Uppsala University

Project CS 2022

1DT054

Spreadnet

Authors:
ChanVuth Chea
Boli Gao
Jennifer Gross
Ishita Jaju
Haouyuan Li
Akanksha Makkar
Mishkuat Sabir
Paarth Sanhotra
Gaurav Singh
George Stoian
Sofia Afnaan Syed
Haodong Zhao

April 24, 2023

Contents

1 Introduction

The shortest path (SP) problem is something that we encounter every day. Things such as biking to the university or going to the grocery store, require us to give an approximate solution of how we would move from one place to another. While we all know that a straight line is the shortest way to go, as we delve into the realm of longer paths that go beyond what we can see, we start realizing that the intuition tools nature provided us with, become inadequate. As a result, finding the shortest path in more complex or abstract scenarios requires formal methods that can solve the problem with minimal information and operations. There can be other solutions that can be considered as Sub-optimal. A sub-optimal path is an alternative path that is still acceptable if we first define some accepted error margin.

In many scenarios, we can look at the shortest path problem as a graph where we want to find the series of points that we can visit in order to find the traversal of the minimal number of nodes or nodes weights. In this case, there are a number of algorithms that have been proven to return solutions within acceptable time limits. While it is true that the shortest path problem inside a graph has plenty of fast enough algorithms that manage to fulfill our everyday needs, it is still worth occasionally trying novel approaches that tackle the problem from a different perspective.

The great advantage of the classic methods Dijkstra [1], Bellman-Ford [2], A* [3], and so on is that they are algorithms that we can fully understand and properly analyze. With an algorithm we can tell if it will solve a problem or not, we know what it is made for and what it does. On the other hand, if we talk about the limitations of these classic algorithms, Dijkstra does a blind search and wastes alot of useful resources and time. Also, Dijkstra can handle positive weights only. Bellman-Ford can handle positive and negative weights but for negative weights, it can handle directed graphs only. While A* has high complexity in terms of implementation and execution, and comsumes a lot of memory. Another algorithm that we will later define in more depth is that of Graph Neural Networks (GNN) [4], [5]. Deep learning is capable of processing structured data, such as speech, images, and natural language, which has led to significant improvements in processing speech, images, and natural language. The problem jumps in when we do not have structured data like social networks, knowledge graphs, complex file systems, economic networks, chemical molecules, and worldwide webs. Typical Neural Networks (NN) like Convolutional Neural Networks (CNN) [6], and Recurrent Neural Networks (RNN) [7] are not capable of handling unstructured data input properly because these networks stack the feature of nodes by specific order [8]. In this kind of unstructured problem GNN can outperform to extract information from the unstructured data. However, instead of relying on a table of information extracted from the graph or on a distance approximation function, it uses neural networks to extract the knowledge from graphs it has seen. This knowledge is general, and it tries to encompass the shortest paths seen in all the graphs as opposed to the table of values constructed by Dijkstra which is particular to a specific graph.

Using GNNs to solve the shortest path problem is something worth exploring because GNNs are basically Deep Neural Networks (DNNs) and DNNs are data driven method. GNNs can only perform better than the classical algorithms if training and testing data have same distribution otherwise they will not perform well. In the real world there often are patterns in the data, and where we have patterns, machine learning techniques have been excellent at uncovering and making use of them. Additionally, given that our very

first and still most used method of solving the shortest path problem is applying natural intuition coupled with the knowledge contained in our brains, using an apparatus, the neural network, built by us in our *mental* image has a certain beauty. Finding methods to compare GNNs to classical algorithms is a valuable toolkit to identify when it makes sense to replace a classical method with a GNN.

2 Our Contributions

We have created Python scripts that allow the user to generate datasets, train, and experiment with different GNN models implemented with PyTorch[1]. A keynote would be that during the training, we extract a variety of metrics that are described in Section 6.2. In the experiments we have implemented, some metrics that we consider to be better at verifying the quality of a prediction have been described in Section 6.2.1. Additionally, based on the results and behavior we have seen, we have implemented some hybrid approaches that combine the GNNs with classical methods, in this case, Dijkstra's algorithm, in order to guarantee that a solution is always given. Theoretically, in some situations, the GNNs can reduce the amount of work required to be done by Dijkstra. However, a complexity analysis and memory usage of the GNN algorithm coupled with Dijkstra is required to see if there are any actual benefits.

Other contributions, guiding the GNNs to learn the patterns exhibited by the shortest path problem is paramount to the quality of the predictions. This line of thinking where the focus is on ensuring that the model learns what the shortest path is something that was not present in what we considered to be our baseline paper [4].

All the models and variations we have attempted aim to close the gap between what the model learns and the definition of a path. More precisely, the models need to focus on predicting paths not just the attributes of the nodes. The difficulty lies in the fact that the attributes are predicted at the element level, be it a node or an edge. While the problem requires a series of nodes and edges. Our attempts at fixing this difference between the models learning knowledge at the elementary level while being required to make predictions that need a higher perspective that returns a path are our main theoretical contributions.

Our main attempts can be split into 3 categories:

- **GNN algorithm changes**, where we modify the base algorithm presented in [4]. This includes an adaptive number of message-passing steps described in Section 4.3.1 and other variations of knowledge aggregation that result in different models as seen in Section 4.3.1. Some results from our models are listed below.
  - Message Passing Neural Network (MPNN) and Adaptive MPNN work well. Adaptive MPNN has better accuracy but it needs more resources to train.
  - Graph Attention Network (GAT) works fine when it is trained on a smaller dataset. But in-path accuracy gets decreasing when the size of the dataset increases. The performance of GAT couldn't beat the MPNN and Adaptive MPNN.
  - Graph Convolutional Network (GCN) works fine on small datasets, but not on medium and big datasets. Its accuracy is close to GAT but it uses more memory resources and is slower than GAT.

[1] `https://github.com/pytorch/pytorch`

  - Gated Graph Sequence Neural Network (GGSNN) achieved high accuracy on sequence datasets with more than 500 epochs but the performance validation accuracy is very poor.

- **Loss function** modifications, given that the loss function is the one that affects the way the model is performing using a loss function that takes into account the characteristics of a path seem promising. Those ideas have been described in Section 4.1.4.

  - The "weighted loss" implementation significantly increases the performance of the model while training with the MPNN algorithm on small and medium sized datasets.
  - The "connectivity favouring loss" decreases the performance while training.
  - The "Euclidean weighted loss" also decreases the performance while training.

- **Series Predicting GNN**, this approach deals with the problem of making individual node and edge predictions when the desired result is a series by simply predicting a series of nodes. While a promising approach, implementing it has proven to be quite difficult. The paper we based our implementation on [9] is still under review. Following are the reasons that prompted us to interrupt further research:

  - PyG[2] do not support graph to sequence models.
  - Lack of innovations in the required model and fundamental ideas that are being used in the research paper was borrowed from other papers.
  - Poor code maintenance and original paper using low version of TensorFlow framework.

3 Preliminaries

The flow of Section 3 will provide a base for the rest of the paper including an overview of the shortest path problem, Dijkstra's algorithm, and GNNs. Section 3.1 defines the shortest path problem as it is crucial to our research to remove ambiguity. Dijkstra's algorithm is explained in Section 3.2, which is significant to our project since we are using it to evaluate the GNNs. Section 3.3 will cover GNNs, which form the basis of our research.

[2] `https://github.com/tensorflow/tensorflow`

Notation	Description
G	A graph
$\mathbf{u}$	Global attribute
$\mathbf{V}$	Set of nodes
$\mathbf{E}$	Set of edges
v_i	Node's attribute with index i
N^v , N^e	Number of nodes, edges in the graph
e_k	Edge's attribute with index k
r_k , s_k	Receiver node with index k , Sender node with index k
e'_k	Updated edge with index k
ϕ^e, ϕ^v, ϕ^u	Update function of edge, node and, global attributes
v'_i	Updated node with index i
$\bar{e}', \bar{v}'$	Aggregated edge, aggregated node
$\rho^{e\to v}\rho^{e\to u}, \rho^{v\to u}$	Aggregation functions
$\mathbf{E}', \mathbf{V}'$	Set of all updated edges, set of all updated nodes
u'	Updated global attribute
$\mathbf{A}$	Adjacency matrix for edges
$\mathbf{D}$	Diagonal node degree matrix
$\mathbf{W}$	Learnable weight matrix
$\overrightarrow{\boldsymbol{a}}$	Learnable weight vector
LeakyReLU	The LeakyReLU Function
$\mathbf{T}$	Transpose matrix
$\|\|$	Concatenation operation
h	Feature vector
$X^{(k)}$	Intermediate node annotation at index k
$\mathrm{H}^{(k,T)}$	Output score at the output step k and propagation step T
$\mathrm{F}_o^{(k)}$	GGNNs unit to predict output sequence with output step k
p, n	Prediction for positive class, prediction for negative class
p', n'	Ground truth for positive class, ground truth for negative class
P	Power consumption

Table 1: Notation used in the paper

3.1 Defining the Shortest Path Problem

Graphs are an effective way of representing relationships between complex and non-linear objects. A graph G is a set of three attributes $(\mathbf{u}, \mathbf{V}, \mathbf{E})$, the $\mathbf{u}$ represents the global attribute, the $\mathbf{V} = \{v_i\}_{i=1:N^v}$ is the set of nodes (of cardinality N^v) where each v_i is the node's attribute at index i. While $\mathbf{E} = \{(e_k, r_k, s_k)\}_{k=1:N^e}$ represents the set of edges (with cardinality N^e), where each e_k is the edge's attribute with index k, r_k is the index of the receiver node, and s_k is the index of sender node. Typically, nodes in the graph represent objects or concepts while edges are the relationships between the nodes.

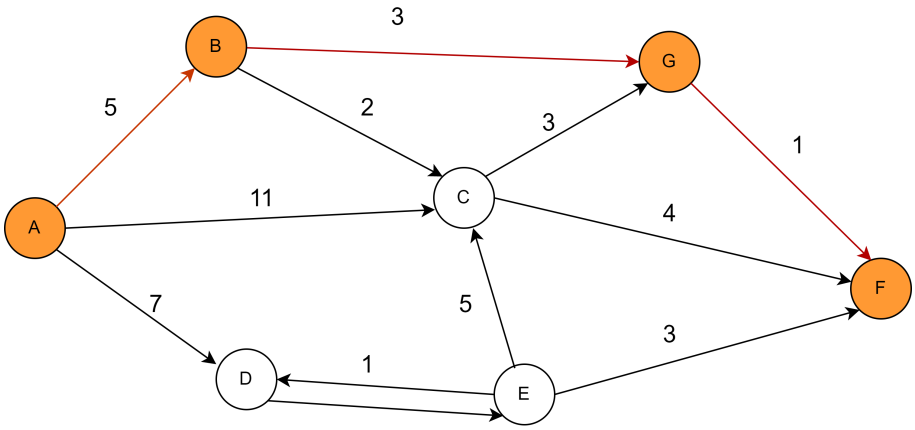


Figure 1: A weighted directed graph showing the shortest path from node A to node F.

Figure 1 is a weighted graph that colors the nodes in the shortest path from node A to node F. There are several key concepts to note about shortest paths.

There are algorithms that are capable of finding the SP in a reasonable time that are good for generating the training data and bad if we are looking into a sub-optimal solution, that can be still considered usable.

For checking the solution we need to find the solution first because we can not test the Machine Learning (ML) approach without solving the problem.

The SP is a higher order concept because the path consists of multiple nodes and edges and it is hard to represent it in a way that ML approaches can make good use of it.

In practice, if the problem is more specific we can add some other attributes that relate to the path. We can also include sub-optimal paths as node attributes but it can raise other issues.

3.2 Dijkstra's Algorithm

Dijkstra's algorithm is considered a static, single-source algorithm [10]. Dijkstra's algorithm is a greedy algorithm that can provide the shortest path without excessive use of runtime and memory.

The basic idea in Dijkstra's algorithm is rough as follows: First, a set S of vertices/nodes, and the path weights from the source point (starting point) s to the nodes in the set have been determined. During the operation of Dijkstra's algorithm, different paths are repeatedly selected, that is, the nodes with the optimal path estimation are added to the infinite array, the node i is added to S, and all the path weights of the node i is weighted by the function, and the final function weight value is selected. The SP is the optimal path [11].

This provides Dijkstra's algorithm time complexity as $O(|V|^2)$. This is considered a poor approach because of the use of an unsorted array.

To make Dijkstra's algorithm more efficient it can be implemented with a priority queue (PQ) and binary heap instead of an unsorted array. A priority queue is created with each of its elements containing the node and the distance to that node. The element with minimum distance shall be considered as the one with the highest priority. Instead of linearly searching for the minimum element priority queue takes care of extracting the element.

In this scenario of Dijkstra's algorithm, the time complexity is dependent on the running of Dijkstra's operation and the key heap operations. Dijkstra's operations will occur O($|V| + |E|$) times, with the key operations taking O($\log|V|$). The simplification provides the total time complexity of O($|E|\log|V|$).

A simple implementation of Dijkstra's algorithm with over Figure 1, considering the Source and End node as A and F respectively is explained below:

1. Initialize all the nodes' distance/weight to infinite.

2. An empty PQ with key-value pair is created storing nodes and weights/distance of the path traversed. And weights will be used to compare the shortest path between other pairs in the priority queue.

3. Source node A enters first in the priority queue with weights equal to 0. As it takes 0 distance to reach node A.

4. Source node A traverses to all three neighbors (nodes B, C, and D) shown in table 2 to get their weights and add them to the priority queue. The PQ uses min-heap adding the neighboring edges will be put in minimum to maximum weight order.

nodes	A	B	C	D
Weights	0	5	11	7

Table 2: Priority queue from node A

5. Node B has the least weight and now it will traverse to its neighbors similarly to add all the path's weight to the priority queue shown in Table 3.

nodes	B	C	G
Weights	0	2	3

Table 3: Priority queue from node B

6. Node C has the least weight and it will add all its 2 neighbors (G and F nodes) shown in Table 4. F node is also an end node.

nodes	C	G	F
Weights	0	3	4

Table 4: Priority queue from node C

7. With this whole traverse path's weight will be calculated and stored in a distance array with the path found and its total weight.

8. Dijkstra's algorithm backtrack to node C and take the second neighboring node G. Node G has a single neighbor which is end node F. Traversed path is calculated again and added to the distance array.

9. Again the backtracking reaches node B and the second neighbor node which is node G has already been traversed. Similarly, the path will be added to the distance array.

10. As Dijkstra's algorithm traverses through the nodes, all its nodes even the shortest path are found. It will traverse all other nodes similarly.

Traversed Path	Equivalent Weight
$A \rightarrow B \rightarrow C \rightarrow F$	11
$A \rightarrow B \rightarrow C \rightarrow G \rightarrow F$	11
$A \rightarrow B \rightarrow G \rightarrow F$	9
$A \rightarrow D \rightarrow E \rightarrow F$	11
$A \rightarrow D \rightarrow E \rightarrow C \rightarrow F$	16
$A \rightarrow C \rightarrow F$	15
$A \rightarrow C \rightarrow G \rightarrow F$	15

Table 5: Distance path array from source to end node

11. After adding all the traverse paths in the distance array shown in Table 5, the minimum weight path which was found as $A \rightarrow B \rightarrow G \rightarrow F$ and will be considered as the shortest path.

Using the priority queue does decrease the runtime and increases optimization for Dijkstra's algorithm. No need of sorting of the unsorted array needs to be done while traversing from different paths. That's why Dijkstra's algorithm can be considered an ideal solution to finding the shortest path while keeping in mind that Dijkstra can handle positive weights only and wastes time and resources because of it's blind search.

3.3 Graph Neural Network Approach

Data can be naturally represented by graph structures in several application areas [5], including proteomics, image analysis, scene description, software engineering, and natural language processing. A graph neural network is a state-of-the-art neural network model that is capable of capturing the dependencies between graphs by transferring messages between graph nodes. GNN can perform inference on data represented by graphs and can also be explored to find the shortest path. The GNN method, which is predictive in nature and a potential solution to solve the shortest path problem, will be explored as a foundation for this experiment even though Dijkstra.

A GNN is proposed to collect and summarize information from the graph structure [12]. GNNs are deep learning models for graph-structured data, which achieve state-of-the-art results for many graph-related tasks. GNNs can be classified into different categories depending upon their model architecture [13] such as recurrent GNNs, convolutional GNNs, Graph Auto-Encoders and Spatial-Temporal GNNs.

We will be denoting a node as v_i, an edge as e_k, and the global attributes as $\mathbf{u}$. We also use the symbols s_k and r_k for indicating the indices of the sender and receiver nodes respectively.

Algorithm 1 General Algorithm for Computational steps of GNN

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

The Algorithm 1 stated above is drawn from [12], which explains the general computational steps in the GNN. In the coming sections, it will be observed how this algorithm serves as a base in the implementation of this research. Also, it will be seen that how work done in this research is different from the above algorithm.

Whereas ρ and ϕ used in the above algorithm are the aggregate and update functions respectively and could be the NNs. ϕ^e, ϕ^v, and ϕ^u are the update functions that update the edge attributes e , node attributes v and global attribute u respectively. Similarly $\rho^{e\rightarrow v}$ is the aggregate function which aggregate information from edge to node, $\rho^{e\rightarrow u}$ aggregates information from edge to graph representation, and $\rho^{v\rightarrow u}$ aggregates information from node to graph representation. A detailed explanation of every symbol used in the Algorithm 1 will be provided in the coming sections.

3.3.1 Internal Architecture of GNN

In this section, we will see how these aggregate and update functions are performing operations individually.

$$\begin{aligned} e'_k &= \phi^e(e_k, v_{r_k}, v_{s_k}, \mathbf{u}) & \bar{e}'_i &= \rho^{e\rightarrow v}(\mathbf{E}'_i) \\ v'_i &= \phi^v(\bar{e}'_i, v_i, \mathbf{u}) & \bar{e}' &= \rho^{e\rightarrow u}(\mathbf{E}') \\ \mathbf{u}' &= \phi^u(\bar{e}', \bar{v}', \mathbf{u}) & \bar{v}' &= \rho^{v\rightarrow u}(\mathbf{V}') \end{aligned} \tag{1}$$

Equation 1 is explaining how updation and aggregation are performed in a GNN. e'_k is the updated edge attributes with index k that collects the updated information from edge e using the information from the corresponding edge, node and the graph representation.

$\bar{e}'_i$ collects the aggregated information coming from edge to node representation at node i. Similarly v'_i is the updated node attributes with index i by using the updated information from previously calculated $\bar{e}'_i$, node v_i and global representation. Similarly $\bar{e}'$ collects the aggregated information from edge to graph representation. $\mathbf{u}'$ is the information that is collected globally. Similarly $\bar{e}'$ collects all the aggregated information from edge to graph representation. $\bar{v}'$ collects the information coming from the aggregation of node to graph representation.

