STEINER TREE: A DEEP REINFORCEMENT LEARNING APPROACH

by

Siqi Wang

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

2021 Spring

© 2021 Siqi Wang
All Rights Reserved

STEINER TREE: A DEEP REINFORCEMENT LEARNING APPROACH

by

Siqi Wang

Approved: _____
Guangmo Tong, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____
Kathleen McCoy, Ph.D.
Chair of the Department of Computer Science

Approved: _____
Levi T. Thompson, Ph.D.
Dean of the College of Engineering

Approved: _____
Louis F. Rossi, Ph.D.
Vice Provost for Graduate and Professional Education and
Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my gratitude to everyone who helped me with my research.

I want to express my sincere appreciation to my advisor, Dr. Guangmo Tong, for his support and guidance throughout my research.

I also want to thanks Dr.Rui Zhang and Dr.Rahmat Beheshti for their gentle help throughout my research.

Also, I want to give my thanks to my colleagues in the CDS group and all school faculty for their help.

Lastly, I want to thank my parents, my family, and my friend for their encouragement and support.

TABLE OF CONTENTS

LIST OF FIGURES . . . . . **vi**
ABSTRACT . . . . . **vii**

Chapter

1 INTRODUCTION **1**

2 BACKGROUND KNOWLEDGE **4**

2.1 Steiner Tree Problem 4
2.2 Reinforcement Learning 4

2.2.1 Reinforcement Learning and Combinatorial Optimization Problems 6

2.3 Attention and the Transformer 8

3 THE PROPOSED WORK **10**

3.1 Generate Candidate Set 11

3.1.1 Pruning Method 12

3.2 Learn a Representation 13
3.3 Find a Steiner Point 16
3.4 Merge Steiner Points to Original Node Set 18
3.5 The REINFORCE Method 18

4 EXPERIMENT **20**

4.1 Experiment Setting and Result 20
4.2 Experiment Learning 23

5 CONCLUSION **26**

REFERENCES **27**

LIST OF FIGURES

1.1 An example of using Steiner tree to build light rail transit 2

1.2 Comparison between Minimal Spanning Tree and Steiner tree . . . 3

2.1 Properties of optimal Steiner points 5

2.2 Interaction between agent and environment 5

2.3 Agent generates solution for TSP 7

3.1 Structure of an iteration to select Steiner point 11

3.2 Steiner Arc 11

3.3 3 nearest nodes 12

3.4 Arc approximation 13

3.5 Encoder module 14

3.6 Decoder module 17

3.7 Before merging Steiner points to original node set 18

3.8 After merging Steiner points to original node set 18

4.1 Worse than MST 21

4.2 No obvious difference 21

4.3 Better than MST 22

4.4 Agent chooses an awful points 23

4.5 Quality of candidate set generated by arc approximate approach . . 24

ABSTRACT

The Steiner tree problem is a classical combinatorial optimization problem that targets interconnecting a set of points by a network whose total length is the shortest, where the network consists of the original points and the newly added points called Steiner points. Since the problem can be reduced to the minimum spanning tree problem if the Steiner points in the optimal solution could be found, finding the optimal Steiner points is the key part of solving Steiner tree problems. Recent research shows that reinforcement learning can learn effective heuristic algorithms, suggesting a new way for computing the Steiner points. Motivated by this, we attempt to use deep learning and reinforcement learning to solve the Steiner tree problem. In our approach, we first generate a candidate set for Steiner points based on the Steiner points' property, and then compute the node representation through the attention mechanism. Based on the learned representation, we learn a policy for selecting Steiner points using the REINFORCE algorithm.

Chapter 1

INTRODUCTION

Given n original points in the plane, the goal of the Steiner tree problem is to interconnect these points by a network whose total length of edges is the shortest. The Steiner tree problem was proposed by V.Jarnik et al. in 1934 [1], and it has many applications in the real world, such as urban planning [2], distribution network planning [3], and wireless networks planning [4]. An example is illustrated in Figure 1.1.

The Steiner tree problem is different from the Minimal Spanning Tree problem. We can use not only the original points but also newly added points, which are called Steiner points, to construct a Steiner tree. Meanwhile, if we have found the optimal Steiner points, we can construct a Minimal Spanning Tree contains all the original points and optimal Steiner points. An example is illustrated in Figure 1.2.

The Steiner tree problem is an NP-hard problem [5]. Traditional methods to solve the Steiner tree problem can be divided into exact algorithms based on enumeration methods, and heuristic algorithms using human-designed heuristics. Warme et al. [6] propose an exact algorithm to solve the Steiner problem based on backtrack searching, dynamic programming, and integer programming. Their algorithm can guarantee to find the optimal solution but requires high running time when the scale of the problem is large. Bereta et al. [7] propose a method based on the memetic algorithm, which shows a good heuristic is helpful to solve the Steiner tree problem. Recently, researchers have been seeking to combine deep learning with reinforcement learning to learn a more effective heuristic algorithm for solving combinatorial optimization problems, such as the Travelling Salesman Problem (TSP) [8] [9] [10] [11] and the Vehicle Rescheduling Problem (VRSP) [11] [12] [13]. Kool et al. [9] use a deep reinforcement

