors attribute vectors to determine their new feature vector.

In our work, the node embedding is calculated using MPNN which aggregating node-neighbor information through non-linear transformation and aggregation functions. So MPNN applies an iterative message-passing algorithm to propagate information between graph nodes. Then, nodes and edges are labeled in the input graph at the MPNN end. Finally, by propagating the error regarding the Dijkstra solution, we try to minimize the error and teach the neural network how to correctly label the graph elements on the network. Then, we can guarantee to find the path using a post-processing approach. We assessed the performance of our ML framework and Dijkstra algorithm to find the shortest possible route, Achieved results show that our ML framework is scalable which means the model obtains nearly optimal and reasonable performance with test graphs twice as large as the training graphs. Additionally, the proposed algorithm has a linear temporal complexity whereas the Dijkstra algorithm, a well-known and fast criterion, has a complexity of $N.\log N$ order. In other words, the larger the network would be, the longer computational is used by Dijkstra algorithm. While the proposed algorithm was faster for large networks and offers better performance. As the results show, this approach can be used for routing in large telecommunication networks and get the shortest possible path with good speed and accuracy.

Bibliography

[1] Fatemeh Salehi Rizi,Joerg Schloetterer,Michael Granitzer, "Shortest Path Distance Approximation using Deep learning Techniques," *arXiv,* vol. 1, 12 Feb 2020.

[2] Hong Qua, Simon X. Yang, Zhang Yi, Xiaobin Wang, "A novel neural network method for shortest path tree computation," *Elsevier,* June 2012.

[3] Fernando Michel Tavera, "Dijkstra's Shortest Path Algorithm," no. University of Mexico (UNAM).

[4] William J. Cook, William H. Cunningham,William R. Pulleyblank Alexander, Schrijver, "Combinatorial Optimization," *ISBN 0-471-55894-X,* September 18, 1997.

[5] Jiyang Dong, Junying Zhang, "Neural Network Based Algorithm for Multi-Constrained Shortest Path Problem," *Springer-Verlag Berlin Heidelberg,* p. 776–785, 2007.

[6] Woo Young Kwon, Il Hong Suh, and Sanghoon Lee, "SSPQL: Stochastic Shortest Path-based Q-learning," *International Journal of Control, Automation, and Systems,* pp. 328-338, 2011.

[7] Peter W. Battaglia, Jessica B. Hamrick, et al, "Relational inductive biases, deep learning, and graph networks," *arXiv,* vol. 3, October 2018.

[8] Introduction to Machine Learning, The Wikipedia Guide.

[9] Amgad Madkour, Walid G. Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, "A Survey of Shortest-Path Algorithms," May 8, 2017.

[10] Gori, M., Monfardini, G., and Scarselli, F., "A new model for learning in graph domains," *Proceedings of the International Joint Conference on Neural Networks (IJCNN),* vol. 2, pp. 729-734, 2005.

[11] Paul Almasan, José Suárez-Varela, et al, "Deep Reinforcement Learning meets Graph Neural Networks: exploring a routing optimization use case," *arXiv,* vol. 2, 14 Feb 2020.

[12] "https://github.com/deepmind/graph_nets," [Online].

[13] "https://ai.stackexchange.com/questions/5728/what-is-the-time-complexity-for-training-a-neural-network-using-back-propagation," [Online].

[14] "Computational resources provided by HPC@POLITO.," which is a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino., [Online]. Available: (http://hpc.polito.it).