Where $\mathbf{E}'_i = \{(e'_k, r_k, s_k)\}_{r_k=i,k=1:N^e}$ is the set of updated edges connected to i, $\mathbf{V}' = \{v'_i\}_{i=1:N^v}$ is the set of all updated node , $\mathbf{E}' = \cup_i \mathbf{E}'_i = \{(e'_k, r_k, s_k)\}_{k=1:N^e}$ is the set of all updated edges. These notations will be explained more in Section 3.3.2.

3.3.2 Computational steps in GNN

We can dissect Algorithm 1 and see in-depth that what are the operations that are performed by each argument line by line.

- At every edge ϕ^e which is an update function with arguments $(e_k, v_{r_k}, v_{s_k}, \mathbf{u})$ is performed and in return giving the value of e'_k, e'_k is the updated edge attributes at index k. Looking into a specific node i the set of resulting per edge, will be $\mathbf{E}'_i = \{(e'_k, r_k, s_k)\}_{r_k=i,k=1:N_e}$, while $\mathbf{E}' = \cup_i \mathbf{E}'_i = \{(e'_k, r_k, s_k)\}_{k=1:N^e}$ is the set of all per-edge outputs. Figure 2a is showing how edge is updated.

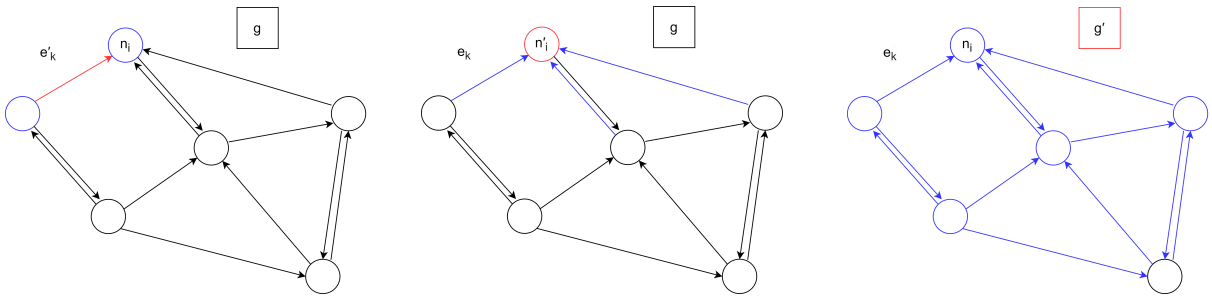
- Now these updated edge attributes are aggregated for node i, $\rho^{e\to v}$ is applied to E'_i and returns $\bar{e}'_i$. $\bar{e}'_i$ is the aggregated edge attribute with index i.

- Nodes are updated by applying ϕ^v to each node i and updated node attribute v'_i is computed and the set of resulting per-node output is $\mathbf{V}' = \{v'_i\}_{i=1:N^v}$. For reference Figure 2b is showing how nodes are updated.

- All edge updates are aggregated into $\bar{e}'$ by applying $\rho^{e\to u}$ to E' and will be used in global update.

- Now node updates are aggregated into $\bar{v}'$ by applying $\rho^{v\to u}$ to V' and will be used in next step to compute global update.

- Global update $\mathbf{u}'$ is computed once per graph by applying ϕ^u to the graph and can be seen in Figure 2c.



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

Figure 2: Updates in a GNN. Red indicates the attribute being updated, and blue indicates attributes that are involved in the updation.

3.3.3 Message Passing Neural Network

MPNN [14] is a (Graph Neural) GN block that is being deployed in this research. Based on the node, edge, and global properties, this MPNN predicts node, edge, and global output attributes. Keep in mind that aggregated edges are not included in the global prediction. Figure 3 depicts the internal architecture of MPNN.

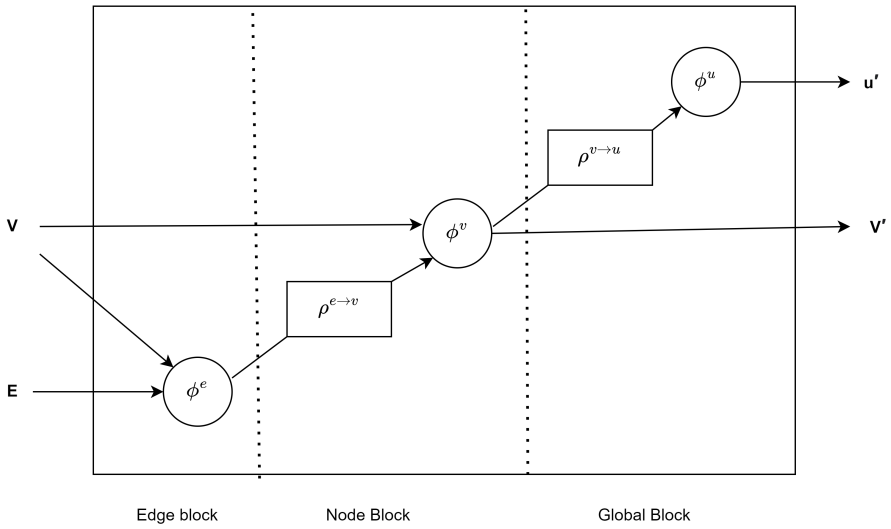


Figure 3: Message Passing Neural Network

3.3.4 Previous work with GNNs

Different architectures of GNNs have been adopted in the study to compare results: The Encode Process Decode (EPD) [4] being the base for every GNN implemented in this has four Multilayer Perceptrons (MLPs) and a GN block and the details are mentioned in Section 4.1.3. GAT [15] is taken into consideration for improving the GNN implementation. As GAT uses the attention mechanism to check the most relevant features in the connected nodes. Instead of just understanding every feature of the graph to see the prediction, they are essential in recognizing key information. This strategy also includes the notion of the masked attention mechanism. GCNs are able to accept input as a matrix of graphs and train a function of features on them. Since they were inspired by CNNs, which apply filters to grids to extract features and produce feature maps. Finally, GGSNN were considered a beneficial technique because this GNN gives a sequence of outputs. The idea behind trying GGSNN is that it will beat the previous GNN because of its nature and it was thought it will be helpful in giving the shortest path more accurately. This node classification scenario may create many consecutive outputs, which would increase the effectiveness of the outputs in determining the shortest path. The detailed description of every GNN can be seen in Section 4.1.3.

4 Methodology

We will first attempt to give an intuition to each approach, followed by specific implementation details.

4.1 Approaches and Intuition

4.1.1 Overview

The GNN adapted to the SP problem implemented in [12] acted as our baseline model. The particular was initially written in TensorFlow 2.11.0[3] and TensorFlow GNN 0.5.0[4]. Their specific implementation can be found at[5]. Given that it was decided to experiment with multiple approach, the model in PyTorch 1.13.0[6] PyTorch Geometric 2.2.0 [7] was re-implemented as the library was mature compared to TensorFlow. From the preliminary experiments, we have observed that there was room for improvement. This re-implementation together with all the different approaches implemented can be found on Ericsson Research's GitHub[8].

4.1.2 Adapting a Full GNN Block to SP

The concepts presented by [12] and Section 3.3 are general to use in our project. However depending on our problem we made some changes.

While adopting GNNs, we have to take into account the particular characteristics of the problem. In some problems, it is easy to get information from the system itself because the information is contained in the system. In the SP path problem, we do not have an easy way to define the path attributes in the graph. One idea that might add something related to it is adding landmarks and this idea will be discussed in Section 11.4.

4.1.3 Models Used and their Modifications

The models all follow the same architecture, consisting of three different parts: the encoder, the processor, and the decoder. The idea of the architecture is initially proposed in physical simulation [16]. Figure 4 shows the architecture of EPD.

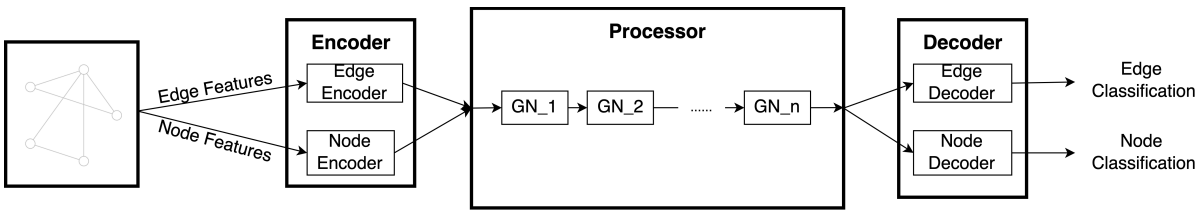


Figure 4: The architecture of EPD. The figure of the processor is introduced by Battaglia et al. [12]

The target of the encoder is to turn the features of a graph into low-dimensional representations so that we can utilize the converted numeric representation in the next step. In the Encode, we set 2 MLPs to encode the graph information. One encodes the input edge information, and the other encodes the input node information. The decoder is

[3] https://github.com/tensorflow/tensorflow/releases/tag/v2.11.0

[4] https://github.com/tensorflow/gnn/releases/tag/v0.5.0

[5] https://colab.research.google.com/github/tensorflow/gnn/blob/master/examples/notebooks/graph_network_shortest_path.ipynb/

[6] https://github.com/pytorch/pytorch/tree/v1.13.0

[7] https://github.com/pyg-team/pytorch_geometric/releases

[8] https://github.com/EricssonResearch/spreadnet

similar to the encoder, it also has 2 MLPs to take the output embeddings of the processor and try to reconstruct the data as output.

The most important part of the architecture used by us is the processor. The processor is a stack of GN blocks, which help us solve the Shortest Path problem. In each GN block layer, both nodes and edges have hidden representations. There are various GN blocks implemented by us in this project. We will introduce them in the following sections.

Message-passing Neural Network The structure of the MPNN used in our Shortest Path problem is different from the general GNN architecture introduced in Section 3.3. Since we just need to classify the nodes and edges, the global attributes updating and representing can be removed. Therefore, the modified MPNN just focuses on updating the edge and node embeddings.

Figure 5 and Equation 2 show how the modified MPNN updated the edge and node embedding. First, for each edge k, we use edge feature e_k, and the corresponding node features v_{r_k} and v_{s_k} to compute the updated edge feature e'_k. Then, we use aggregating operation $\rho^{e\to v}$ and the set of updated edge features E'_i for each node i to do the information aggregation. Next, for each node i, we use the aggregated edge feature $\bar{e}'_i$ and the original node feature v_i to compute the new node feature. Finally, we output the updated edge feature e'_k and node feature v'_i. The updating structures used in our MPNN are MLPs.

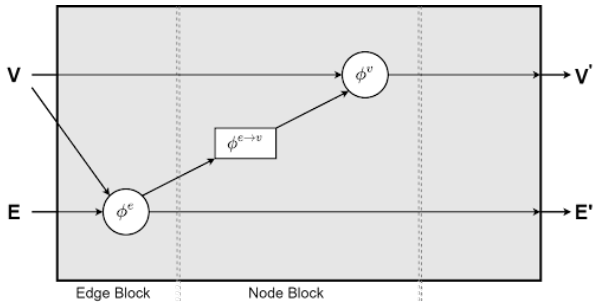


Figure 5: Modified Message-passing Neural Network

$$
\begin{aligned}
e'_k &= \phi^e(e_k, v_{r_k}, v_{s_k}) \qquad & \bar{e}'_i &= \rho^{e\to v}(E'_i) \\
v'_i &= \phi^v(\bar{e}'_i, v_i) & &
\end{aligned} \tag{2}
$$

Figure 6 is an example of message passing introduced by Battaglia et al [12]. With multiple MPNN layers stacked together, the propagation of the information happens simultaneously for all nodes and edges in the graph during the full message-passing procedure. And we can conclude that after the full message passing steps, the information can be propagated to all the nodes and edges in the graph.

Adaptive Message-passing Neural Network The stacks of MPNN layers introduced in the last section can help us propagate the information in the graph. When we do the SP task on a larger graph, it is intuitive to think about adding more MPNN layers to fulfill the propagation.

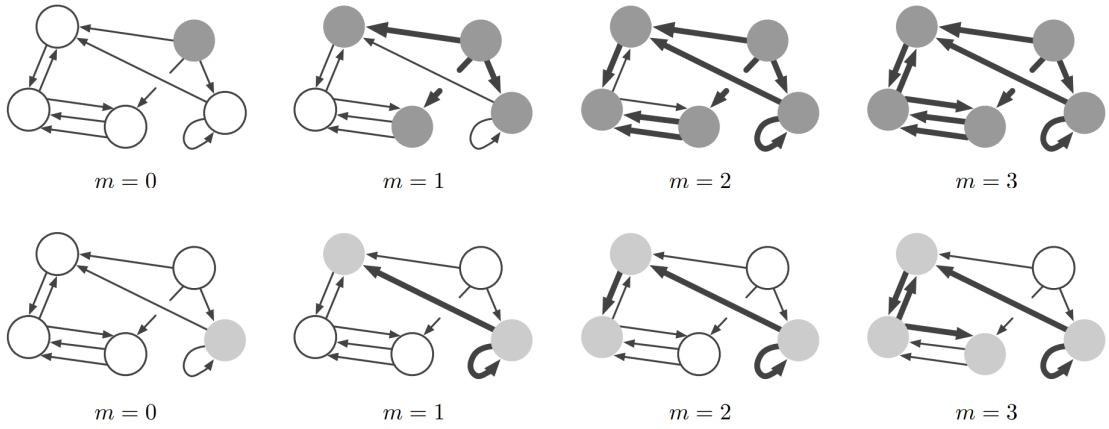


Figure 6: Example of Message Passing introduced by Battaglia et al [12]

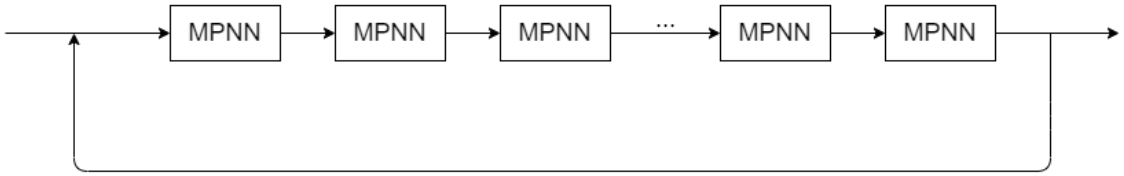


Figure 7: Adaptive MPNN: reusing the MPNN layers according to the size of the input graph.

There is also another way that can help us get a similar result — try to reuse the stack of MPNN layers multiple times. We first input the node and edge embeddings into our MPNN, after the initial message passing steps, we then extract the output and make it the input again and continue to do the message passing steps, as shown in Figure 7. This process can repeat many times. The number of reusing depends on the graph sizes in the dataset. The larger graph, the more reusing times.

The purpose of reusing the MPNN layers is that we want to pass as much information as possible in a network of limited size. With reused blocks, we can define a relatively small network and do many times of message-passing steps when we need to find the shortest path in a huge graph.

Graph Convolutional Network The first well-known GNN model explored for the project is GCN [17], and based on the terminology introduced in [13], it belongs to the category of convolutional graph neural networks (ConvGNNs). GCN borrows the idea of CNNs and generalizes the operation of convolution from regular grid data to irregular graph data. The main idea is to extract a node's high-level representations by first aggregating the features from its one-hop neighbours, and the sum is used to update the node's features.

However, irregular graph data are usually variable in size and structure, which means the number of a node's one-hop neighbours is non-deterministic. In other words, GCN cannot rely on a fixed-size kernel to perform aggregations and the key design of GCN, a function of aggregation and propagation rules, is what conquers the problem. Specifically, the function of each GCN layer takes as input a feature vector for all nodes and an adjacency matrix representing graph structure and outputs an updated node-level

representation. It can be written as:

$$H^{(l+1)} = f(H^{(l)}, A) \tag{3}$$

where f stands for formula, H is the input feature vector of node and A is the adjacency matrix.

The simplest form of propagation rule is to multiply the feature vector with the adjacency matrix. Unfortunately, simple multiplication will cause two problems: nodes will lose their own features, and the scale of the feature vector will change completely. To fix them, an identity matrix is first added to the adjacency matrix, which is identical to adding a self-loop edge to each node, so the node can retain its own features. Further, the diagonal degree matrix is adopted to normalize the multiplication, so the feature vector can remain in a certain scale through layers. Combining these two techniques, the final propagation rule introduced in [17] is:

$$f(H^{(l)}, A) = \sigma(\hat{D}^{\frac{1}{2}}\hat{A}\hat{D}^{\frac{1}{2}}H^{(l)}W^{(l)}) \tag{4}$$

Where $\hat{A} = A + I$, and I is the identity matrix. $\hat{D}$ is the diagonal node degree matrix. W is a learnable weight matrix and σ denotes an activation function.

On the other hand, the limitation of GCN is caused by the diagonal degree matrix. As the propagation rule guides a node to gather a weighted sum of features from its neighbors and the weights are calculated by the degrees of neighbors. If a neighbor has a larger degree, the weight for it will be smaller, so the features it contributes will be less. The more neighbors a node has, the less import the node is to its neighbors. Nevertheless, the rule is not desired for solving the SP problem because the shortest path can possibly have a lot of nodes and edges in the graph. The model introduced to overcome this limitation is GAT.

Graph Attention Network Although the graphs for the SP problem do not exhibit any particular structure patterns, the features of the start node should propagate long enough to the end node and moreover, the propagation should ideally toward a certain direction along the shortest path, thus if a node is in the path, only one of its first-order neighbors should have the similar features as the node and both then can be classified as in the same class. By applying an attention mechanism, GAT can weigh the neighbors of a node and control the propagations of node features.

Like GCN, a single graph attentional layer takes the vector of node features as input and produces a new vector of node features as output. First, a learnable weight matrix W is applied to every node to transform the input features h to high-dimension features, and a shared attention mechanism is then performed on the nodes with their first-order neighbors to compute attention coefficients. The coefficient indicates the importance of a neighbor's features to a central node, and for easy comparison among neighbors, it is then normalized via the softmax function. The key is the function for the attention mechanism, which is normally a single-layer neural network, i.e., MLP, that is parameterized by a learnable weight vector $\vec{a}$. Typically, an activation function is applied to the neural network and the LeakyReLU function is selected for GAT because it has a small slope for negative values instead of a flat slop in ReLU and helps to achieve a better performance in this case. After applying a LeakyReLU function as the activation function to the neural network, the fully expanded computation of coefficients is:

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\vec{a}^T[W\vec{h}_i||W\vec{h}_j]))}{\sum_{k\in N_i}\exp(\text{LeakyReLU}(\vec{a}^T[W\vec{h}_i||W\vec{h}_k]))} \tag{5}$$

where T represents transposition, W is the weight matrix, h is the input features of nodes, i is the target node, j is the source node, N is the amount of neighbours of i, k is one of the neighbours, and $||$ is the concatenation operation. Once the coefficient for each neighbor is obtained, it will be linearly combined with the neighbor's features and the sum of combinations from all neighbors is the final output features of the central node:

$$\vec{h}'_i = \sigma(\sum_{j \in N_i} \alpha_{ij} W \vec{h}_j) \tag{6}$$

Furthermore, to stabilize the process of attention mechanism, GAT employs a technique called *multi-head attention*, which is that in each GAT layer, multiple independent attention mechanisms are performed, and their results can be either concatenated or averaged to generate the final output features as shown in the Figure 8.