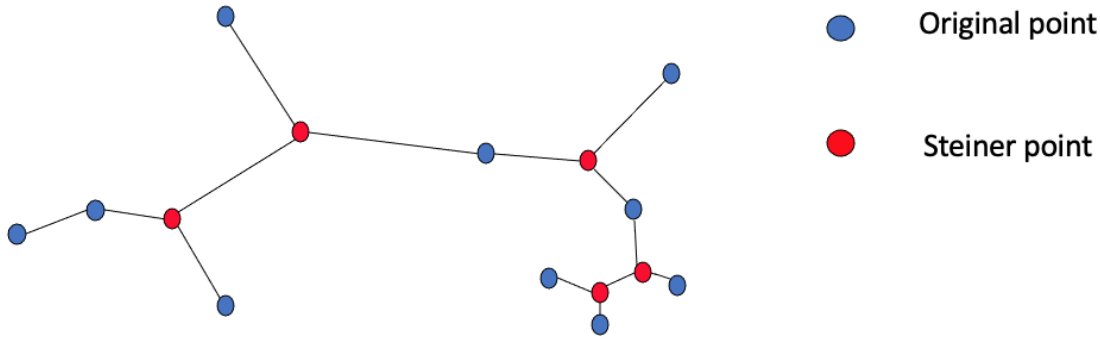


Figure 1.1: An example of using Steiner tree to build light rail transit: There are ten land buildings (blue points). We interconnect them by a Steiner tree to ensure the total length of light rail transit is the shortest. The newly added vertexes (red points) in the network are the Steiner points.

learning method to solve the TSP problem and get a better solution than the state-of-the-art heuristic solver. Lu et al. [13] also beat the state-of-the-art algorithm in solving the VRSP problem by designing a deep reinforcement learning method. Their research follows the idea to learn heuristics for combinatorial optimization problems in a reinforcement learning model. They also use the representation learning technique by introducing the attention mechanism to help with learning the heuristic. The outstanding results of their researches show that a reinforcement learning framework based on deep learning may be a good approach to solve combinatorial optimization.

Motivated by the previous researches, we aim to apply the deep reinforcement learning approach to solve the Steiner tree problem. This thesis presents a deep reinforcement learning method for the Steiner tree problem. In our method, we use a deep reinforcement learning model to generate a feasible solution for the Steiner tree problem and optimize the model based on the quality of the feasible solution using the REINFORCE algorithm [14]. In our deep reinforcement model, we first generate

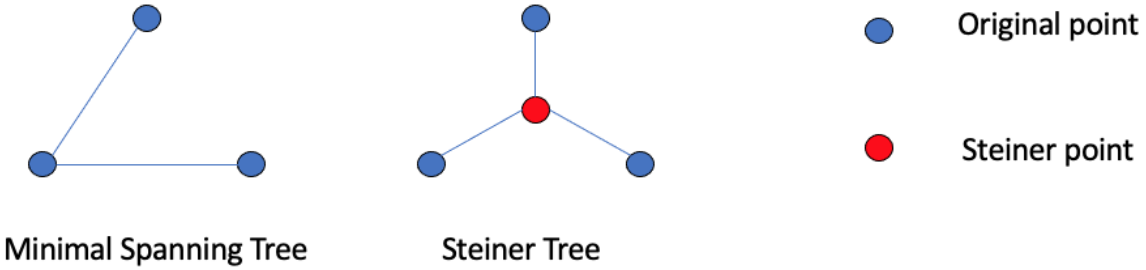


Figure 1.2: Comparison between Minimal Spanning Tree and Steiner tree

a candidate set for Steiner points based on the Steiner points' property. Then we will use an encoder module based on multi-head attention to compute node representation for every point in the candidate set and original node set. The learned node representation is the input to a decoder module, whose role is to use the attention mechanism to generate a policy to select Steiner points. Based on the generated policy, we can generate a feasible solution.

In Chapter 2, we introduce the background knowledge related to our research. In Chapter 3, we present the proposed method. In Chapter 4, we present experimental studies.

Chapter 2

BACKGROUND KNOWLEDGE

2.1 Steiner Tree Problem

Definition 1. (Steiner Tree Problem) [15]: Given an original node set S in the plane, find a geometric network $T = (V, E)$ such that $S \subset V$ and $N = V \setminus S$ is a (possibly empty) set of points known as Steiner points, and such that $\sum_{e \in E} |e|$ is minimized (where $|e|$ denotes the Euclidean length of edge $e \in E$).

In our project, we will use two important properties of Steiner point [16]:

Property 1. There are at most $(n-2)$ Steiner points for a minimal Steiner tree which has n original points.

Property 2. Optimal Steiner points must have a degree of three, and the three edges incident to such a point must form three 120 degree angles.

An example is illustrated in Figure 2.1.

2.2 Reinforcement Learning

Reinforcement learning is a machine learning method, which focuses on training an agent to take actions to maximize a goal. We aim to train the agent to select high-quality Steiner points to solve the Steiner tree problem through reinforcement learning. In reinforcement learning, an agent learns optimal behavior in interacting with a dynamic environment, which gives the agent feedback based on the agent's behavior [17]. The agent will modify its performance through trial and error. We can take how a baby learns to walk as an example. When the baby learns to walk, he wants to keep balance and avoid falling. He feels happy if he keeps the balance, feels pain if he falls. Finally, he learns how to walk from these experiences.

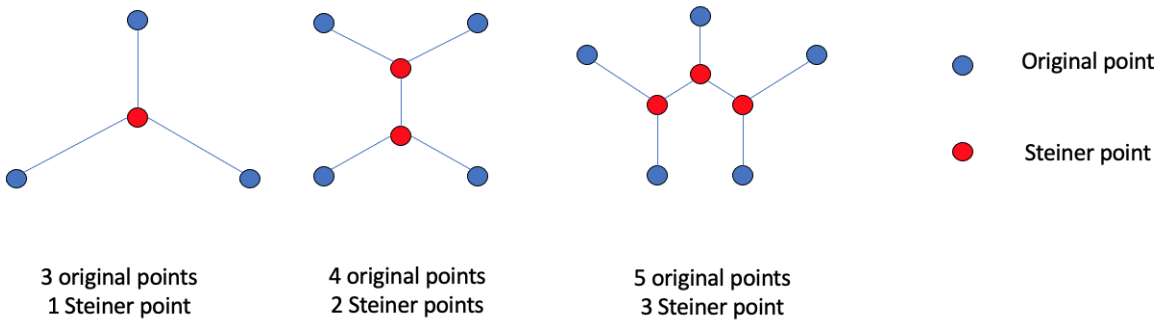


Figure 2.1: Properties of optimal Steiner points: All Steiner points (red points) has degree of three and forms three 120 degree angles

For a reinforcement learning task, we can design a reinforcement learning model to simulate the processes of learning. As shown in Figure 2.2, on each step of interaction, t, the agent senses the environment by a sensor and receives a representation of the environment's state, s. Based on the state, the agent will choose an action, a, following a policy, π, which is a probability distribution of selecting each possible action according to the state. As the consequence of the agent's action, the agent will receive a numeric reward, r, from the environment as an evaluation of his action and move to a new state due to the effect of its action on the environment. During the interactions with the environment, the agent updates the policy by trial and error to maximize the sum of reward it gets over the long run.