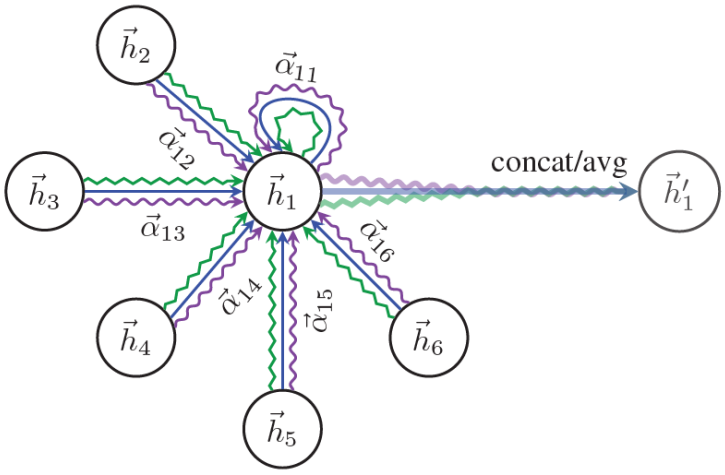


Figure 8: An illustration of multi-head attention [15]

Gated Graph Sequence Neural Network Unlike previous models, the sequence model is another way to crack the SP problem. In the previous methods, our models tried to learn node/edge representation based on decoded node/edge embedding. However, the models we have so far are only the single output prediction. It can be misleading when we check the accuracy on the nodes/edges that are not in the shortest path. Therefore, we proposed a sequence model to make path prediction at the graph level instead of node/edge prediction.

To handle the sequential output problem, two main parts need to be learned within the sequence model. In the beginning, the model needs to learn a representation of the input graph based on nodes and edges. And then, it needs to learn the representation of the internal states during the process of producing a sequence of outputs. Here we used gated recurrent units to learn the partial output sequence. Here we adopted the GGSNN [18] to solve the shortest path prediction problem. GGSNN consists of two different GGNN units, one is the propagation model, and the other is the output model. The propagation model computes the node representation of each step during the sequence-producing process. The output model maps the node embedding to the output score at each step.

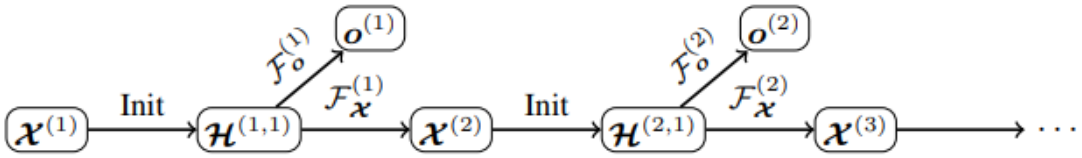


Figure 9: Architecture of GGSNN model by [18]

Here, $\mathbf{X}(k)$ is intermediate node annotation at the output step k. $\mathbf{H}(k,T)$ is the output score at the output step k and propagation step t. $\mathbf{F}_o(k)$ and $\mathbf{F}_x(k)$ are two GG-NNs units to predict the output sequence.

4.1.4 Observed Issues and Proposed Solutions

The GNN is trained to classify the edges and nodes to be either "in path" or "not in path". However, in most of the training graphs, the edges and nodes are likely to be "not in path" in a much higher frequency than they are found to be "in path". This ratio can be highly imbalanced, such as when more than 99% of the nodes are actually "not in path". This is usually commonplace in medium to larger sized graphs. Therefore, it was not surprising when the trained model started predicting all of the nodes to be classified as "not in path" and the accuracy of the prediction was still found to be quite high.

This posed a "class imbalance" problem wherein the imbalance in the frequency of occurrence for the two classes caused the model to be trained with an inherent bias. A possible solution for this imbalance was to modify how the loss is calculated.

Weighted Loss The original function for calculating the loss was a "cross-entropy" loss, with a mean reduction over the calculated values for each of the nodes and the edges. This idea now awards a greater penalty to the particular components which have been *incorrectly classified* to be "not in path." The penalty in question is a multiplication factor that is calculated based on the size of the graph.

Equations 7 and 8 describe the way that the penalty is computed for nodes and for edges respectively. This penalty is multiplied to the calculated loss for the selected nodes/edges. Thereafter, a mean is computed over the entire array.

$$ratio_{nodes} = \frac{\text{number of nodes not in path}}{\text{number of nodes in path}} \tag{7}$$

$$ratio_{edges} = \frac{\text{number of edges not in path}}{\text{number of edges in path}} \tag{8}$$

Weighted Loss using Euclidean Distance An alternative to the same idea is to calculate the penalties based on a factor other than the graph sizes. This multiplication factor is based upon the Euclidean Distance [19] between the *predictions* and the *original values*. This method is quite different since it penalizes all incorrectly classified nodes and edges regardless of whether they were supposed to be in the path or not. The multiplication factor r is also not a constant but is different for each node and edge in this case.

Equation 9 describes the way that this penalty is computed. Here, r is an array, which first computes the Euclidean distance between the ground truth and the predictions. An exponential function is then applied to this distance for better scaling. This array is

multiplied with the array containing the calculated loss. Finally, the mean is calculated over the loss values.

$$r = exp(||ground\_truth - predictions||) \tag{9}$$

Weighted Loss favouring Connectivity Another issue that was noticed in the predictions was that the nodes and edges classified as "in path" were sometimes quite disconnected from each other. They did not create one path from the start node to the end node, having a few random predictions all throughout the graph. To solve this, the idea was to modify the loss calculation with another perspective. This time the nodes and edges were targeted in a manner that if they were disconnected or isolated, they received a greater penalty. Therefore, the connectivity of the components was favoured while penalizing the losses.

This was awarded two types of components:

1. When an edge is correctly classified as "in path" but the surrounding nodes are not.

2. When two neighbouring nodes are correctly classified to be "in path" but the edge between them is not.

4.2 Meta Models

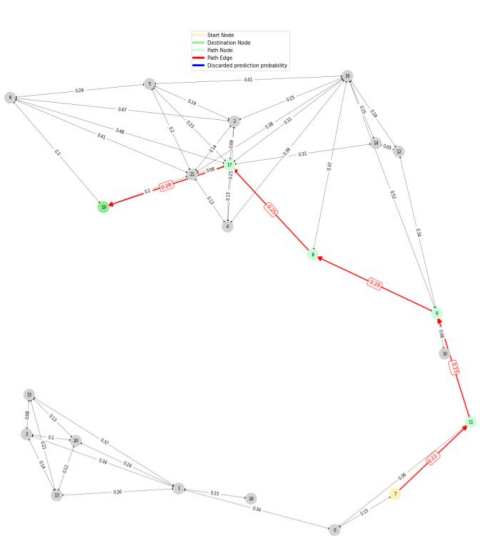
4.2.1 Hybrid Models

Dijkstra's algorithm, in comparison with any of the GNN models we have discussed in this study, performs well in terms of path accuracy. Our major motivation while designing the hybrid model is to harness the potential of both GNN models and Dijkstra's algorithm. In this post-processing step, the predicted path by the GNN is evaluated for its completeness. In a digraph, a complete path should be a continuous path in the intended direction from start to finish.

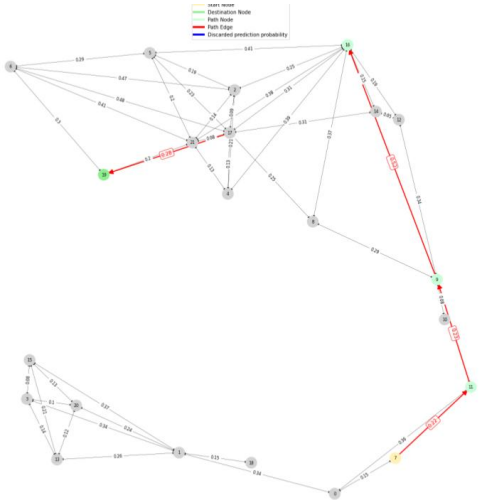
The hybrid model traverses each node and edge in order from the start node, while looking ahead one node at a time. Nodes and edges are classified to be in the path based on GNN predicted probability. The hybrid approach detects this disconnection in the path and then uses Dijkstra's algorithm to calculate the remaining path until the end node. These two paths are then aggregated into a single path, which is then returned as a result.

In essence, the hybrid model traces the path predicted by the GNN as far as it goes continuously, for the remaining part it uses Dijkstra's algorithm for path completion. One of the main advantages of this approach is that the continuity and completeness of the path (start and end nodes are connected) are guaranteed. This helps in filling the gaps where the GNN model failed to discover appropriate nodes or edges. In the best-case scenario, there will be one odd edge or node gap that is patched up by the hybrid model through Dijkstra's algorithm. While on the other hand, the worst-case scenario would be using Dijkstra's algorithm from start to end as the GNN fails to predict any nodes or edges as a path.

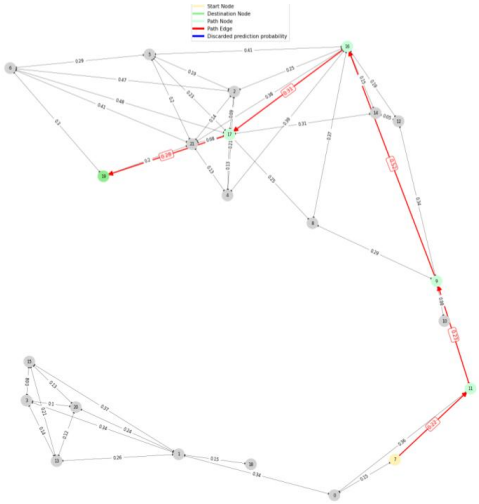
A downside of this approach is that it fails to optimally utilize the result already produced by the GNN models, as it disregards any predicted path fragments that exist after the first encountered point of disconnection.



(a) Ground truth



(b) disconnected MPNN path prediction



(c) Hybrid result

Figure 10: Illustration of Hybrid model

4.2.2 Bi-Directional Model

It is believed that alongside the normal start-to-end GNN inference, the end-to-start inference could improve the prediction performance. The reversed prediction could be done by swapping the start and the end nodes, reversing all the graph's directed edges, and passing it to the same trained model. Both inferences are then merged into a single prediction favoring nodes and edges with higher probabilities.

4.3 Implementations

The actual methods undertaken by the different GNN models differed in implementation. There was variance in setting the number of layers at every step as well as the order and method in which the node and edge embeddings were updated. These routines are also compared with our non-GNN implementation.

4.3.1 GNN Implementations

Message-passing Neural Network The MPNN uses MLPs to update the node and edge embeddings. Note the size of node embedding from the encoder or the previous MPNN layer as *node_in*, and the size of edge embedding from the encoder or the previous MPNN layer as *edge_in*.

First, we do the edge updating. The input size of the MLP for updating edge embeddings is $node\_in + node\_in + edge\_in$. The size of the output of edge updating is the preset *hidden_size*.

Then, we do the node updating. The input size of the MLP for updating edge embeddings is $hidden\_size + node\_in$, and the output is also the preset *hidden_size*. Therefore, the output size of edge and node embeddings are both *hidden_size*. The aggregation operation used by us in the implementation is $AVERAGE$.

Adaptive Message-passing Neural Network We need to set the number of the basic MPNN layers, and we calculate how many times we need to repeat the blocks based on the average number of nodes per graph and the number of nodes in the path, as shown in Equation 10.

$$repeatTimes = \lceil \frac{avgNumNodes}{ratioNodePath} \rceil \tag{10}$$

Graph Convolutional Network The implementation of GCN follows the same Encoder and Decoder architecture, where the Encoder first projects the original features of nodes and edges into high-dimension embeddings and the Decoder is used to extract the representations of nodes and edges from the high-dimension embeddings. What between them is a multi-layers GCN block and each GCN layer performs exactly the propagation rule 4. Following the suggestion in [17], a two or three-layers GCN usually performs better than a deeper GCN. However, the testing result was not good. The potential explanation is that in the datasets of the experiments of GCN, the similarities between a node and other nodes depend on the orders of the neighborhood and some datasets exhibit a similar graph pattern that different types of nodes are already roughly clustered into different groups, therefore three steps of feature propagation and aggregation are just enough for a node to pass its features to the nodes in the same category, which is certainly not the same case as the SP problem because in a graph of the SP problem, the similarities among all the nodes in the shortest path should keep the same across a few to many orders of the neighborhood, so the neural network has to be deeper to pass features of a node to some distant nodes.

Graph Attention Network As shown in Figure 11, the implementation of GAT follows the same architecture as MPNN and GCN. In the encoder, the original features of nodes and edges are first projected to high-dimension embeddings, which are then fed into the GAT processor. For each GAT hidden layer in the processor, six-heads attention is computed, and their results are concatenated before passing forward, but if the next layer is the output layer, the results are averaged instead to generate the final high-level representations. In the end, the decoder, a 2-layer MLP, is responsible to extract the classes of nodes and edges from the final representations.

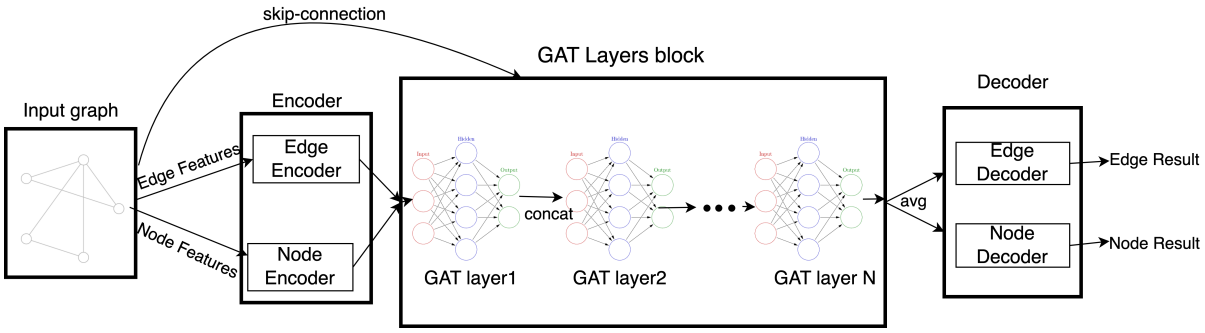


Figure 11: The architecture of Graph Attention Network

4.3.2 Non-GNN Implementations

The chosen algorithm was the NetworkX[9] Djikstra's shortest path algorithm[10]. This algorithm was chosen due to its efficient code and that we were already utilizing it for graph construction. Memoization was added with two different strategies detailed below. This information is then saved into a dictionary structure. The memoization table ensures that Dijkstra's algorithm is only ever needed to be called one time for a specific search, then afterward it will be a dictionary search. Figure 12 is the flow chart diagram of Dijkstra with memoization.

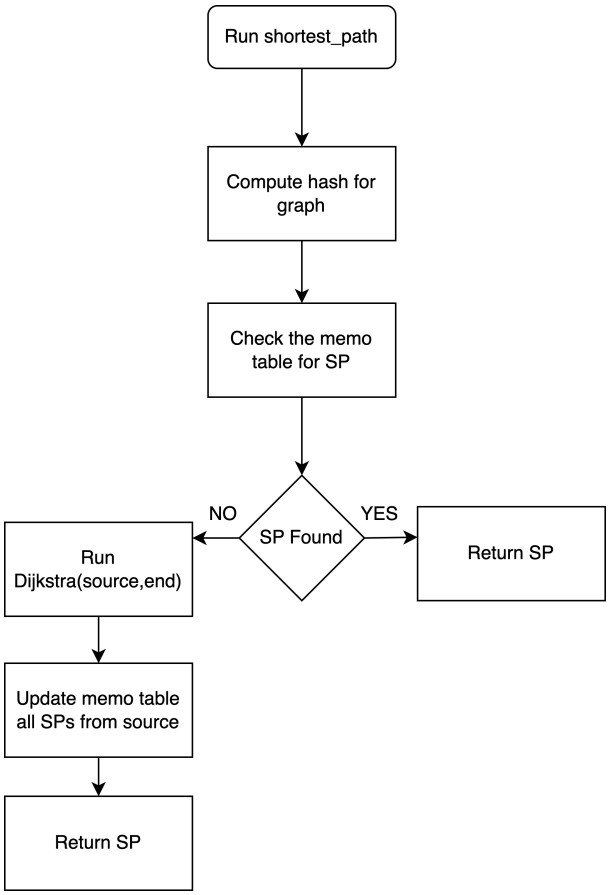


Figure 12: Dijkstra's algorithm with memoization flow diagram

[9] https://github.com/networkx

[10] https://networkx.guide/algorithms/shortest-path/dijkstra

The limitations of Dijkstra's algorithm are that it can only be used with non-negative graphs. This isn't a problem for us since this fits the types of graphs used for this research. If the node is not connected, the only way for the algorithm to return that information is when it visits all the nodes that are connected.

The memoization table is created with a Python dictionary. The dictionary can get and set items on average in a time complexity of O(1) and O(n) in the worst case[11].

Two methods for amending Dijkstra's algorithm with memorization were tested. One method was to find the complete minimum spanning tree of a graph from a source and to identify only the shortest path of the graph. Dijkstra's Full Path Search (DFPS) identifies all shortest paths from one source node defined as the shortest path tree. These can all then be stored in the dictionary. The Dijkstra's Single Path Search (DSPS) will stop searching once the shortest path to the end node has been found.

5 System design

5.1 Dataset Generation

Compared to the general description of a graph, the data we used has some specific patterns that have to be mentioned. We use geographical threshold graph[12], that in simple terms creates a rectangle that is then populated with nodes whose coordinates are sampled from a uniform distribution. Those nodes also have weights, in addition to their positions. The node weights are randomly sampled from an exponential distribution. Two nodes are connected by an edge if their added weights W_1 and W_2 are greater or equal to a parameter θ multiplied by their euclidean distance, R, between the two nodes.

$$W_1 + W_2 \geq \theta R \tag{11}$$

After applying Equation 11, we need to first convert the graph to a directed one. Between two connected nodes A and B we will now have two edges $A \to B$ and $B \to A$ both with the same weight. The included edge weights are the actual distance between the connecting node positions.

The last step in dataset generation is to make sure that the graph is connected. We do this by adding an edge between two estimated closest nodes between any two sub-graphs. Minimum spanning tree algorithms are not being used for this purpose as it requires too much memory and time for larger graphs.

The keynotes with regard to how we are generating the data are:

- The above-mentioned is implemented with the geographical threshold graph from NetworkX.
- Nodes have positions and weights. The positions are sampled from a uniform distribution, while the weights are sampled from an exponential distribution.
- Outgoing and incoming edges between two nodes have the same weight.
- The connectivity of the graph is decided based on θ.

[11] `https://wiki.python.org/moin/TimeComplexity`

[12] `https://networkx.org/documentation/stable/reference/generated/networkx.generators.geometric.geographical_threshold_graph.html#geographical-threshold-graph`

- While the nodes have both positions and weights, the shortest path is defined based on the weights of the edges only.

Multiple parameters are defined in a YAML[13] format to characterize the generated dataset. This controls the range of the nodes in each graph as well as the θ. The θ value increases after the set rate to promote graph diversity. Figures 13 and 14 show examples of graphs generated with the θ value of 5 and 10 respectively. Moreover, it is possible to specify the desired minimum ground-truth path length as well. The dataset is divided into training-validation and test sets, configurable by editing YAML files. The training-validation set would then be split during training via a ratio specified in the YAML file as well. Random()[14] is a pseudo-random number generator library that is built on algorithms to help generate a random sequence of data for the user. Python uses the Mersenne Twister[15] as the core generator. Seed allows the user to reproduce the same random sequence of data as it acts as a starting point on the algorithm on which the Random library is built.

The shortest path is added afterward using Dijkstra's algorithm as ground truth from a randomly selected start and end node that satisfies the specified minimum path length. The techniques used prioritize speed and efficiency. The different configurations and stochastic steps minimize the chance of repeating graphs to a bare minimum or non-existent at all. Finally, the generator outputs the graph in the JSON format and can be seen in the code snippet below as well as a figure visualizing the graph.

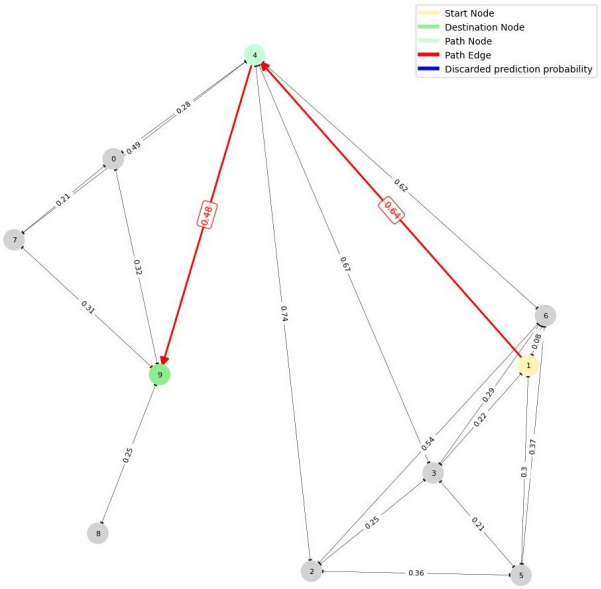


Figure 13: Graph with 10 nodes, $\theta = 5$

[13] https://github.com/yaml/

[14] http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/mt.pdf

[15] https://docs.python.org/3/library/random.html

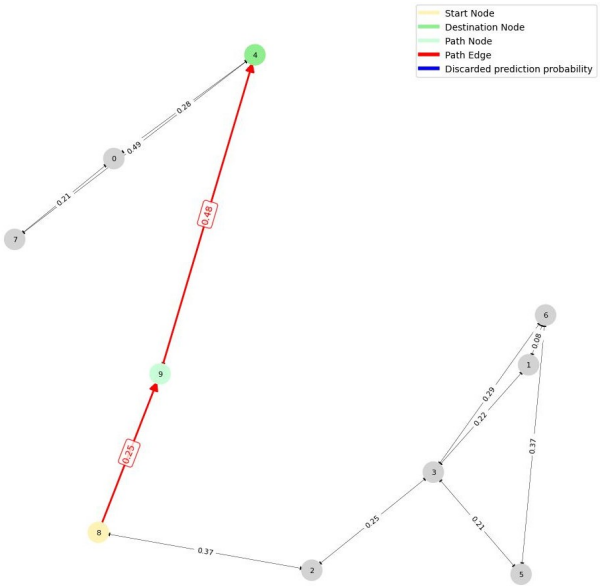


Figure 14: Graph with 10 nodes, $\theta = 10$

```
{
  "directed": true, "multigraph": false,
  "nodes": [
    {
      "id": 0, "weight": 1.561796983726534,
      "pos": [0.6857123175110225, 0.9855760506615988],
      "is_start": true, "is_end": false, "is_in_path": true
    },
    {
      "id": 1, "weight": 1.161796983726534,
      "pos": [0.6057123175110225, 0.9555760506615988],
      "is_start": false, "is_end": true, "is_in_path": true
    }
  ],
  "lin+9-9-ks": [
    {
      "weight": 0.14015498908798513, "is_in_path": true,
      "source": 0, "target": 1
    }
  ]
}
```

Code 1: Dataset generation- JSON file snippet

5.2 Training Pipeline

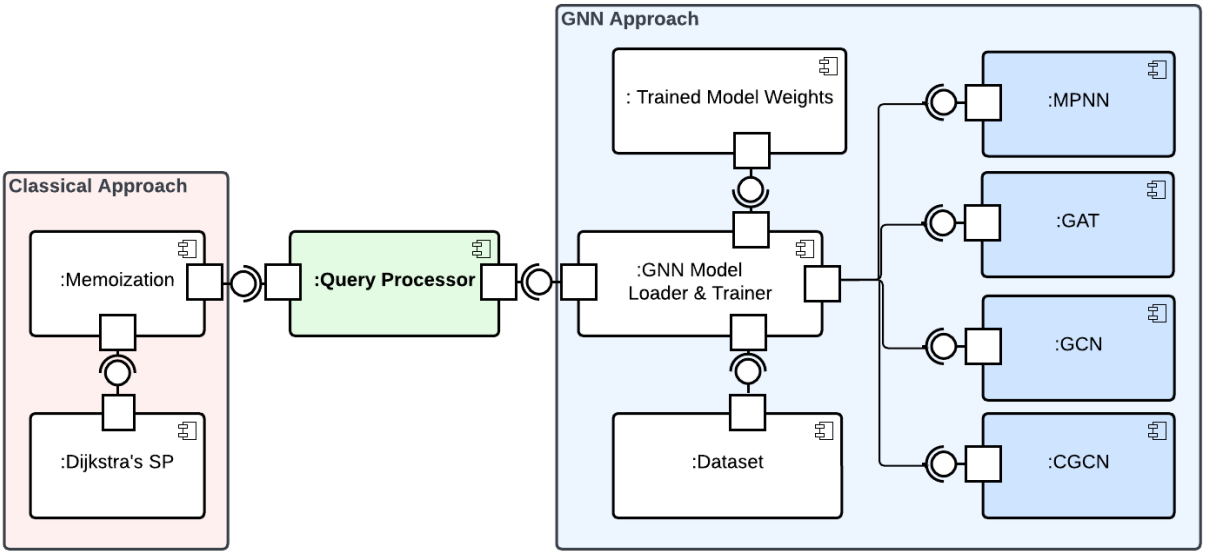


Figure 15: SpreadNet Components Diagram

Figure 15 is showing how Spreadet components are connected together. After the dataset is generated, the GNN Model Loader and Trainer can train one of the models and output the trained model weights inside the model's directory to be used for inferences. The model name can be specified by using the *model* argument in `experiments/train.py`. Just before the model training starts, the dataset is split into training and validation sets according to the set ratio and passed on to the trainer. In each epoch, the model is trained and validated. The metric graphs are also drawn after set epochs for the user to visualize. With *wandb*[16] flag set to true, it will upload all the metrics to wandb. Wandb is a central dashboard to keep track of the hyperparameters, system metrics, and predictions in order to compare models live, and share the results. Moreover, the GNN Model Loader and Trainer supports pausing and resuming for the user's convenience.

5.3 Post Training Pipeline

Because we have a variety of models and post-training methods we need to define a unified evaluation method. Doing this aids us in writing more general code for the evaluation part as it decouples the inner working of the inference and evaluation methods from the experiment writing. The three components of the post-training pipeline are:

- The unified model handler that loads and performs inference. It acts as a wrapper that unifies the loading and inference calls so that regardless of the model-specific procedure the person writing the experiment only needs to call for inference using the model.
- The accuracy metrics object contains different implementations of the accuracy metrics. It takes a model and a dataset and returns the metrics for that dataset.
- The experiment runner handles the models defined and metrics results. It is independent of the models used.

[16] `https://wandb.ai`

5.4 Query Processor

The query processor's role is to process one graph input at a time from the user and return a complete path that is the shortest, or an estimated shortest path, in as little time as possible. It is believed that after a certain graph size, the GNN processing speed could beat Dijkstra's shortest path algorithm. Therefore, experiments that compare the run-time between Dijkstra's algorithm and the GNNs are crucial to determine when and which method to use for a given graph size. It is also a program that conveniently grants users access to all the GNN models and their variations as well as the memoized Dijkstra's algorithm. The query processor can be started by using `experiments/qp.py`

The query processor's modes include *AUTO*, *GNN*, and *DIJKSTRA*. The *AUTO* mode would try to select the most suitable method to find the shortest path. However, since the runtime experiment in Figure 18 showed that the GNN could only come close to beating Dijkstra's SP algorithm, the *AUTO* mode threshold to switch to *GNN* mode is set to graphs when nodes go over 4000. The query processor's models include *MPNN*, *GCN*, *CGCN*, and *GAT*. The model parameter is used when the mode is set to *GNN* or *AUTO GNN*. The default model is set to *MPNN* as the experiments showed that it has the best validation accuracy. Furthermore, the query processor supports bi-directional GNN inference and DFPS as feature flags. The bi-directional flag defaults to False as it takes a longer time and has the potential to worsen the result. DFPS flag also defaults to False as it is slower than DSPS. After making the GNN inference, it uses Probability-first Search to generate a path using the output probabilities. The program's output is a NetworkX graph in JSON format and a figure showing the processed predicted path as well as the probabilities like the one shown in Figure 24.

6 Evaluation Methods

6.1 Metrics Overview

The nature of the SP problem does not directly translate to a nodes and edges classification problem. This approach results in the loss function missing some aspects of the SP definition presented in Section 3.1. A prediction from such a model will not inherently construct a sequence and will also not guarantee that a path can be formed. A fist step when designing a model for this problem would be to define metrics that are capable of adequately measuring the quality of the predictions. Only after we have defined those metrics we can start the search for better models.

The following metrics used have been split based on their usage, during and after training. Those used during the training phase require only a comparison of the prediction to the ground truth. The metrics used during post training are there to test for the actual presence of a path and its characteristics. The post training metrics often use different custom made algorithms that try to extract the path out of the prediction. Things to keep in mind while evaluating the model is that good training metrics results do not guarantee good post training metric and post training metrics should be later included as losses/training metrics.

6.2 Metrics Used During Training

Generating each training and testing sample as a single path prediction creates an imbalance in the number of nodes that are in path verses not in path. Metrics such as accuracy, if used alone, can be misleading. As a result we have chosen to use a number of metrics in order to get a better picture of what happens during training.

In short, we want metrics that are good at portraying the accuracy for both classes. One of the best way to show the accuracy is through the use of a confusion matrix followed by the F1-Score for getting a specific value that we can plot and use early stopping with.

In Table 6 we have defined the confusion matrix for our specific problem.

		Prediction outcome		
		p	**n**	**total**
ground truth	**p′**	True Positive	False Negative	P′
	n′	False Positive	True Negative	N′
	total	P	N	

Table 6: Confusion matrix. Here p and n stand for the prediction of a positive or negative class. p' and n' refer to the ground truth values. In capital letters we have the summations of the prediction or the ground truth.

In Equation 12 we have the F1 metric defined based on the confusion matrix in Table 6. Where TP means true positive, FP means false positive and FN means false negitive.

$$F1 = \frac{2 * TP}{2 * TP + FP + FN} \tag{12}$$

Other metrics we are using with severe drawbacks in one way or another are:

- Accuracy over all nodes. Severe class imbalance in favour of the not *in path* class. A model that never predicts nodes and edges to be in path has high accuracy.
- In path accuracy only. It worked well while we knew that the models were more likely to not predict the *in path* class. However, as soon as we introduced adaptations to the loss function this metric became deficient as the new models can be biased for the *in path* class.
- Precise accuracy. The ground truth and predicted nodes or edges must match exactly. This metric is too strict as missing only a node or an edge would not earn a score.

Other than the numerical results, we have often used visual representations of the predictions and compared them to the ground truth. While a numerical description of

the path-like features of the prediction would have been good, our rapid prototype and focus on the models instead of *good to have* but not essential features limited us to only visual remarks. Note that whatever assumptions were made based on these visual patterns have been to some extent included in the experiments we have performed.

6.2.1 Metrics Used Post-Training

Using the optimal path given by Dijkstra as the ground truth works. However, it is a scenario where the available solutions are heavily constrained.

We could relax the problem by saying that we also consider alternative paths that are not that much longer compared to the original paths. Doing this requires different approaches to how we identify a path as a good prediction might be different from the ground truth.

We can relax the problem by defining multiple paths from A to B where the paths have different scores depending on the length. This, however, changes the problem quite substantially as we are now predicting multiple paths/paths of different quality.

The alternative we have chosen is to use different algorithms that extract a path from the prediction and then compare it to the ground truth.

Probability walks The core idea behind what we call probability walks, is that we trust that a path formed out of nodes and edges with a high probability is a path close or identical to the optimal path.

Depending on how hard we are trying to look for a path we can have different variations of the same algorithm that we have named *Maximum Probability Walk* or (MPW) in short.

We will now give a quick intuition between those algorithms followed by what we actually measure.

- **Base MPW** Begining at the start node choose the edge with the maximum probability above a specific threshold. If no edges are available then chose the neighbouring node with the highest probability above the desired threshold. Stop when the end node has been reached when no edges or neighbours have a probability above the threshold, or the path leads to a dead end. No backtracking was used.

- **Exhaustive Probability Walk** is a greedy depth-first search algorithm. At each depth, it ranks the outgoing edges based on the end node's and its probability and explores the one with the highest probability first. It will return the strongest path if it cannot find the end node. The strongest path is defined as the path with the highest probability.

- **Probability-first Search** is a greedy hybrid of breadth-first search and depth-first search that guarantees a complete path, even if the start and end nodes are not connected by nodes/edges with probabilities. The algorithm explores edges/nodes with the combined highest probability first. If the end node cannot be found based on the specified probability threshold, the threshold will be lowered to find a complete path.

All the above algorithms return a path, be it from complete between the start and end nodes or just the start and a few other nodes. There are several things that we can measure in order to assess the quality of the path found:

- The existence of a path from start to end (complete). Given a set of graphs with a start and end node for a path, what percentage of the paths are found.
- If a predicted path is complete, how many more nodes does it have compared to the number of nodes in the ground truth.
- For a complete predicted path, how does its length compare to the length of the ground truth. Note that in topological graphs, the number of nodes and edges is directly related to the path length as the start and end node is randomly chosen. More nodes with fewer edges allows longer paths.

6.2.2 Memory Usage

In order to track the memory usage of Dijkstra's SP algorithm with the Memoization process. Python's memory-profiler[17] package was used to get the whole memory of the Dijkstra's algorithm process. It gives memory usage for each line in MiB 'Mebibyte' size converted to MB 'Megabyte' size. For the memoization table's size, as the memoization is a dictionary type which contains the paths found using the Dijkstra's algorithm. We stored the dictionary's data in a JSON format file using python's json package dump function. The data is stored in the string format in JSON file as we can easily find the string size. Nvidia's GPU CUDA memory usage is captured using PyTorch built-in function $cuda.mem\_get\_info$ [18].

6.2.3 Environmental Impact

Another important metric used to gauge our algorithms is the environmental impact of computation. Computations like dataset generation, model training, and model inferences for larger graphs take a significant amount of time (spanning from 30 minutes to hours). To evaluate and compare algorithms based on their environmental impact, carbon dioxide emissions due to the consumption of electricity were tracked. CO_2-eq[19] is a global standard to measure the impact of carbon dioxide in the atmosphere, known as Global Warming Potential (GWP). Thus, we use kilograms of CO_2 equivalents (CO_2-eq) per kilowatt-hour.

CO_2-eq is calculated as a product of two variables C and P, where C is the carbon intensity (in CO2 equivalent) and P is power consumption of the infrastructure (in kilowatt-hours).

$$CO_2eq = C * P \tag{13}$$

An energy grid pulls electricity generated through different sources like fossil fuel, renewable, natural gas, etc. for distribution, known as Energy Mix. Carbon intensity (CI or C) of electricity consumed is the weighted average w of the emissions of each energy source s in the energy mix. Each source is associated with a specific value of C. Calculating weighted averages allows us to track the contribution of a particular source in the energy mix. For example, carbon intensity of an energy mix with 10 per cent fossil fuel source and 90 per cent renewable source will be much lower than an energy mix where ratios are the opposite. Each source is associated with a specific value of C. Hence, net carbon

[17] `https://pypi.org/project/memory-profiler/`
[18] `https://pytorch.org/docs/stable/generated/torch.cuda.mem_get_info.html`
[19] `https://co2.myclimate.org/de/offset_further_emissions`

intensity can be calculated as follows where all the energy sources in the energy mix are $s_1, s_2, ...s_n$ multiplied by their corresponding weighted averages denoted as $w_1, w_2, ...w_n$

$$C = s_1w_1 + s_2w_2 + ... + s_nw_n \tag{14}$$

If the global carbon intensity for a region is not available, calculations are made using the energy mix values. If both are missing, then the calculations are made using the global average carbon intensity.

Power consumed by the hardware P is calculated at 15 seconds interval for the processes associated with the algorithm, on both CPU and GPU. If the tools to record the power consumption are not available, the calculations are made by detecting the CPU or GPU models and then comparing them with manufacturer's data listing for thermal design powers (TPD) for the specific model. If the CPU model is not found in the manufacturer's data, then a global constant is applied.

The above-mentioned method allows us to individually track an algorithm process for its CO_2 emissions irrespective of the time taken for the computations.

6.2.4 Path Length

In the context of utilizing GNNs for shortest path finding, the path length can serve as a metric for evaluating the performance of the GNN. The path length is defined as the total distance along the path from the source node to the destination node.

To utilize path length as a metric, it is necessary to first define a distance function that assigns a cost or distance to each edge in the graph. This distance function could be based on the physical distance between nodes or could incorporate other factors such as the time required to traverse the edge or the cost of utilizing the edge.