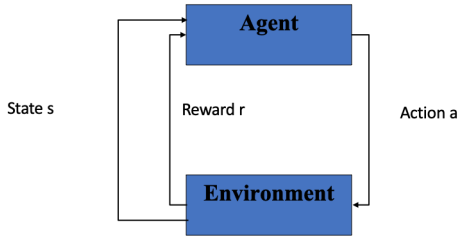


Figure 2.2: Interaction between agent and environment [18]

Formally, reinforcement learning can be described as a Markov Decision Processes [19], which consists of a set of states, $s_t \in S$, a set of actions, $a_t \in A$, a policy $\pi_t(s_{t+1}|s_t, a_t)$, and reward function $R(s_t, a_t)$. In the Markov Decision Processes, the

agent tries to find the optimal policy π^* that maximizes the long-term reward. If the interactions will stop after T more steps, we can define π^* as:

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}[\sum_{k=0}^{|T|} R(s_{t+k}, a_{t+k})] \tag{2.1}$$

There are two reinforcement learning methods to search for the optimal policy π^*. The first method is the value-based method [18]. In the value-based method, we will estimate the Q function.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[\sum_{k=0}^{|T|} R(s_{t+k}, a_{t+k}) | s_t = s, a_t = a, \pi] \tag{2.2}$$

The Q function is the expected reward the agent will get if he follows a certain policy after taking the given action a in the given state s. There is always a Q function larger than other Q function, which is called the optimal Q function:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \tag{2.3}$$

The optimal Q function is the expected reward the agent will get if he follows the optimal policy after taking the given action a in the given state t. The agent will always choose the action that maximizes the Q function.

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^{\pi}(s, a) \tag{2.4}$$

The performance of the value-based method is based on the estimation of the Q function. So it's very important to design an approach to estimate an accurate Q function in an efficient way. The Second method is the policy-based method. In the policy-based method, we let the policy $\pi_\theta(s)$ be related to parameters θ. We can optimize the parameters θ by maximizing the final reward.

2.2.1 Reinforcement Learning and Combinatorial Optimization Problems

There are some recent researches shows that it is possible to use reinforcement learning to solve combinatorial optimization problems [20] [9]. Their motivation is that

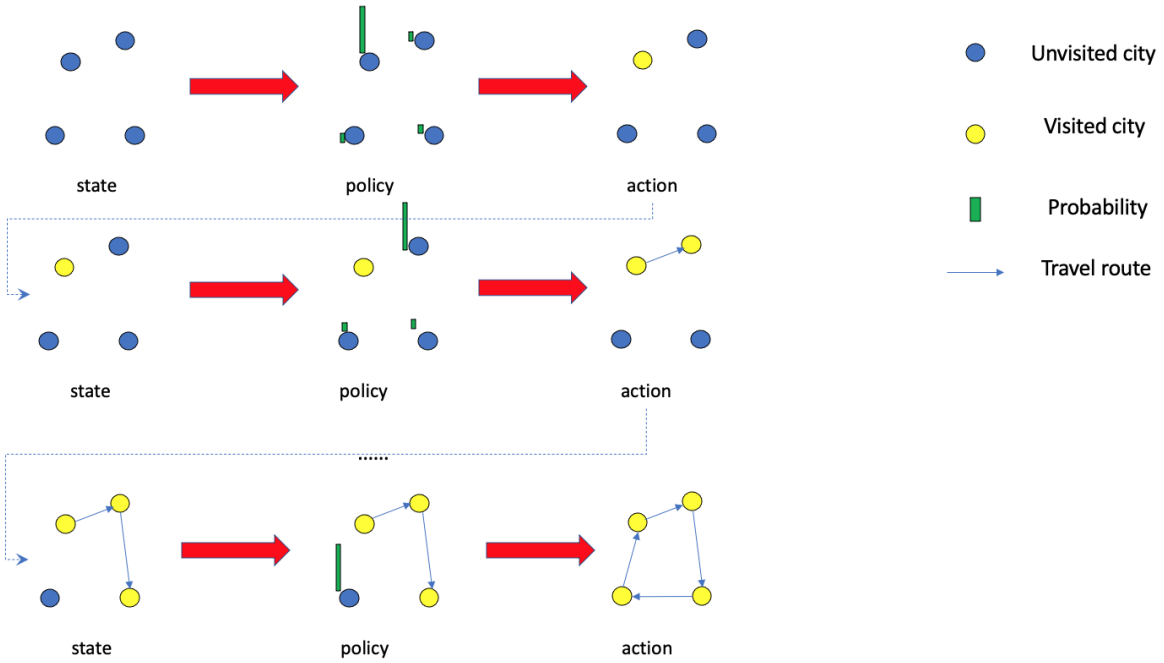


Figure 2.3: Agent generates solution for TSP: The state is the cities the agent has visited. Agent's action is to choose a city he has not selected and move to the city. The agents takes action following the policy meanwhile he expends the solution. Finally, the agent will get a solution for the TSP.

combinatorial optimization problems can be modeled as Markov Decision Processes [21], and reinforcement learning is a good framework for solving problems expressed as Markov Decision Processes [18].