Once the distance function has been defined for the problem at hand, it can be utilized to compute the path length of the shortest path identified by the GNN through the use of estimation. This can be achieved by summing up the distances of all edges in the path.

The path length of the GNN's solution can then be compared to the true shortest path length to evaluate the performance of the GNN. If the path length of the GNN's solution is close to the true shortest path length, it can be considered a good solution. Conversely, if the path length of the GNN's solution differs significantly from the true shortest path length, it may suggest that the GNN is not performing effectively.

In summary, path length serves as a valuable evaluation metric for GNNs used for shortest path finding, as it allows for the quantitative measurement of the accuracy of the GNN's solutions and their comparison to the true shortest path.

6.2.5 Percentage of Path Found

In the context of shortest path problem estimation using GNNs, the percentage of shortest paths found serves as a useful evaluation metric for analyzing the results. This metric represents the percentage of correct shortest paths identified by the model out of the total number of shortest paths in the dataset. It is calculated as the ratio of the number of nodes in the estimated shortest path to the number of nodes in the true shortest path. The equation for the percentage path found is:

$$\text{percentage path found} = \frac{\text{number of correct nodes in estimated SP}}{\text{number of nodes in true SP}} * 100 \tag{15}$$

This metric can range from 0% to 100%, with higher values indicating more accurate shortest path estimates.

To calculate this metric, it is necessary first to determine the true shortest paths in the dataset, which can be done through the use of ground truth labels or a reference shortest path algorithm. The predicted shortest paths produced by the graph neural network can then be compared to the true shortest paths, and the percentage of correct predictions can be calculated as depicted in Equation 15.

There are several advantages to using the percentage of shortest paths found as an evaluation metric. It is a straightforward and easily interpretable measure that can be effectively communicated to a broad audience. Further, this is directly relevant to the performance of the model on the task of shortest path estimation, making it a relevant and meaningful metric for this problem.

6.3 Experimental Setup

6.3.1 Overview

Over the course of this project, we have come up with many experiments. For every idea, there can be a number of slightly different versions of the same experiment that portrays the same question from different perspectives. While we would have ideally run, rigorous experiments for each small variation the rhythm of development combined with the frequency of new ideas have resulted in two primary ways of running experiments.

- **Preliminary Experiments**. Used during the development and verification of our implementations. Those experiments were regarded as secondary as at the time of running them the main objective was to establish the validity of a certain method. Those experiments serve as a base for the hypothesis we made and give us an intuition for the direction of the rigorous experiments.
- **Final Experiments**. The results of those experiments have been included in the report and where done after we have settled on all the models that we intend to compare.

6.3.2 Validation and Max Probability Walk

In Table 8 we have the validation F1 scores for nodes and edges. In Table 9 we show the base max probability walk results. Keep in mind that max probability walk results show paths that reach the end. As previously mentioned the loss function is not properly describing the problem as it does not take into account that a path is incomplete. This means that models that get good training or validation scores are not necessarily the best at finding paths.

6.3.3 Increasing Size Experiments

Training on large datasets takes a long time. Given that we have to also compute the ground truth for those training samples, we are very interested in how the GNNs predictions scale.

Because we are interested in the scaling capabilities, we consider having good results on these increasing tests is a good indication of how the models can scale.

6.3.4 Increasing Size Data Sets Generation and Description

In order to ensure that the datasets generated accurately represent the behaviors exhibited by the networks we generate a number of overlapping datasets where the number of graphs inside each dataset remains stable while the minimum number of nodes continually increases up to a certain threshold. This approach guarantees that we can see the underlying pattern of the scaling capabilities of the GNNs.

6.3.5 Runtime Experiments

Data was created at set intervals with graphs having a given range of nodes. The graphs used a boxplot to better see any overlaps between the data sets. Two sets of graphs were completed: comparing the MPNN to the different Dijkstra-memoization algorithms as well as comparing the different memoization versions to the Dijkstra shortest path algorithm.

The dataset range was between 10-1000 nodes in the graph, 20 graphs per set and a 10 node range for the graphs.

6.3.6 Memory Consumption Experiments

Datasets with multiple graph sizes have been used to analyze the GNN and Dijkstra memory consumption. The different line graphs and block graphs have been used to show the consumption increase or decrease for the different size datasets. Two sets of data were used as shown in Table 7. The finding of memory consumption has been done over: The memoization table, Dijkstra with memoization, and the other GNN models.

	Dataset Range (nodes)	Nodes in between each set	Graphs per set	Node Range
set 1	10-990	100	20	15
set 2	990-1010	NA	20	10

Table 7: Dataset used for Memory Consumption Experiment

7 Results

7.1 Training

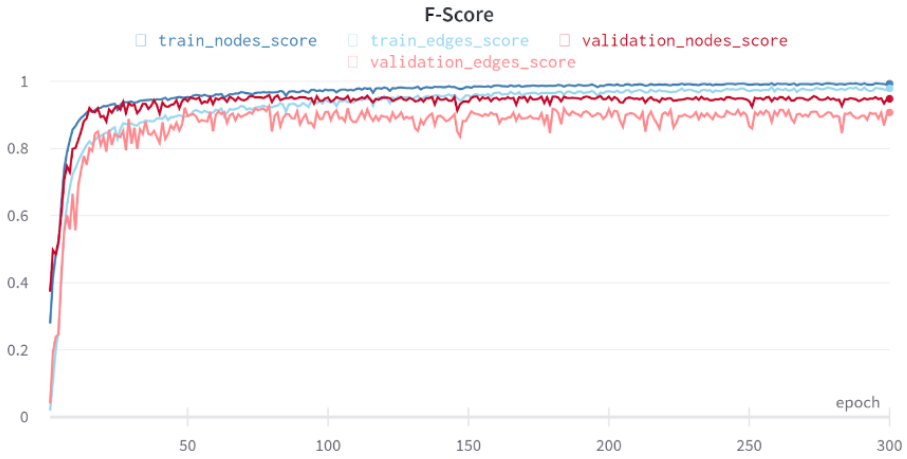


Figure 16: MPNN Training performance, 50 - 100 nodes per graph, 3584 Training graphs, 1536 Validation graphs. With hyper-parameters: Learning rate = 1e-4, Batch size = 16, Encoder layers = 2, Processor layers = 12, Decoder layers = 2.

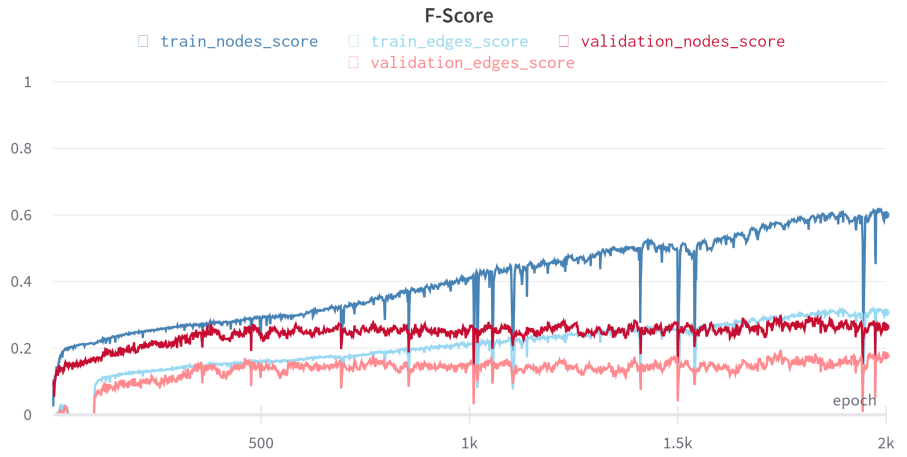


Figure 17: MPNN Training performance, 1000 - 2000 nodes per graph, 3584 Training graphs, 1536 Validation graphs. With hyper-parameters: Learning rate = 1e-4, Batch size = 16, Encoder layers = 2, Processor layers = 12, Decoder layers = 2.

Figure 16 shows how the F-Score changes during training on 50-100 nodes graphs. The model converged after around 150 epoch. On the other hand, Figure 17 shows the result for larger 1000-2000 nodes graphs. The training time took significantly longer and required many more epochs. The training did not converge even after 2000 epochs. This indicates that the bigger the graph size, the more training time and epochs are needed. However, the validation score is likely to be the same or only increased slightly even when the training score reaches 1. This also shows that the model cannot learn big graphs well.

7.2 Runtime Experiment Results

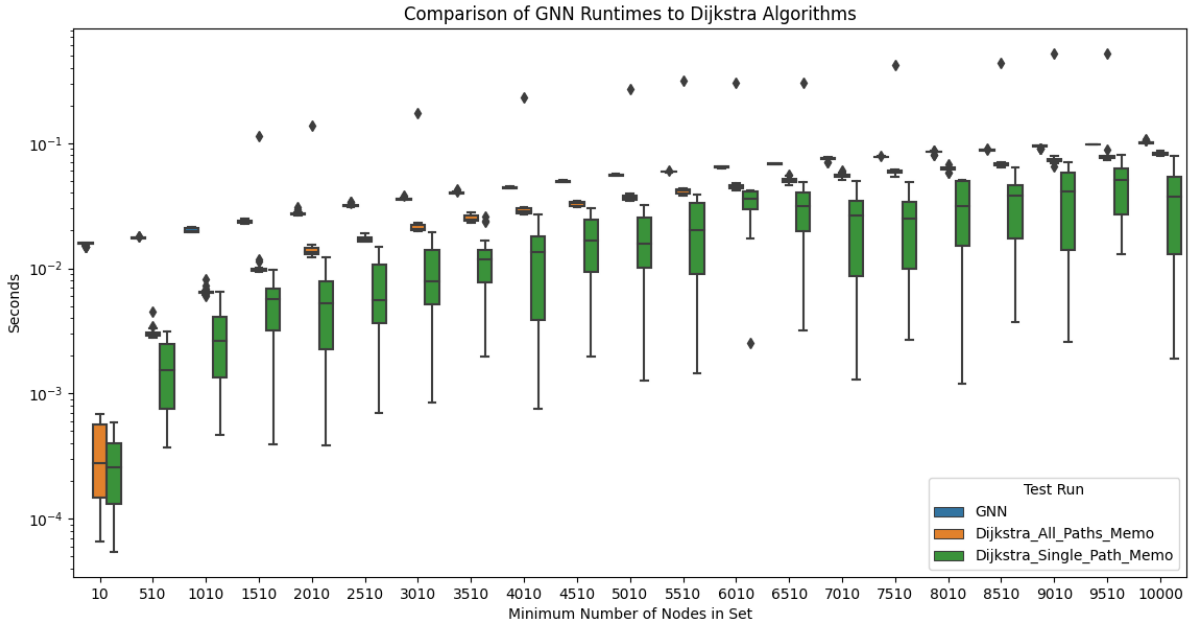


Figure 18: Comparing algorithms for 20 node sets

Figure 18 shows that the runtime speed increases faster for the two Dijkstra's methods than for the GNN. Figure 19 shows that all three Dijkstra's method runtimes increase at the same rate. The fastest is Dijkstra's without memoization, however, it's a minimal difference. The Dijkstra's algorithm that finds all paths has a much lower spread and does have some overlap with the upper range of the other two implementations.

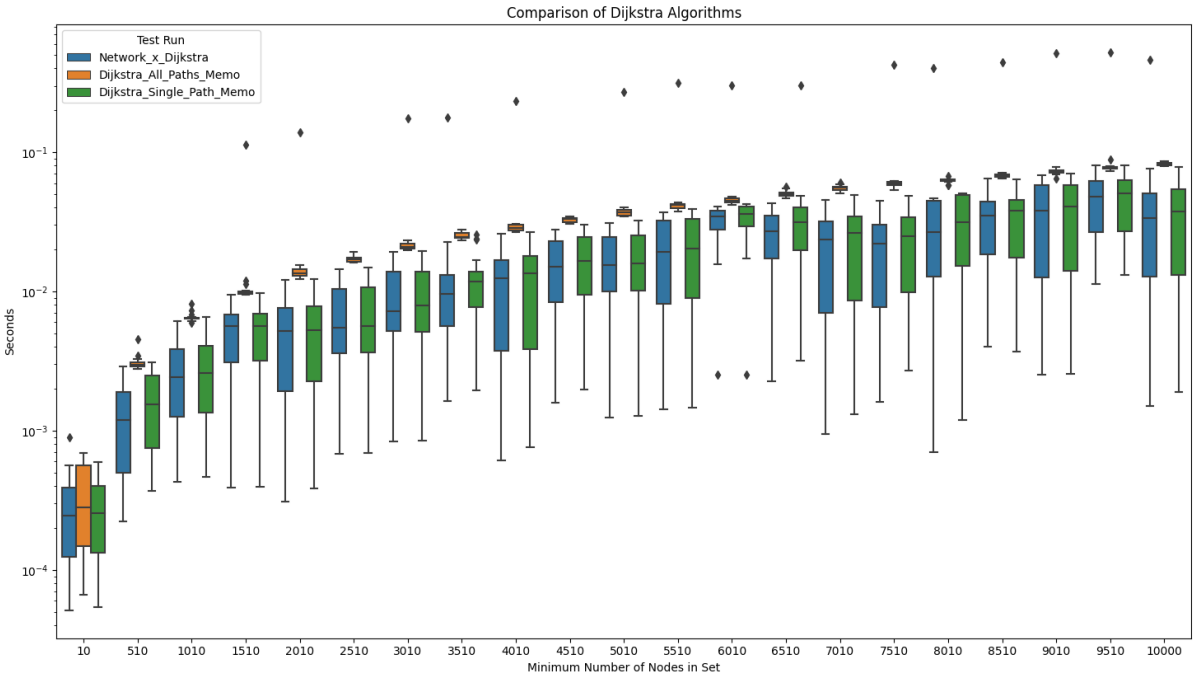


Figure 19: Dijkstra's algorithm comparisons with 20 node sets

7.3 Memory Consumption Experiments

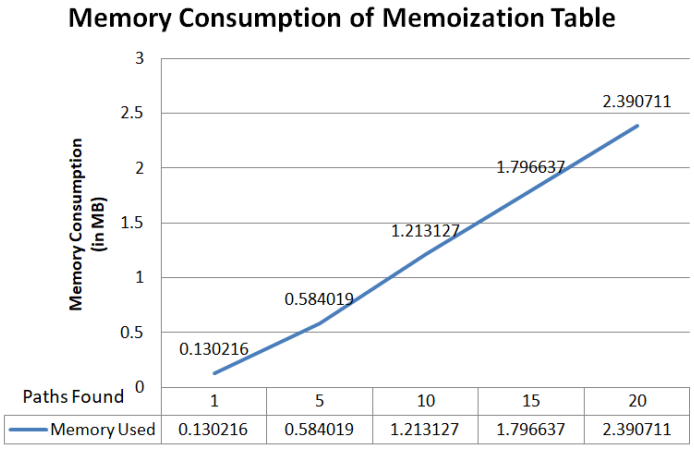


Figure 20: Difference in memory consumption of DFPS to DSPS memoization table. Used increasing size dataset of 990-1005, number of graphs = 20.

Figure 20 shows a small linear increase in memory consumption as the new shortest path keeps adding to the memoization table. This also results in stating that no memory consumption will be done by the memoization table once all the shortest path has been found of the dataset graphs and have been added to the memoization table. Also, the results show better scaling for the difference in memory consumption of the DFPS memoization table and to DSPS memoization table.

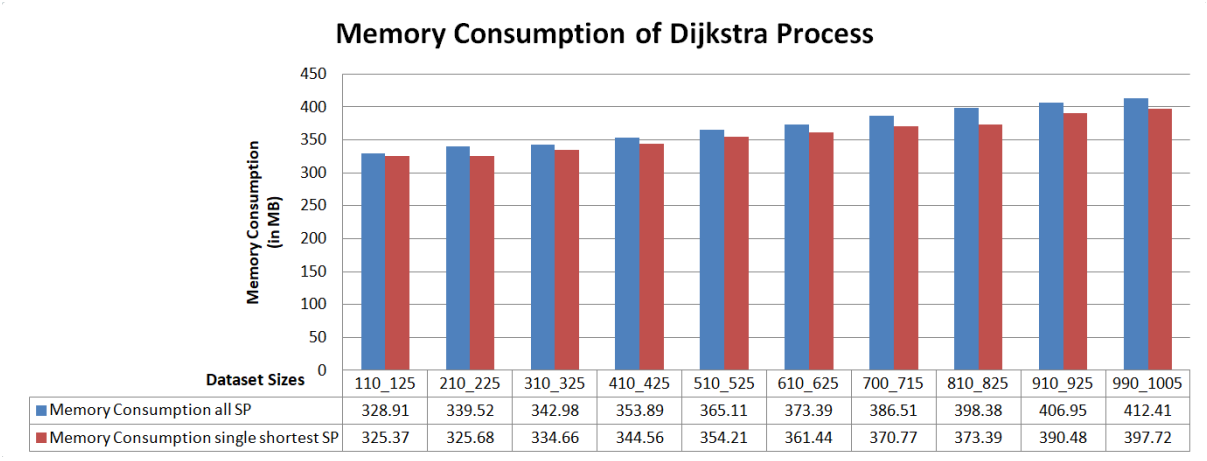


Figure 21: Memory consumption of Dijkstra's process with memoization

Figure 21 shows an increase in memory as per the dataset size increase which means a large dataset can contain longer/multiple shortest paths, increasing the memory consumption of the memoization table and the resulting increase in memory of the whole process.