When we use reinforcement learning to solve combinatorial optimization problems, we follow the pipeline proposed by Mazyavkina et al. [22]. We use Markov Decision Processes to model the combinatorial optimization problem. For different combinatorial optimization problems, we define diffident states, actions, and rewards. Then we train a reinforcement learning model to generate a feasible solution by taking states as input and outputting action selected following a parameterized policy. As shown in Figure 2.3, the agent generates the solution following the policy. Based on the final reward of the feasible solution, the model optimizes parameters θ in policy for the purpose of generating a better solution.

2.3 Attention and the Transformer

Representation learning [23] is a technique that allows a deep learning model to learn a better representation for input features. Attention [24] is a mechanism that allows a model to selectively focus on different parts of the input. By assigning different weights to different parts of the input, attention allows the model to hold all the information of the input and meanwhile pay more attention to the more valuable ones, which makes attention is a good approach to do representation learning. In our approach, we use multi-head self attention, a kind of attention mechanism, to learn a good representation for the points in the plane.

The multi-head self attention is proposed by Vaswan et al. [25] in the Transformer model and is designed based on self attention to solve machine translation tasks, whose objective is to translate a given source sentence in a language to a target sentence in another language. The Transformer model is an attention model proposed by Vaswan et al. [25] and is in the Encoder-Decoder structure. While their encoder is a stack of six identical layers rather than only one layer, which is diffident from other attention models in the past. The Transformer model follows the idea proposed by Miller et al. [26], which is that every word in the source sentence is a pair <Key, Value> and every word in the target sentence is a Query. Their method use Key, Value, Query to calculate the attention. Firstly, they calculate the similarity between a Query to each Key in the source sentence:

$$\text{similarity}_i = \text{Query} \cdot \text{Key}_i \tag{2.5}$$

Secondly, they use softmax function to calculate the weights based on similarity, which is called attention score:

$$a_i = \text{softmax}(\text{similarity}_i) \tag{2.6}$$

In the third step, they can get the attention as the weighted sum of Values:

$$\text{attention}(\text{Query},\text{Value}) = \sum_{i=1}^{L_x} a_i \cdot \text{Value}_i \tag{2.7}$$

L_x is the number of words in the sentence.

The Transformer calculates the attention score in a similar way, but they divided the similarity by $\sqrt{d_k}$, d_k is the dimension of the Key:

$$\text{attention(Query,Key,Value)} = \text{softmax}(\frac{\text{Query} \cdot \text{Key}^T}{\sqrt{d_k}})\text{Value} \tag{2.8}$$

In this way, softmax function will output other numbers rather than 0 or 1 even if the inner product calculated from key and value being too large.

The self attention in the Transformer is different from the traditional attention in that they focus on the relationship between words in the source sentence rather than the relationship between source sentence and target sentence, which means that Query, Key, and Value are all from the same place: the source sentence or the former encoder layer's output. As shown in equation 2.9, we can calculate the self attention and get attention matrix Z in a similar way:

$$Z = \text{attention(Query,Key,Value)} \tag{2.9}$$

Vaswan et al. find the observation that if they only calculate one self attention, it's hard to learn a good representation of the input. So they propose the Multi-head self attention for the purpose of getting the attention from different aspects to learn better representation. At first, they map the Query Value, Key into different subspace N times to get different Q_i, K_i, V_i and calculate attention head_i based on each pair (Q_i, K_i, V_i). Secondly, they concatenate the N attentions to get the multi-head attention:

$$\text{multihead(Query,Key,Value)} = \text{concate}(\text{head}_1, ..., \text{head}_N) \tag{2.10}$$

Thirdly, they let the concatenation times a weighted matrix to get the attention matrix Z:

$$Z = W_W \cdot \text{multihead(Query, Key,Value)} \tag{2.11}$$

Chapter 3

THE PROPOSED WORK

In chapter 3, we propose a deep reinforcement learning approach to solve the Steiner Tree problem.

In our approach, we use a parameterized reinforcement learning model to generate a feasible solution for the Steiner tree problem in the 2-D plane. The model generates a feasible solution by using an iterative approach to select Steiner points. Each iteration has four parts (Figure 3.1).

- Given an input node set V_i, we firstly generate a candidate set PSP_i. We merge the points in the candidate set PSP_i to the input node set V_i to get a new node set $V_i^{'} = V_i \cup PSP_i$, from which we generate a complete Graph $G_i(V_i^{'})$.
- In the second step, we take the G_i as the input to the encoder module, whose role is to learn a good representation to represent each node in the node set $V_i^{'}$. The encoder will output the learned representation as the node embeddings for every node in the node set $V_i^{'}$ and a graph embedding for the whole Graph G_i.
- In the third step, we will calculate the probability of each node in PSP_i selected as the Steiner point in the decoder module based on the node embeddings and graph embedding. We will select a node $st_i \in PSP_i$ as the Steiner node based on the probability.
- At last, we merge the node st_i into the original node set V_i, let $V_{i+1} = V_i \cup \{st_i\}$, which is the input for next iteration.

Based on the property 1 of the Steiner points, we need to select at most $n-2$ Steiner points. So we will repeat the iteration $n-2$ times. Then we use the REINFORCE algorithm to optimize the parameters θ in the reinforcement learning model based on the feasible solution V_{n-1}.

In the following four sections, we will explain each part separately. Then in selection 3.5, we will introduce how to use the REINFORCE method to train the model.

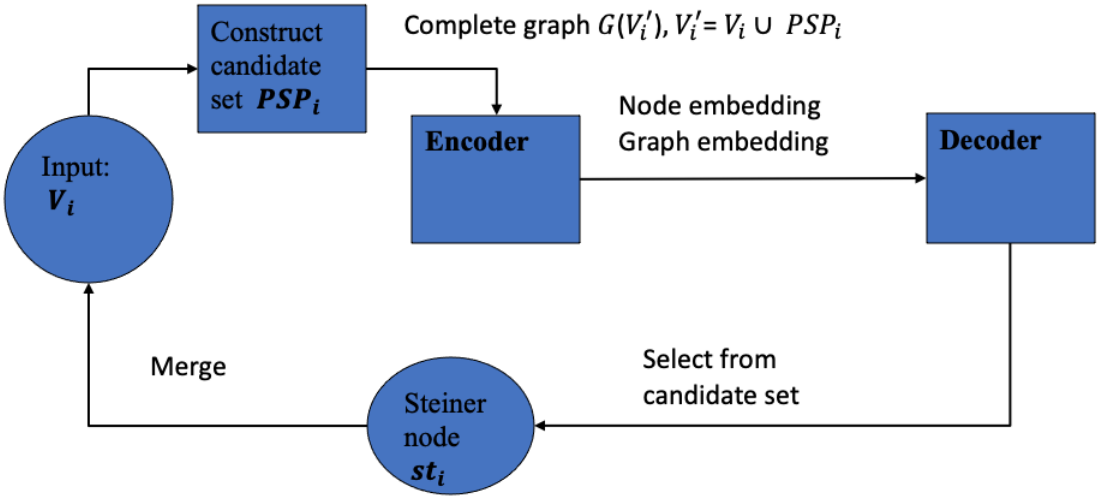


Figure 3.1: Structure of an iteration to select Steiner point

3.1 Generate Candidate Set

In this section, we introduce a method to generate a candidate set for the Steiner points, which will reduce the search space for the Steiner points from the 2-D plane to the candidate set. The idea of generating candidate sets is from the property 2 for the Steiner points. Based on property 2, we define the Steiner Arc. An example is illustrated in Figure

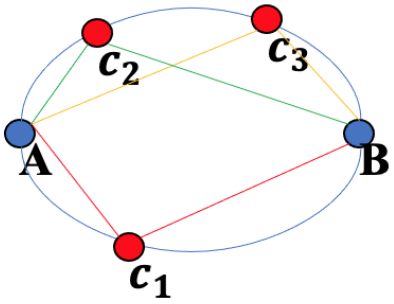


Figure 3.2: Steiner Arc: All points in $\overset{\frown}{AB}$ is the vertex of a 120 degree angle, such as c_1, c_2, c_3, $\angle Ac_1B = 120$, $\angle Ac_2B = 120$, $\angle Ac_3B = 120$.

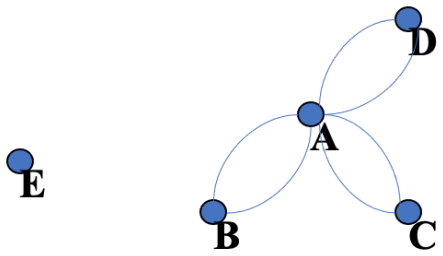


Figure 3.3: 3 nearest nodes: The point A will generate Steiner Arc with 3 nearest nodes, such as B, C, D. It will not generate Steiner Arc with node E.

3.2.

Definition 2. Steiner Arc: Given two points A and B, we consider the set $\overset{\frown}{AB} = \{C : C \in R^2, \angle ACB = 120\}$ is the Steiner Arc.

Given an input node set V_i, which is the original node set S in the first iteration, we assume $|V_i| = m_i$. For every two nodes v_a, v_b in the node set V_i, we can generate a Steiner Arc. So for all node pairs (v_a, v_b) in node set V_i, we can generate $\frac{m_i(m_i-1)}{2}$ Steiner Arcs. We will construct a candidate set PSP_i including all the points in the $\frac{m_i(m_i-1)}{2}$ Steiner Arcs, making every node in PSP_i is a possible Steiner point. Then we can find a Steiner points st_i from the candidate set PSP_i.

After getting the candidate set PSP_i, we merge the candidate set PSP_i to the node set V_i and get a new node set $V_i^{'} = PSP_i \cup V_i$, from which we generate a complete Graph $G_i(V_i^{'})$. The $G_i(V_i^{'})$ will be the input for encoder module.

3.1.1 Pruning Method

When we generate the candidate set, we use two pruning methods. The first one is K nearest node, and the second one is arc approximate.

K nearest nodes: In our proposed method, we will generate a Steiner Arc for every node pair in the nodes set V_i. After observing the minimal Steiner tree, we find that the Steiner point locates in the Steiner Arc generated by two nodes close to each

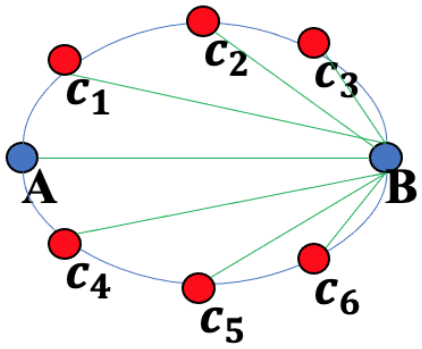


Figure 3.4: Arc approximation: We will get six endpoints $c_1, c_2, c_3, , c_4, c_5, c_6$ in $\overset{\frown}{AB}$. And let $\angle ABc_1 = \angle ABc_4 = 15$, $\angle ABc_2 = \angle ABc_5 = 30$, $\angle ABc_3 = \angle ABc_6 = 45$. Then we will add the six endpoints $c_1, c_2, c_3, c_4, c_5, c_6$ into candidate set rather than the whole arc.

other. It means we do not need to generate a Steiner Arc for the two points far away from each other. So for a node v_a in the nodes set V_i, we only choose v_a's K nearest node to generate Steiner Arc. An example is illustrated in Figure 3.3

Arc approximation: Given a Steiner Arc $\overset{\frown}{AB}$, even the length of $\overset{\frown}{AB}$ is short, there are also infinity points in $\overset{\frown}{AB}$. So we need to use some points in the $\overset{\frown}{AB}$ to represent it. In our method, we will cut every Steiner Arc into eight parts. So we will get six new endpoint to represent the Steiner Arc. An example is illustrated in Figure 3.4