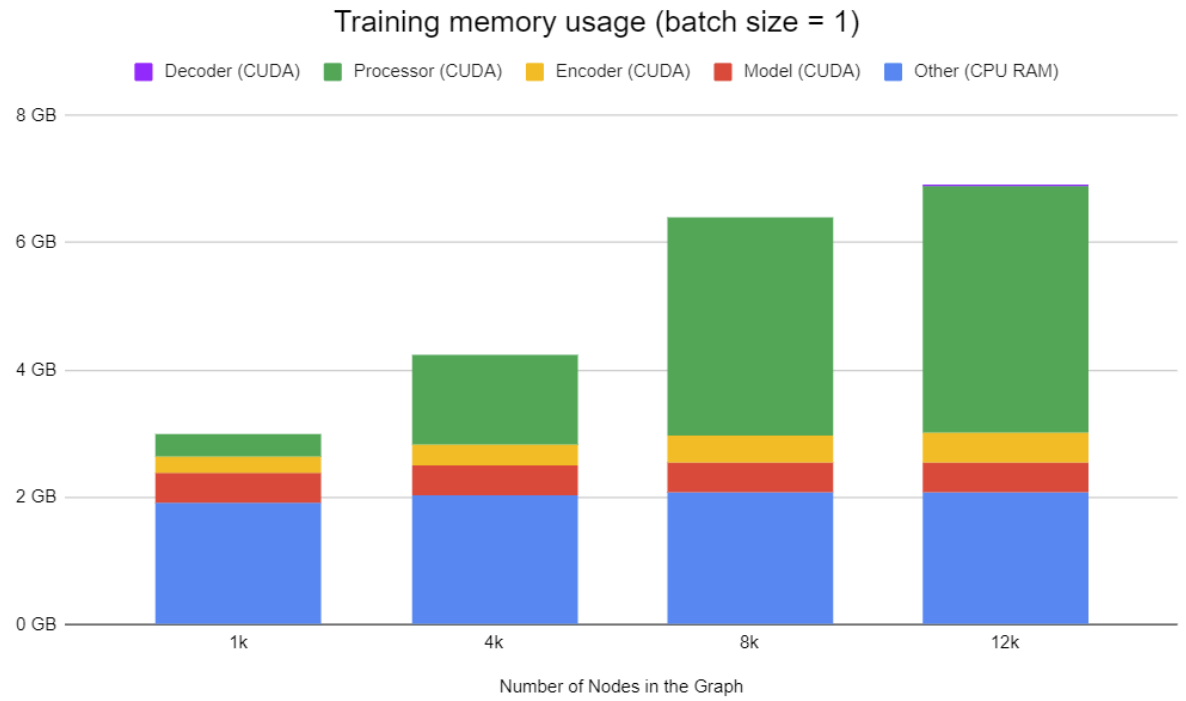


Figure 22: MPNN Training memory usage, With hyper-parameters: Batch size = 1, Encoder layers = 2, Processor layers = 12, Decoder layers = 2.

Figure 22 shows the memory usage while training on different dataset sizes. The blue section is the memory consumption on the main RAM with the model and one graph data loaded. Since we are doing calculations on the GPU, the other four sections show the memory consumption on Nvidia CUDA Memory as well. There is a steady increase in all sections, most notably, the *Processor* layer. The result tells us that with limited CUDA memory size in modern GPUs, there is a hard limit on the dataset we can train on.

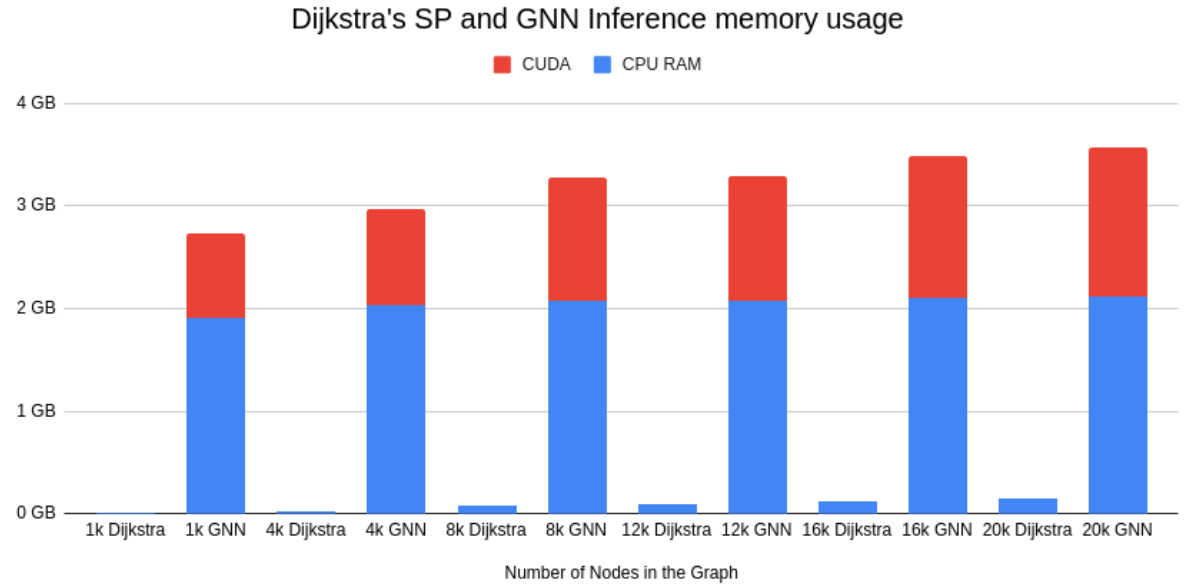


Figure 23: Dijkstra's SP and MPNN Inference memory usage. GNN Config: Encoder layers = 2, Processor layers = 12, Decoder layers = 2.

Figure 23 shows the memory usage for inference. The increase in memory consumption is significantly lower as it does not calculate and store intermediate gradients. Overall, it is still considerably higher than Dijkstra's SP algorithm.

7.4 Bidirectional model

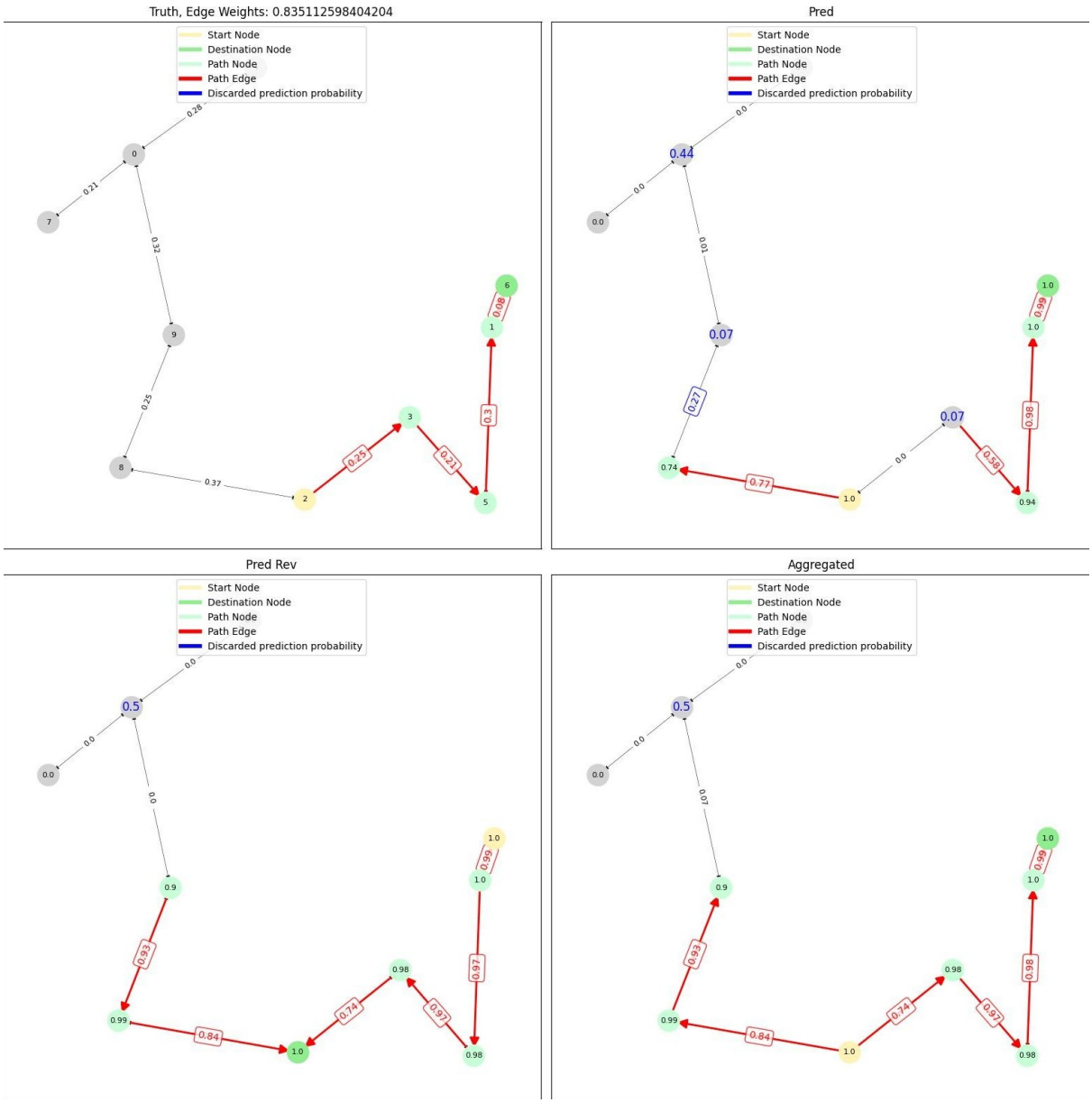


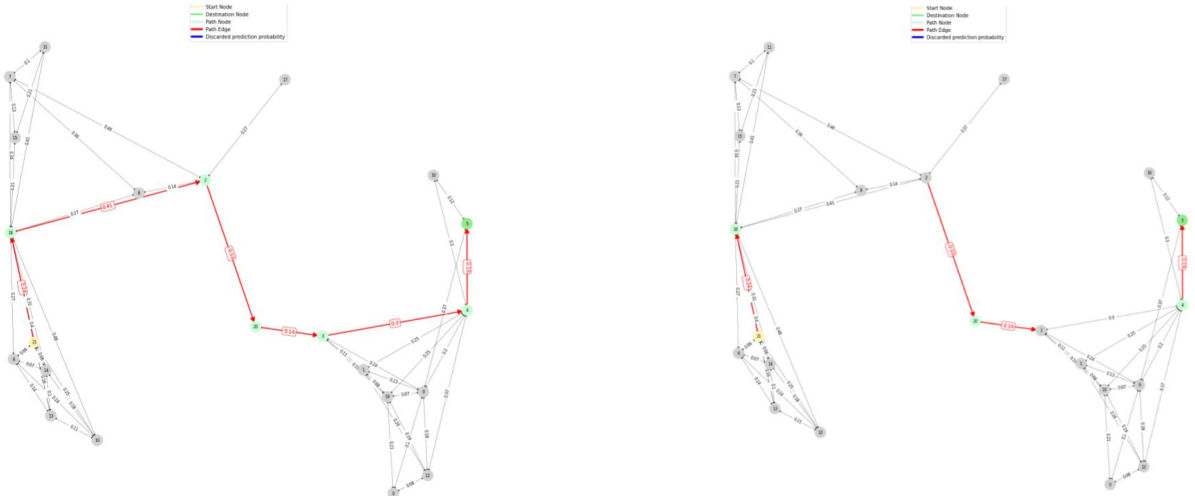
Figure 24: Bidirectional inference demo

The upper left sub-figure shows the ground truth with the total edge weights of around 0.83. The upper-right sub-figure shows the inference result when predicting from start to end node while the bottom-left shows the inference result when predicting from end to start. Finally, the bottom-right sub-figure shows the aggregated result of both start-to-end and end-to-start predictions. The results in Figure 24 show that the model increases the overall output probabilities and improves the chance of predicting edges and nodes to being in-path. However, this could be a non-desirable outcome, as more false positives

lead to more branching paths. For all the inference results, the values on the nodes and edges show the GNN output probabilities.

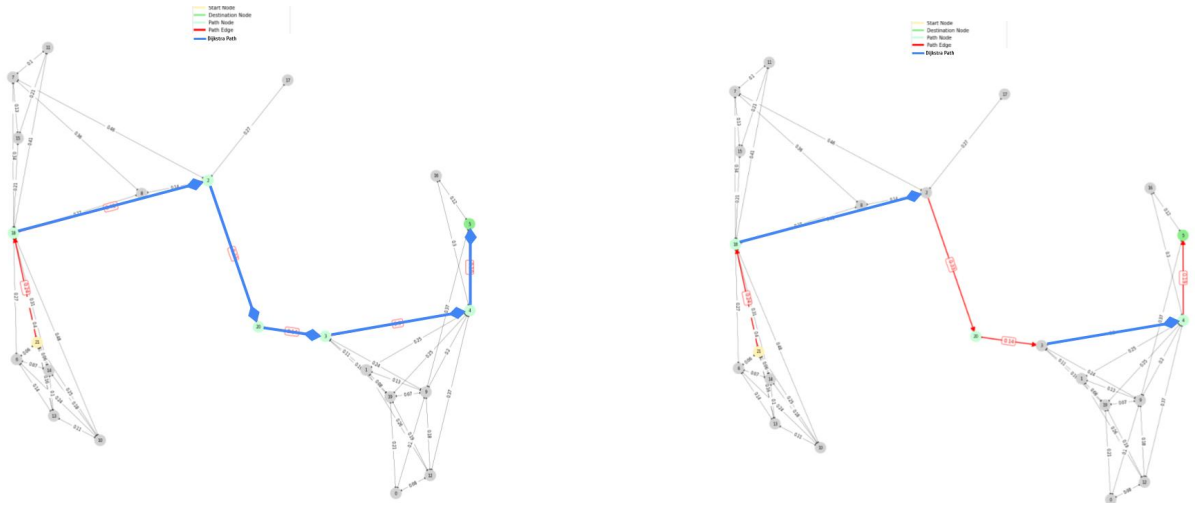
7.5 Hybrid Model

As seen in Figure 10, the hybrid model completes the path by evaluating the path and detecting disconnection at nodes.



(a) Ground truth (b) MPNN

Figure 25: Comparison: GNN vs Dijkstra



(a) Hybrid: current implementation (b) Hybrid: suggested improvement

Figure 26: Suggested improvements in Hybrid implementation

Dijkstra's algorithm is used to complete the rest of the path, but discards any subsequent path fragments that were already predicted by the GNN models. While the hybrid model ensures path completion, it does not optimally utilize GNN results. This can be seen in Figure 25.

7.6 Accuracy Results

	Small		Medium		Large	
	F1 node	F1 edge	F1 node	F1 edge	F1 node	F1 edge
MPNN	0.9477	0.9137	0.6771	0.3122	0.4442	0
A MPNN	0.9785	**0.9587**	0.6511	0.2754	0.4943	0
GCN	0.8677	0.8103	0.671	0.082 21	**0.5354**	0
GAT	0.873	0.816	0.4403	0	TNF	TNF
MPNN W.	**0.9867**	0.9455	**0.8305**	**0.454**	TNF	TNF
MPNN E.	0.489	0	0	0	TNF	TNF

Table 8: Models training F1 scores for the validation sets. The table presents both F1 scores for the edges and nodes. Additionally, in the cases where the *training did not finish* the "TNF" value is given. This means that the model took longer than 48 hours to train. Each model has been trained and tested on datasets of different sizes. In this case, the sizes are Small 10-100, Medium 200-300

	Small			Medium		
	1%	25%	50%	1%	25%	50%
MPNN	0.92	0.90	0.89	0.71	0.60	**0.41**
Adapt MPNN	0.79	0.73	0.62	0.26	0.26	0.15
GAT	0.82	0.76	0.58	TNF	TNF	TNF
MPNN W.	**0.95**	**0.94**	**0.92**	**0.78**	**0.79**	0.39
MPNN E.	0.26	0.10	0.04	0.00	0.00	0.00

Table 9: Model scores in post-training for the base max probability walk. The TNF values stand for *training not finished*, in these situations the training took more than 48 hours and it did not finish. The dataset sizes are 10-100 nodes for the small dataset and 200-300 for the medium. The percentages stand for the minimum probability treshold for nodes and edges. The results represent the percentage of paths found.

7.6.1 Training and Max Probability Walk Results

The key take away is that the training metric does not directly translate to a good result. The model can have good accuracy on 99% of the nodes part of the path, however if a single node is miss-classified then the path is incomplete and we can not reach the end node. As expected the weighted loss does tackle some of the class imbalance inherently present. This can be seen in Table 10 where the the weighted model is the all out winner.

We can see that our implementation of the Euclidean distance was hindered by a likely implementation bug that caused it to have 0% accuracy which is likely impossible as the start and end node are very easy to classify correctly, given that they are unique and always part of the path.

7.7 Increasing Graph Size Experiment

From the previous results we have observed that the best results were achieved by the base and the weighted loss. In this section, we will explore the scaling capabilities of the models trained on the small datasets.

7.7.1 Base Model

Figure 27 describes the different accuracy results for finding the path on graphs of increasing sizes with a model trained on the base model. The path is calculates using the "max probability walk" method for decreasing probability thresholds. This model is trained on a dataset of graph sizes containing 10 - 100 nodes.

Figure 27a shows how much percentage of the entire path was found. While hard to observe, the results gradually degrade until it reaches 0% paths found at 1600 nodes. In terms of probability threshold the pattern is that as we lower the confidence threshold, we can get more paths.

Figure 27b shows the ratio between the ground truth and the complete, predicted paths. It seems that when a path is found, this path is not much worse then the path found by Dijkstra.

Figure 27c describes the actual values of the max probability walk for the base model. We can observe how the models does sometimes produce paths up until 1300 nodes.

7.7.2 Weighted Loss Model

Figure 28 describes the different accuracy results for finding the path on graphs of increasing sizes with a model trained on the weighted loss model. The path is calculates using the "max probability walk" method for decreasing probability thresholds. This model is trained on a dataset of graph sizes containing 10 - 100 nodes.

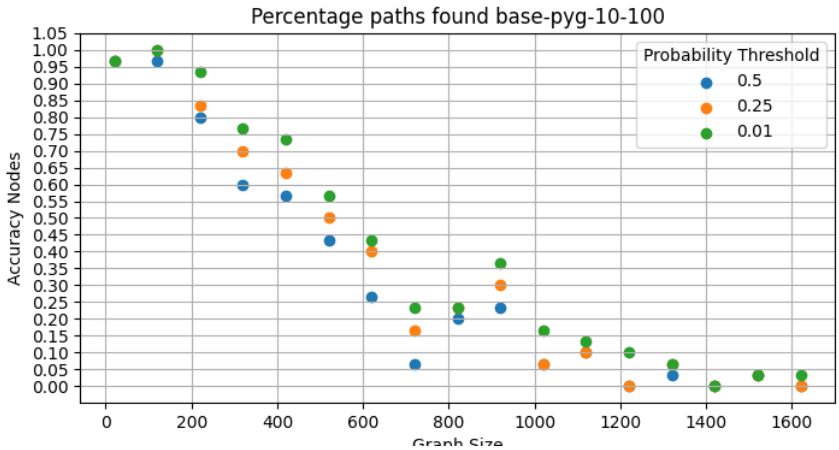
Figure 28a shows how much percentage of the entire path was found. We can see that the accuracy of the predictions were higher than that of the base model.

Figure 28b shows the ratio between the ground truth length and the prediction. This is only computed for complete paths. When a path is found, it seems that this path is not much worse then the one found using the Dijkstra algorithm.