3.2 Learn a Representation

The role of the encoder module is to do representation learning. Given the complete Graph $G_i(V_i^{'})$, we can directly train the decoder module to optimize a parameterized policy instructing the agent to find the Steiner points from PSP_i. If we do not introduce representation learning, the representation of each node is 2-dimensional, a (x, y) pair representing a point's location, which will make it difficult for the decoder module to generate a high-quality policy. So we attempt to learn a good representation of the input feature in the encoder module to help the decoder module to generate

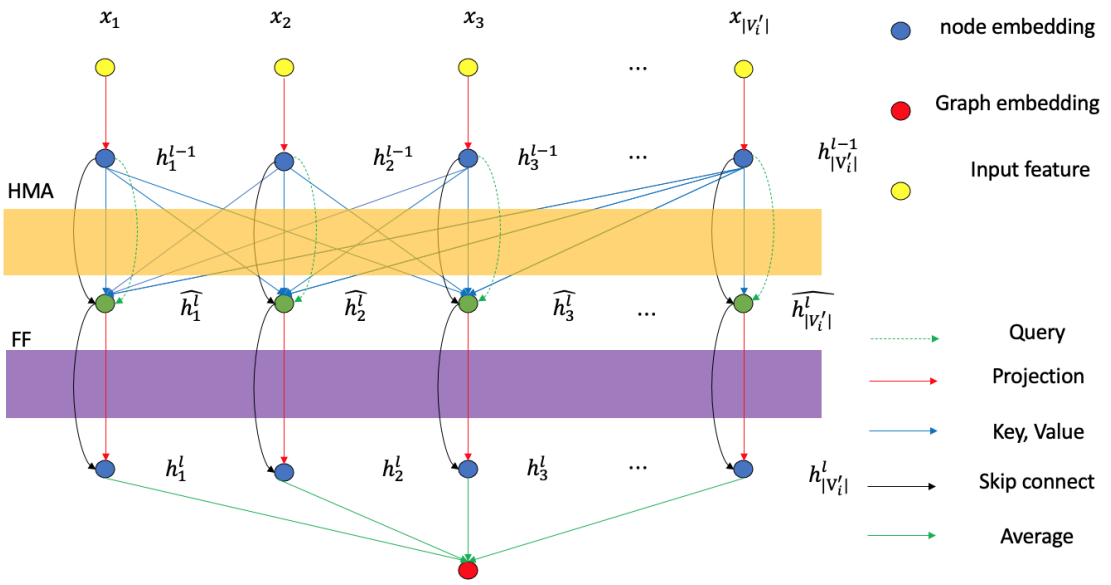


Figure 3.5: Encoder module [9]

better policy by introducing the encoder module.

Motivated by Kool et al. [9] using attention model to solve TSP problem successfully, we modify the attention model proposed by Kool et al. [9]. The encoder model (Figure 3.5) we use is the same as Kool et al.'s model, which is similar to the encoder used in the Transformer [25]. The encoder module has three attention layers taking node embedding as input and each attention layer consists of one multi-head attention layer and one feedforward layer. So before we input the node set $V_i^{'}$ into the encoder module, we will transform it to 128-dimensional node embedding:

$$h_j^{(0)} = W^X l_j + b^x \tag{3.1}$$

The $h_j^{(0)}$ is the initial node embedding for the jth node in the node set $V_i^{'}$, l_j is the location (x_j, y_j) for the jth node in the node set $V_i^{'}$, and W^x, b^x are the parameters that need to be learned.

The multi-head attention is based on the self attention, which is calculated based on Query, key, Value from the node embedding as we mentioned in sub-chapter 2.3. We will use equation 3.2 to get Query, Key, Value from node embedding:

$$\begin{aligned} q_j^{(l)} &= W^Q h_j^{(l-1)} \\ k_j^{(l)} &= W^K h_j^{(l-1)} \\ v_j^{(l)} &= W^V h_j^{(l-1)} \end{aligned} \tag{3.2}$$

W^Q, W^K, W^V are learned parameters. $h_j^{(l)}$ is the node embedding for jth node getting from lth attention layer. $q_j^{(l)}$, $k_j^{(l)}$, $v_j^{(l)}$ is the Query, Key, Value for jth node calculated in lth attention layer.

The calculation of the self attention for a node v_j has three steps. In the first step, the module will uses the dot-product to calculate the compatibility u_{jo} between the Query of the node v_j and the Key of other nodes v_o in the node set (we omit l for readability):

$$u_{jo} = \frac{q_j^T k_o}{\sqrt{d_k}} \tag{3.3}$$

u_{jo} will be used to calculate weights by the softmax function in the second step:

$$a_{jo} = \frac{e^{u_{jo}}}{\sum_{o'} e^{u_{jo'}}} \tag{3.4}$$

In the third step, the module will assign weights to the corresponding Value:

$$h_j^{'} = \sum_o a_{jo} v_o \tag{3.5}$$

Then we can generate M different self attention $h_{j1}^{'}, ..., h_{jM}^{'}$ by mapping the h_j^{l-1} into M different subspace, $h_{jp}^{'}$ is the pth self attention for node v_j. Based on these M self attentions, we can get the multi-head attention for node v_j. The multi-head attention for the node v_j can be calculated in the following equation.

$$\mathrm{MHA}_j^{(l)}(h_1^{(l-1)}, h_2^{(l-1)}, ..., h_n^{(l-1)}) = \sum_{m=1}^{M} W_m^O h_{jm}^{'} \tag{3.6}$$

W_m^O is the learned weighted matrix.

The role of the multi-head attention is to learn representation from different representation subspace, which is helpful to learn a good representation. The multi-head attention layer adds batch normalization [27] and skip-connection [28] to make sure every node can have its unique representation.:

$$\hat{h}_j^{(l)} = \mathrm{BN}^{(l)}(h_j^{(l-1)} + \mathrm{MHA}_j^{(l)}(h_1^{(l-1)}, h_2^{(l-1)}, ..., h_n^{(l-1)})) \tag{3.7}$$

After getting multi-head attention $\hat{h}_j$ from the multi-head attention layer, the multi-head attention will be passed into a feedforward layer, whose role is to improve the capacity of the module and output the learned node embedding $h_j^{(l)}$. The feedforward layer also adds batch normalization and skip-connection:

$$h_j^{(l)} = \mathrm{BN}^{(l)}(\hat{h}_j^{(l)} + \mathrm{FF}^{(l)}(\hat{h}_j^{(l)})) \tag{3.8}$$

After the three attention layers, we will get the node embedding $h_j^{(3)}$ and graph embedding $\bar{h}^{(3)}$ as the input to the decoder module. The graph embedding is the mean of the node embeddings $h_j^{(3)}$:

$$\bar{h}^{(3)} = \frac{1}{n}\sum_{j=0}^{|V_i^{'}|} h_j^{(3)} \tag{3.9}$$

3.3 Find a Steiner Point

The role of the decoder module is generating a policy to select a Steiner point. Different from the decoder designed by Kool et al.[9], we only output one Steiner point rather than the whole sequence in the decoder module. So we make some changes to the decoder to use one single attention layer, which takes the node embedding and graph embedding getting from the encoder module as the input and output a policy based on the Query from graph embedding and Key from node embedding as shown in equation 3.10:

$$\begin{aligned} q_g &= W^Q \bar{h}^{(3)} \\ k_j &= W^K h_j^{(3)} \end{aligned} \tag{3.10}$$

As shown in Figure 3.6, for the reason that we only want to judge whether the node in the PSP_i is the Steiner point, we only calculate the compatibility between the q_g

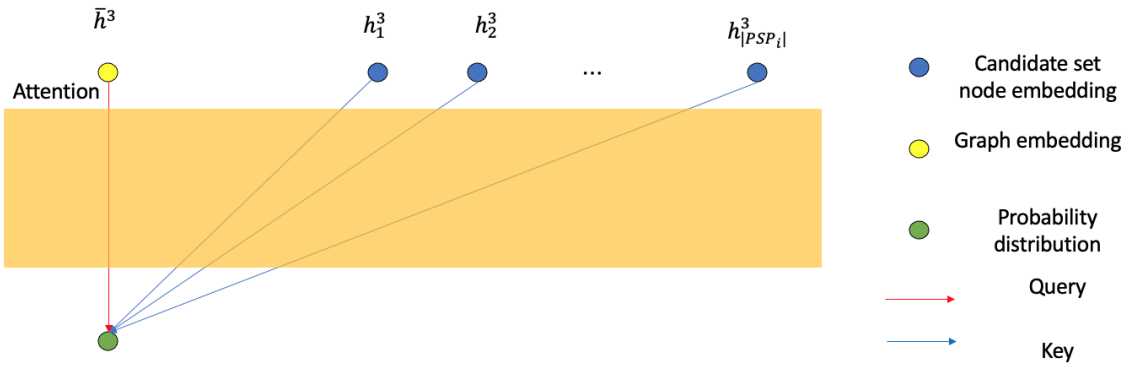


Figure 3.6: Decoder module [9]

and the Key from the nodes in the PSP_i, which leads us to mask all the original node. The equation 3.11 show how to calculate compatibility $u_{(g)j}$.

$$u_{(g)j} = \begin{cases} \frac{q_g^T k_j}{\sqrt{d_k}}, v_j \in PSP_i \\ -\infty, \text{otherwise} \end{cases} \tag{3.11}$$

Following the Bello et al.'s [8] observation that clipping the compatibility helps with exploration, we also clip the compatibility using the $\tanh(\cdot)$ activation function:

$$u_{(g)j} = \begin{cases} C \cdot \tanh(\frac{q_g^T k_j}{\sqrt{d_k}}), v_j \in PSP_i \\ -\infty, \text{otherwise} \end{cases} \tag{3.12}$$

As shown in equation 3.13, we can normalize the compatibility to get a probability distribution, which is the policy for selecting the Steiner point.

$$p_j = p_\theta(\pi_i = j|s) = \frac{e^{u_{(g)j}}}{\sum_o e^{u_{(g)o}}} \tag{3.13}$$

During the training, we use the sampling method to decide the node selected as the Steiner points, which helps with exploration. And we always choose the node with the largest probability during the validation and test.

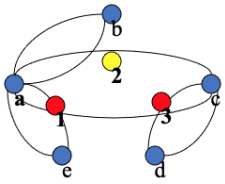


Figure 3.7: Before merge Steiner points to original node set, PSP_1 does not include optimal Steiner point 2.

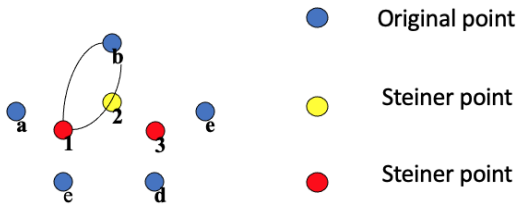


Figure 3.8: After merging Steiner points to original node set, PSP_2 includes optimal Steiner point 2 (Suppose we have found optimal Steiner point 1 in the first iteration).

3.4 Merge Steiner Points to Original Node Set

We merge selected Steiner points to original node set for the purpose of covering more possible Steiner points. As shown in Figure 3.7, for the reason that we only have an original point set $V_1 = S$ in the beginning, the candidate set PSP_1 can only be generated from all the node pairs v_i, v_j in point set S, which means all the possible Steiner points in PSP_1 are connected to 2 original points. However, some of the Steiner points are connected to only one original point. So after we find the Steiner point st_i in the ith iteration, we will merge st_i to the node set V_i, getting $V_{i+1} = V_i \cup \{st_i\}$ as the input for the next iteration. Following this approach, as shown in Figure 3.8, we generate the candidate set PSP_{i+1} in $i+1$th iteration, which includes more Steiner points connected to only one original point.