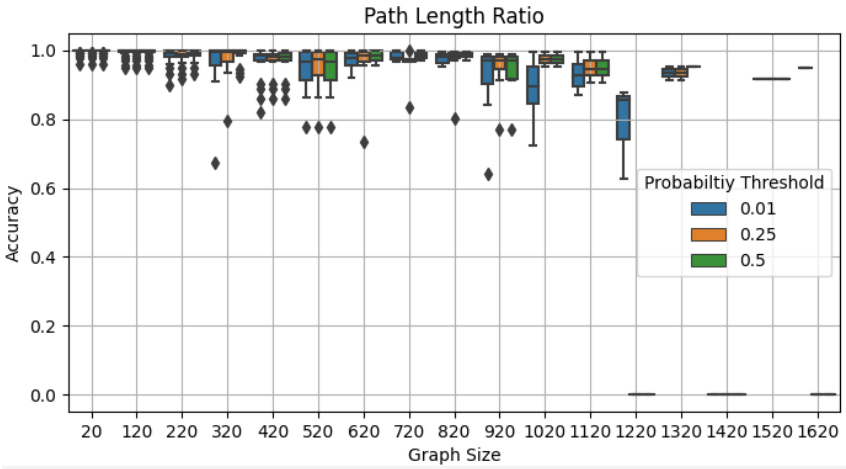
Figure 28c describes the actual values of the max probability walk for the weighted model. For this weighted model, the results are slightly better than the results of the base model. While the score of 0 is reached at about the same number of nodes, the decay in results is slower.

7.7.3 Increasing Size Experiment Results Discussion

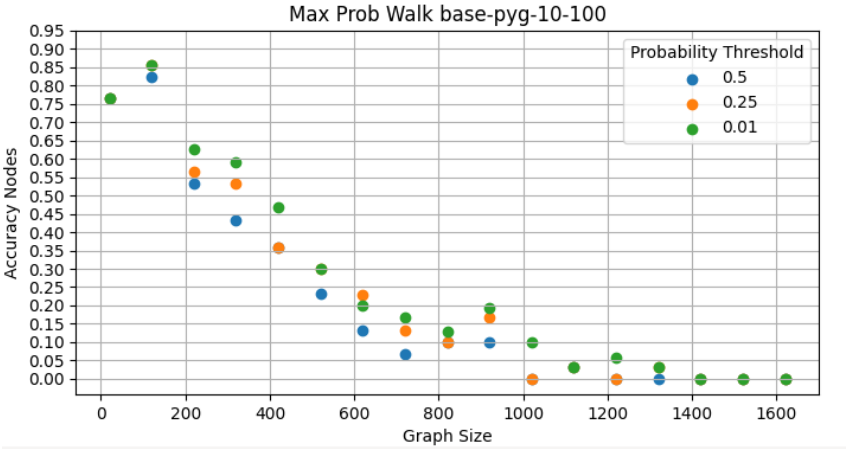
While not really surprising tackling the class imbalance by trying to add more weight to the *in_path* class that improves the results. We observe the model where the loss function was weighted performs slightly better than the base one. One interesting point is that



(a) Percentage of the paths found through the max probability walk.

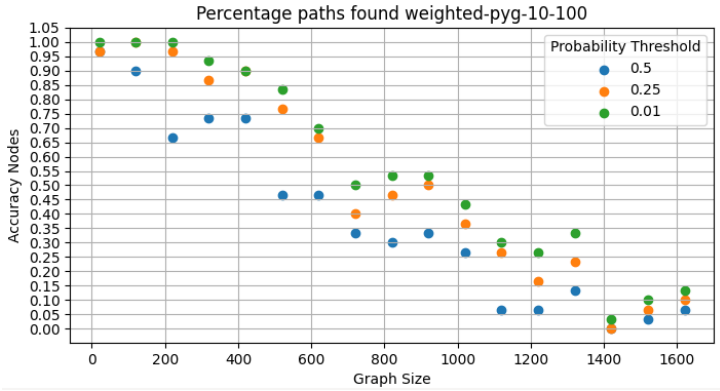


(b) Ratio between the ground truth and complete predicted paths.

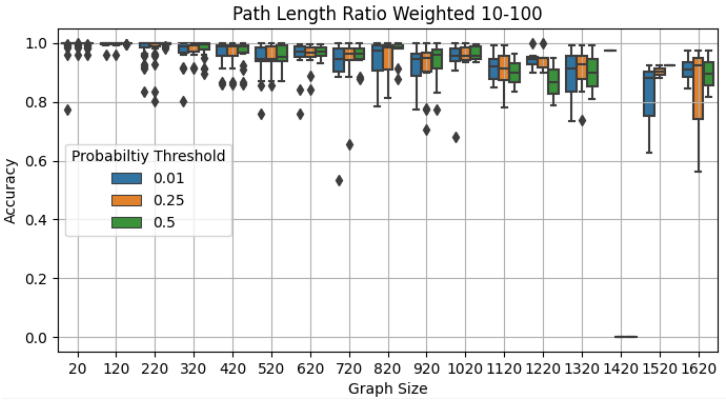


(c) Max probability walk values for the base model.

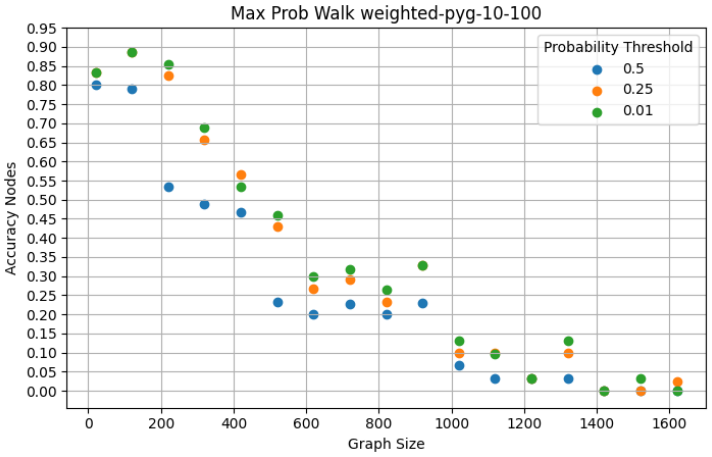
Figure 27: Accuracy results for base model



(a) Percentage of the paths found through the max probability walk.



(b) Ratio between the ground truth length and the prediction.



(c) Max probability walk values for the weighted model.

Figure 28: Accuracy results for weighted loss model

when a path is found this path is usually similar in length compared to the ground truth. This can be seen in the *b)* tables of both models.

7.8 CO2 Emissions results

The CO_2 emissions for the selected runs were used for final analysis.[20]. In the following figures we compared the best model - MPNN and its variations. Each model was trained with three different modes, namely adaptive MPNN, Euclidean distance and weighted average. We also trained each variation of MPNN model with three different graph sizes and classified them as small- 10 to 100 nodes, medium - 200 to 300 nodes and large- 400 to 500 nodes.

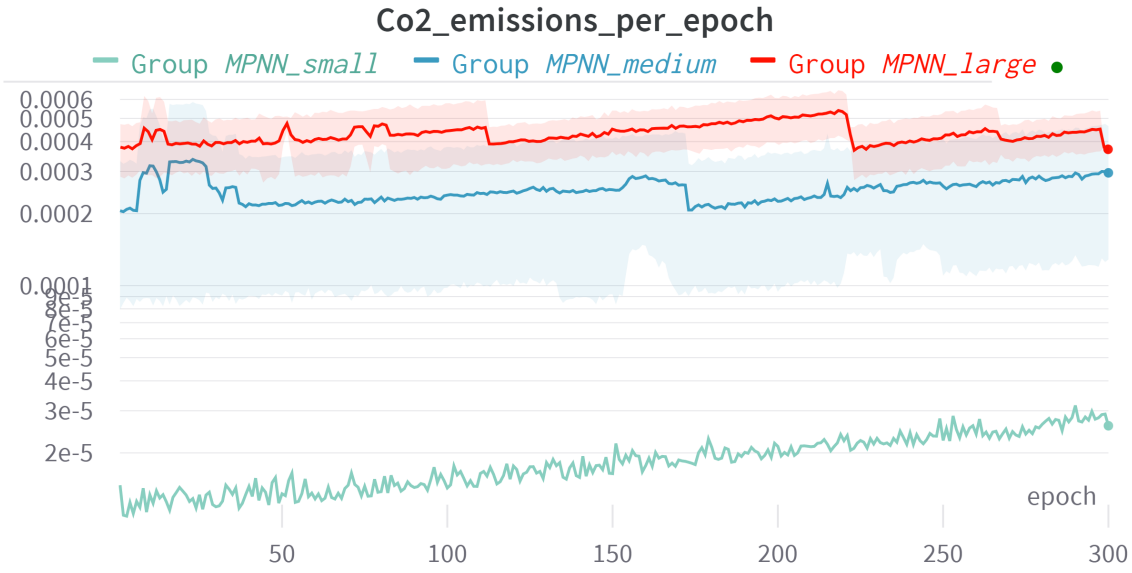


Figure 29: Emissions per epoch

In 29, it is observed that training MPNN with large dataset has the highest emissions. However, emissions for medium model's training exceed large model's emissions in the first few epochs. Overall, there is less difference in emissions for these two models as compared with training of the small model.

[20] `https://api.wandb.ai/report/uu-spreadnet/30knh9br`

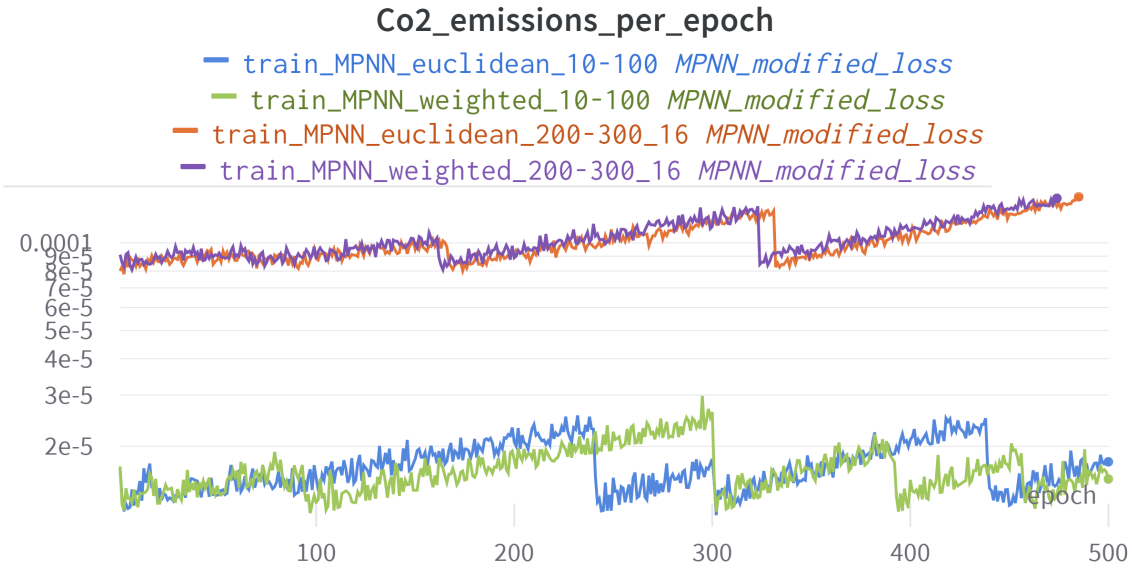


Figure 30: Emissions per epoch

If we consider modified loss model variants 30, we see similar trend of large and medium models having larger carbon footprint than small models. However, the difference between Euclidean distance and weighted average models remains minimal.

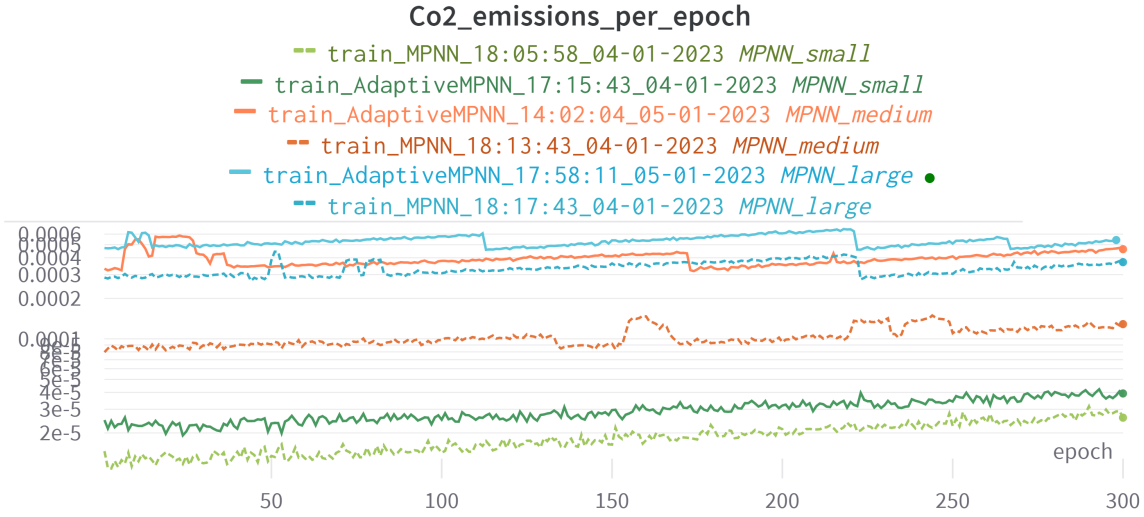


Figure 31: Emissions per epoch

Finally, we compare base MPNN model with adaptive MPNN model for small, medium and large training dataset sizes in figure 31. Adaptive MPNN has higher emissions than base MPNN model, in fact higher than base MPNN model trained on large dataset. The difference between base MPNN and adaptive MPNN is most significant when models were trained on a medium training dataset.

8 Related work

This section aims to reflect on related research and findings that led to this problem being interesting to investigate and could be related to our work. Section 8.1 describes the prior work done in relation to the solutions explored for the shortest path problem and the shortcomings encountered with them. Section 8.2 explains some comparable work done with GNNs and explains how those findings are interesting.

8.1 Shortest Path

The shortest path problem could be categorized based on several features, as explored by Madkour et al, including but not limited to static, time-dependent, parametric, stochastic, dynamic, weighted region, related path and alternative path [10]. The traditional methods using the classical approach are the ones most commonly used to solve this problem, to date.

Given its vast usage, designing new and more efficient shortest path problem solutions is seen to be an unmistakable need and is explored in reference to key metrics including fast computing, dynamic/runtime reconfigurability, low memory consumption, accuracy, and robustness [20].

Dijkstra's and the Bellman-Ford Algorithm are two well-known algorithms for solving shortest path problems [20]. Dijkstra's algorithm [1] is a greedy algorithm that uses positive weights, whereas Bellman-Ford [21] has been used in cases where the graph contains cycles. However, the shortcoming experienced with them is the inability to handle negative weighted edges and have only been used with additive linear path cost models [20]. Further, the issues faced with these approaches were also observed with respect to time inefficiency and convergence issue with larger graphs.

Besides the commonly known classical approaches, heuristic algorithm approaches were also used to solve the shortest path problem and did gain a lot of traction. A* [22], another reasonable algorithm for our problem type, attempts to find a path by applying a heuristic function [23]. Floyd-Warshall is the algorithm to find the fastest path and the shortest distance between two nodes [24], while the program is intended to find the path of more than two nodes in the path, which is an all pairs shortest path approach and not something we have explored in or research. Although, using heuristics significantly had an impact on the shortest-path solutions the quality of the solution depends on the type of heuristic applied to the problem. These also faced issues with inefficient memory usage accompanied by increased computational requirements, thus proving them to be cost-inefficient in several cases, to solve the SP problem [22].

The limitations explored with the classic algorithms for this problem of finding the SP made the search for better solutions highlight the usage of Artificial Intelligence for the domain. Although initially explored for fairly simple and straightforward algorithms like Genetic Algorithms (GA), Particle Swarm Optimisation (PSO) [25], Hopfield Neural Networks [26] and Dynamic Neural Networks [27] as explored in [26], [27], [20], [25], there were still some limitations observed. Although they were able to converge to an optimal path, they were computationally expensive and prone to invalid path detection.

With the increased interest in exploring ML based estimation methods for the purpose of generating the shortest path, there haven't been approaches clearly defined that used solely this method for the prediction. They have been supported by some other algorithms and used to calculate the distances and then use ML to identify the shortest path. One

such example has been as follows: Previously, the use of MLP to generate embeddings using the vdist2vec[21] method was adopted and aided the process of identifying the shortest path. The use of estimation was not carried out in this method, in its entirety, however, the main focus was to establish the distances between the nodes when accompanied by landmark detection and adding embedding for the vertices as a distance label [28].

8.2 GNN

Previous research has developed GNN architectures that mimic the individual steps of specific graph algorithms, such as breadth-first search, Bellman-Ford, and Prim's algorithm [29]. Although not specifically tied to our task, they have been seen to be applied to existing graph-based problems. However, they have also been observed to tackle some combinatorial optimization problems that have shown interesting results. The paper [30] discusses applying GNNs to solve combinatorial optimization problems. The problem types however were NP-hard type which included the minimum vertex cover, minimum cut, and maximal independent set amongst those solved using the GNNs. Further, this paper also went ahead to compare the model's performance to the Boppana-Halldorsson (BH) [11] approach to solve the problem of the maximal independent set (MIS). The results proposed were compared based on runtime scaling as a parameter in comparison to the number of nodes in the graph used for execution. Further, the BH algorithm was used with two possible degrees 3 and 5. For nodes $n < 500$ the runtime scaling was $n^{2.9}$ for the BH approach and with graphs of size $n \sim 10^5$ when run using GNNs had a runtime scaling of $n^{1.7}$. This showed that GNNs are able to perform much better, if not the same while solving the combinatorial optimization problem of MIS. The runtime scaling also included the post-processing times.

As intriguing as the results obtained from the comparison above [30] were, they were contested and based on some further experiments were proposed to be not true at all times. GNNs were compared to Degree Based Greedy Algorithms for solving the combinatorial optimization problem of MIS [31]. However, the results obtained with the same degrees as above (3 and 5) for the greedy algorithm used, were compared with GNNs based on qualitative criteria (Approximation Ratio) and time criteria (runtime including post-processing time). The results obtained from the experiments, based on the approach followed in the paper, showed that Degree Based Greedy Algorithms performed much better than the GNNs with up to 6 percent better results in terms of the approximation ratio and showed a speed-up in the runtime around a degree of 10^4, for graphs with the number of nodes being close to 10^6. Based on these results, the authors of this paper suggested that GNNs have not yet been able to surpass the performance of greedy algorithms on some benchmark tests.

Although, GNNs are still a field of extensive research to solve multiple types of problems now. There have been developments in the algorithm that have features, like those described in this paper, such that it has been an interesting application to solve this problem. Some such variations have been interesting to the problem, but are not limited to GAT, Graph Auto-Encoders [32], Graph Generative Networks [33], and Graph Spatio-temporal Networks [34]. GNNs will continue to expand into new domains as newer architectures emerge.

[21] `https://rdrr.io/github/jfq3/QsNullModels/man/dist2vec.html`

9 Discussion

9.1 Accuracy Analysis

We can not deduce from the F1 score whether we can actually find the shortest path in the graph, but a high F1 score indicates that we have a high probability of successfully constructing the shortest path based on the outputs.

All models show worse accuracy as the dataset increases. MPNN and Adaptive MPNN show better performance in the F1 score compared with other models when we use small and medium datasets. GCN overtakes them as the most performances model when we use the large dataset, although large graphs apply a very large penalty that impacts the learning.

Adaptive MPNN reuses the existing MPNN blocks, the information can be better extracted, and messages can be propagated sufficiently in the network. The reason why Adaptive MPNN shows a better performance than MPNN in the small dataset but not in the medium dataset is that we reduce the number of basic MPNNs because of the limitation of GPU, which memory can not support us to have the same message-passing steps as MPNN. We assume that if we train our AdaptiveMPNN with enough large number of basic MPNNs, we can get better results in all datasets.

MPNN with the weighted loss function can have an overall better performance than other models. The weighted loss is more reasonable than the common-used cross-entropy loss function. It gives a higher penalty on the particular nodes or edges which have been incorrectly classified to be "not in path."

9.2 Runtime Analysis

The runtime of DSPS compared to DFPS is much lower due to the lower amount of nodes it needs to check. However, the only information that can be extrapolated from this data is the given shortest path, and its subpaths, nothing else can be identified as the shortest path due to the stopping point of the algorithm. This makes it a situation on whether the DFPS or DSPS would be a better method to save. One problem with this is that there can be a longer running time, particularly noticeable if the destination node is very close to the source node, the DFPS will still visit all nodes in the graph. On the other hand, if the shortest path is a very long path, there could be minimal extra runtime required to complete the search. The biggest benefit to this can be the amount of information brought back for further searches.

9.3 Memory Consumption of Dijkstra's Algorithm with Memoization

The memoization table and Dijkstra's algorithm are two important components for evaluating memory consumption of dijkstra with memoization. The memory consumption of a memoization table is calculated once the shortest path in the table is added. Additionally, the length of shortest paths has no influence on memory usage. Given that we are storing the paths as hashes, the sizes of each shortest path will be uniform. More memory is used by DFPS than DSPS. Because DFPS looks for every shortest path and adds them to the memoization table where DSPS only looks for the single shortest path and adds it to the memoization table. The experiments' findings demonstrate that using

large datasets and shortest pathways discovered in them does raise the overall memory usage of dijkstra's process.

9.4 Performance Analysis in Comparison to Existed Work

As seen in the issues mentioned in the Related Work Section 8.2, there are many challenges with getting a GNN to be a better model than classic methods with many finding that GNNs are not as effective. There have been comparisons to degree-based greedy algorithms that used approximation ratio and run time to analyze and compare the performance of these models with that of a GNN. In this case, the GNNs were left far behind and did not match up to the performance of the greedy algorithms for an optimization problem. We also note that we were unable to find cases in regards to finding a shorter path. This however is not conclusive based on our report as there is still much more to be researched in the field of GNNs. These methods were also mostly able to scale their experiments to include graphs of much larger sizes with nodes up to the degree of 10^5. However, it was interesting to note the metrics used for evaluation in other approaches versus those in ours. For our models, we have captured the environmental impact and CO_2 emissions along with using memory consumption to analyze the performance. Further, we have also implemented hybrid approaches to overcome the issues and shortcomings faced with finding the shortest path solely using GNNs. Although, not entirely accurate this did lead to lead to improvements in our results, which were still less superior to those of greedy algorithms' results, as seen in earlier research.

10 Conclusion

The project ended not based on a full conclusion on GNN performance compared to a classical shortest path algorithm, but on time limitations due to the large amount of possibilities involved in this project. There were variables that were identified however they were not tested to completion. The testing was limited as well by the accuracy of the GNN. We didn't find the need to spend time and effort to analyze Dijkstra too far beyond where the GNN was showing low accuracy. For example, a clear understanding of the runtime was not understood as the data set to fully test graphs to a significant number, e.g. graphs of a million nodes was never assessed. The runtime comparisons also only compared one of the GNNs. This could also have been further expanded once there was better results from the GNNs. However, it does look like there may be a time when the GNN could bypass the speed of Dijkstra's shortest path algorithm. In finding a GNN with a high enough accuracy for that case, the GNN used may have a completely different build, which would then change the run time. The results are specific to the GNNs used and can't be broadly used beyond the idea that these comparisons should be done in order to better understand which method to use.

Having a focus earlier on to experiment with GNNs that used different strategies would have added value to our work. When it was noticed that the GNNs all were using similar methodologies, we didn't have enough time to implement and experiment with landmarks. This was also due to how our graphs were built and the data that we had to work with. There wasn't a simple way to try out methods that utilized heuristics.

There are many ways to identify why a GNN method may be better or worse than a classical method. Though, in this case, we weren't able to implement a GNN to outperform Dijkstra's algorithm variations given in this report, this topic is not concluded.

Furthermore, GNNs can be used for many types of path-finding besides the shortest path. In that way, there may be much less efficient classical methods for finding the solution. In these cases, utilizing the analysis types given in this research would be helpful.

Though not exclusive, we have identified strategies to help identify if a GNN can outperform the current method of solving a graph problem. By looking at the environmental aspects, accuracy, runtime and memory consumption, many aspects of the algorithms are evaluated.

11 Future Works

We have split this section into functional areas that we focused on in this project.

11.1 Non-GNN Implementations

The focus of the project was comparative between the GNN which didn't outperform the classical method we had chosen. Due to this, not much energy was focused on diversifying and experimenting further with the non-GNN implementations. However, there are many things that could be looked into further that could improve the non-GNN methods.

More classic algorithms can be tested in a more significant way to identify the best algorithm and memoization combination that would work with the specific data. There was no testing that compared graphs that had unique traits, for example, negative paths, that would change the non-GNN algorithm. Another possibility is to look into other algorithms, such as the Best-first search A* algorithm. Having a heuristic for the A* model could also be found to support the GNN models.

If there is a lot of data stored in the memoization table, there may be strategies to delete paths to lessen the required storage, or to store the data in a smaller way. In the current model, the path is stored for each start to end the shortest path that is found. For example, if a graph had a million nodes, there would be a million stored paths for each source node when fully visited. This would then be a million paths each with a million shortest paths. And this would be for just one graph. However, storing not the entire path but a pointer to the previous node that creates the path back to the source could save space though it would increase time, requiring more lookups (in a Python dictionary) but could save space.

One possible implementation is to use both methods of path collection with Dijkstra memoization. For example, if a graph has a given number of nodes, the user can decide what number of nodes need to be visited on the shortest path search for a complete minimum spanning tree search to be done. If the shortest path is small, for example, itself or only one or two nodes apart, returning only the single path without doing the rest of the work to visit all the nodes could be worth the more limited information for a much quicker result.

Another possible efficiency strategy, which would require more space is to store all sub-paths as shortest paths. By definition of the shortest path, all sub-paths are also shortest paths [35]. If a path is 1-2-3-4, then 2-3-4, 3-4, and 2-3 can also be identified as the shortest paths. This would then give the ability to have many more paths in the memoization table without having to run Dijkstra again. This may be an effective way of dealing with a very large graph with only some paths that have run shortest path searches. There may be ways to identify sub-paths without the need to identify the complete path but use some paths from the dictionary. Since these improvements

were outside of the scope of the project as it was already working more efficiently than the GNN, they were not implemented. Another possible improvement could be to use Fibonacci heaps to improve the run time. This has been stated to be more efficient [35].

11.2 GNN Implementations

Our GNN model does not exhibit superior performance on the Shortest path problem to some extent. Despite the fact that we have implemented several state-of-art GNN models, there are existing other effective models worth trying. Or perhaps we can solve the problem from a different perspective. Instead of focusing on the single output of each prediction, having a sequential output for each graph is more promising and more compatible with our shortest path problem.

In Figure 32 we followed the GGSNN (Gated Graph Sequence Neural Network) proposed by Yujia et al. [18]. This model is capable of producing sequential output for path-finding problems. We modified the general GGSNN model with the following architecture:

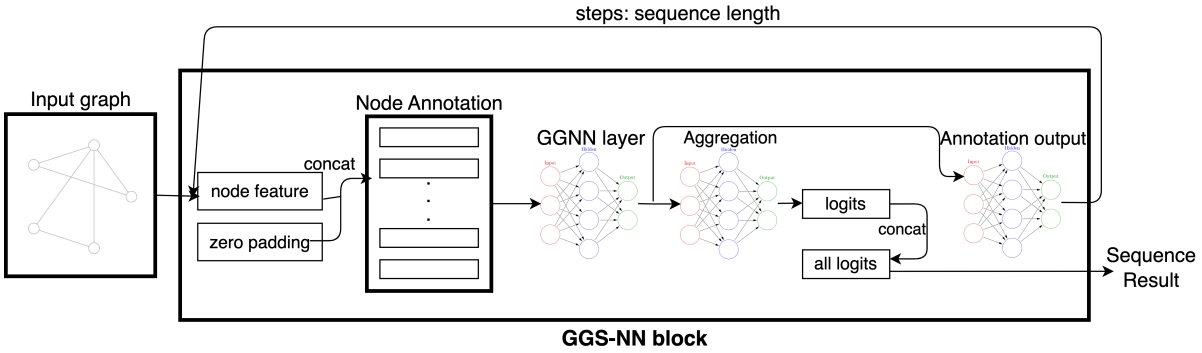


Figure 32: The architecture of our implementation on Gated Graph Sequence Neural Network.

Unfortunately, our SPGGSNN model cannot converge in giving training epochs. Continuously increasing the training epochs can lead to overfitting. Furthermore, the prediction accuracy is not outperformed our edge/node prediction model. Therefore, adapting the model to predict exactly the shortest path is left as future work.

11.3 Possible Improvements for Existing Models

The architectures tested in this project have used an Adam learning rate of 0.0001. An improvement to this model would be implementing Adam Warm-up [36]. According to this paper, a warm-up scheduler helps deep learning models to have more stabilized early stages of training due to regulating the size of parameter updates. This is important since during the early stages of training, the adaptive learning rate has a high variance and can cause the model to converge at worse local minima [37]. Figure 33 showing the graph of learning rate without Adam Warm-up and Figure 34 showing graph for learning rate with Adam warm-up.

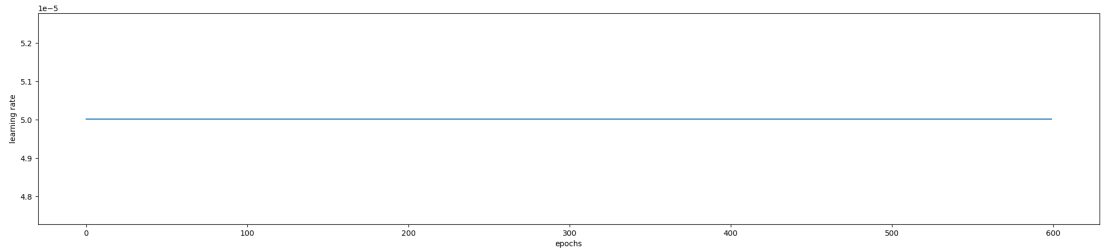


Figure 33: Learning rate without Adam Warm-up (Axes: Learning rate against Epochs)

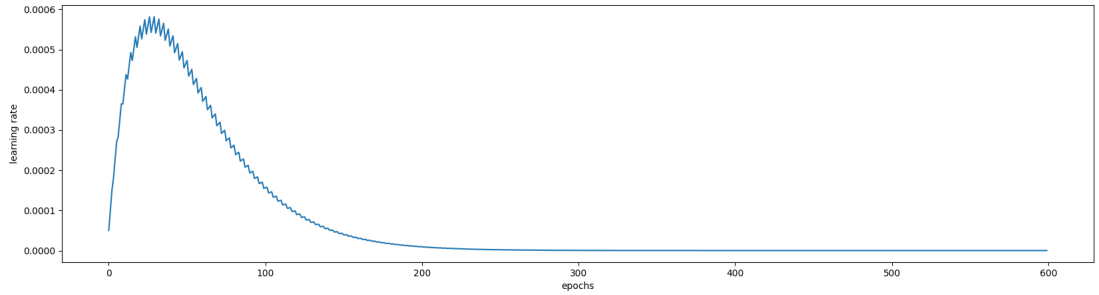


Figure 34: Learning rate with Adam Warm-up (Axes: Learning rate against Epochs)

11.4 Using Landmarks

Taking inspiration for everyday life, we can try implementing points of interest as we use them in everyday travel. For example, when using public transport, a common landmark would be the central station of the city. One pattern that has to be observed is that those landmarks act as central points, or to put it in a different perspective, points, where the possible paths taken, converge.

We consider that there are two possible landmark implementations. In the first case the landmark is a node of importance that is often present in the short path from a number of nodes. Similar to how a transit station like Stockholm Central is present in most of the trips one would take in Sweden. In the second section the landmark is used as a way of providing more features. For example knowing the distance between any node and the landmark nodes. This could serve as an additional feature.

In both cases, we need to generate new data and add an additional class to our prediction. Now some of our nodes need to be classified as landmark nodes.

For the first approach, the nodes considered landmarks need to have a high degree of connectivity and to be part of the shortest path for a large number of pairs of nodes. We can see those landmark nodes as the main transit points in a city. Even if the distance between two points is smaller if we walk it is faster to take the public transport that takes us on a less direct route. A key note here is that our topological graphs are not suited for this as the shortest distance is always the one as the crow flies. To implement this approach we would need to redesign our graph generator.

The second approach involves simply classifying some nodes as landmark nodes and appending more information about them. We could consider that in the case of the landmark nodes we always know the distance to them or something similar. In this situation, we would be able to add another feature that the GNNs could make use of.

For example, we could say that inside a path we have one landmark halfway through the path or one landmark after each quarter of the path has been traversed.

Additionally, the models would need to be changed so that they also predict landmark nodes.

12 Acknowledgements

We would like to thank Konstantinos Vandikas at Ericsson for giving us this interesting project to work on and supporting us throughout the semester and hosting our tour of Ericsson.

Also, we would like to thank our instructors, Shenghui Li and Chencheng Liang for supporting us in our understanding of GNNs this semester and helping us through all of the various challenges that come with a project this large and dynamic.

References

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

[2] "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[3] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[5] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions On Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[6] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015.

[7] R. M. Schmidt, "Recurrent neural networks (rnns): A gentle introduction and overview," *CoRR*, vol. abs/1912.05911, 2019.

[8] J. Gao and L. Hao, "Graph neural network and its applications," *Journal of Physics: Conference Series*, vol. 1994, p. 012004, aug 2021.

[9] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *arXiv preprint arXiv:1804.00823*, 2018.

[10] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. M. Basalamah, "A survey of shortest-path algorithms," *CoRR*, vol. abs/1705.02044, 2017.

[11] M. Zhou and N. Gao, "Research on optimal path based on dijkstra algorithms," in *3rd International Conference on Mechatronics Engineering and Information Technology (ICMEIT 2019)*, pp. 884–892, Atlantis Press, 2019.

[12] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," *CoRR*, vol. abs/1806.01261, 2018.

[13] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019.

[14] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *CoRR*, vol. abs/1704.01212, 2017.

[15] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2017.

[16] A. Sanchez-Gonzalez, J. Godwin, T. Pfaff, R. Ying, J. Leskovec, and P. W. Battaglia, "Learning to simulate complex physics with graph networks," *CoRR*, vol. abs/2002.09405, 2020.

[17] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[18] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[19] D. F. Perez-Ramirez, C. M. Pérez-Penichet, N. Tsiftes, T. Voigt, D. Kostic, and M. Boman, "Deepgantt: A scalable deep learning scheduler for backscatter networks," *CoRR*, vol. abs/2112.12985, 2021.

[20] J. C. Chedjou and K. Kyamakya, "A universal concept for robust solving of shortest path problems in dynamically reconfigurable graphs," *Mathematical Problems in Engineering*, 2015.

[21] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[22] L. Fu, D. Sun, and L. R. Rilett, "Heuristic shortest path algorithms for transportation applications: State of the art," *Computers and Operations Research*, vol. 33, no. 11, pp. 3324–3343, 2006.

[23] D. Rachmawati and L. Gustin, "Analysis of dijkstra's algorithm and a* algorithm in shortest path problem," *Journal of Physics: Conference Series*, vol. 1566, p. 012061, 2020.

[24] Ramadiani, D. Bukhori, Azainil, and N. Dengen, "Floyd-warshall algorithm to determine the shortest path based on android," *IOP Conference Series: Earth and Environmental Science*, vol. 144, no. 1, p. 012019, 2018.

[25] C. W. Ahn and R. S. Ramakrishna, "A genetic algorithm for shortest path routing problem and the sizing of populations," *IEEE transactions on evolutionary computation*, vol. 6, no. 6, pp. 566–579, 2002.

[26] C. W. Ahn, R. Ramakrishna, C. Kang, and I. Choi, "Shortest path routing algorithm using hopfield neural network," *Electronics Letters*, vol. 37, no. 19, pp. 1176–1178, 2001.

[27] A. Nazemi and F. Omidi, "An efficient dynamic model for solving the shortest path problem," *Transportation Research Part C: Emerging Technologies*, vol. 26, pp. 1–19, 2013.

[28] J. Qi, W. Wang, R. Zhang, and Z. Zhao, "A learning based approach to predict shortest-path distances.," in *EDBT*, pp. 367–370, 2020.

[29] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell, "Neural execution of graph algorithms," *arXiv preprint arXiv:1910.10593*, 2019.

[30] M. J. Schuetz, J. K. Brubaker, and H. G. Katzgraber, "Combinatorial optimization with physics-inspired graph neural networks," *Nature Machine Intelligence*, vol. 4, no. 4, pp. 367–377, 2022.

[31] M. C. Angelini and F. Ricci-Tersenghi, "Cracking nuts with a sledgehammer: when modern graph neural networks do worse than classical greedy algorithms," *arXiv preprint arXiv:2206.13211*, 2022.

[32] T. N. Kipf and M. Welling, "Variational graph auto-encoders," 2016.

[33] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. W. Battaglia, "Learning deep generative models of graphs," *CoRR*, vol. abs/1803.03324, 2018.

[34] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[35] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[36] J. Ma and D. Yarats, "On the adequacy of untuned warmup for adaptive optimization," *CoRR*, vol. abs/1910.04209, 2019.

[37] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," *CoRR*, vol. abs/1908.03265, 2019.