3.5 The REINFORCE Method

After finishing $n-2$ iterations, we will get Steiner points sequence $ST = \{st_1, ..., st_{n-2}\}$, which is considered as the Steiner point for the training instance, s, determined by the policy $p_\theta(ST|s)$.

We will make a selection in the Steiner points sequence for the reason that there are $n-2$ Steiner points, but most of the time, we don't need $n-2$ Steiner points. From

the Steiner node sequence $ST = \{st_1, ..., st_{n-2}\}$, we can get a possible solution set SS $= \{PS_1, PS_2, ..., PS_{n-2}\}$ following $PS_1 = \{st_1\} \cup S$, $PS_i = \{st_i\} \cup PS_{i-1}$. Then we define the loss function as:

$$\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(ST|s)}[F(ST)] \tag{3.14}$$

$F(ST)$ is the length of the smallest Steiner tree we have:

$$F(ST) = min(MST(PS_i)), PS_i \in SS \tag{3.15}$$

Following the REINFORCE gradient estimator [14], we optimize $\mathcal{L}$ by gradient descent with baseline b(s):

$$\bigtriangledown\mathcal{L}(\theta|s) = \mathbb{E}_{p_\theta(ST|s)}[(F(ST) - b(s)) \bigtriangledown \log p_\theta(ST|s)] \tag{3.16}$$

The $b(s)$ is the cost calculated by baseline, which is the best model we have. In the first training epoch, we have no comparison and we think the first model is the baseline. After each training epoch T_i, we will get a different model to be compared with the baseline and a different Steiner node sequence ST. When $\mathcal{L}(\theta|s) - b(s) < 0$, it means the model we get in the T_ith training epoch may be better than the baseline we have. Then we will take a paired t-test($\alpha = 0.05$) on 10000 different instances to check whether the two baselines are different. If the model passes the t-test, we will replace the parameters θ from the $b(s)$ with the θ from the model gotten in the T_ith epoch. In general, the baseline keeps the best model.

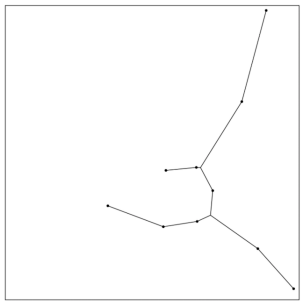
Chapter 4

EXPERIMENT

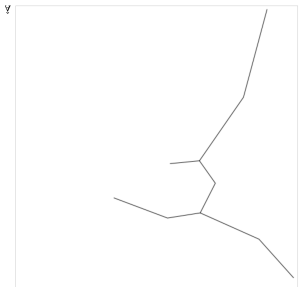
4.1 Experiment Setting and Result

In our approach, we define every training instance as a node set S including n points in a 2-D plane and we use the pair (x_i, y_i) to represent each point's location. We train our model on problem instances of $n = 10$ nodes. All parameters are initialized Uniform$(-1/\sqrt{2}, 1/\sqrt{2})$. We process 200 batches of 50 instances in every epoch and train for 100 epochs using training instances generated by a random algorithm. The algorithm will generate 10 random points from $[0, 100]^2$ for each instance. The test instances, whose size is 1000, are generated by the same algorithm. And we set the K = 3 for the K nearest node pruning method.

We compare the minimal Steiner tree generated by GeoSteiner [6], minimal spanning tree, and the Steiner tree we generate to make some analysis and aim to find some approaches to update our model. Some visualized result are shown in Figure 4.1, Figure 4.2, Figure 4.3. The average length of Steiner tree we generated over the 1000 test instances is 211.197. And the average length of minimal spanning tree over the 1000 test instances is 206.930. Our method performs better than minimal spanning tree in 135 test instances.

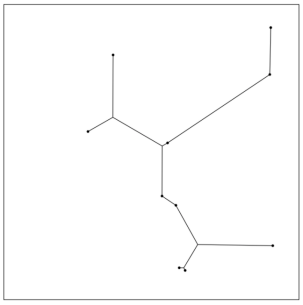


(a) minimal Steiner tree (length 164.230)

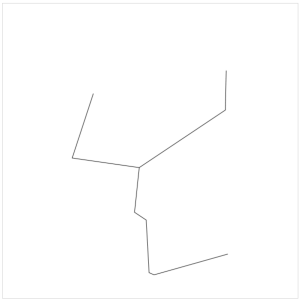


(b) minimal spanning tree (length 165.607)

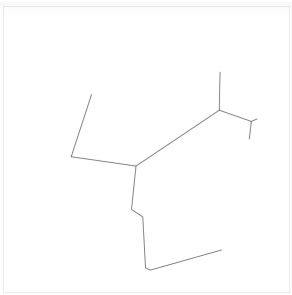
(c) Steiner tree we generate (length 190.296)



(d) minimal Steiner tree (length 186.892)

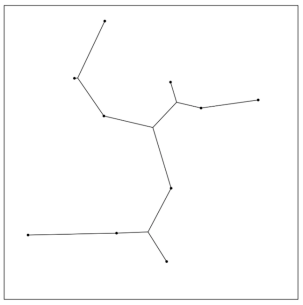


(e) minimal spanning tree (length 192.23)

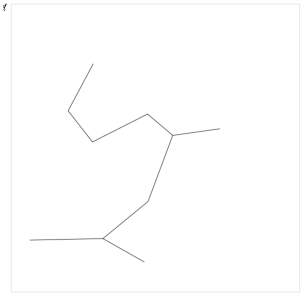


(f) Steiner tree we generate (length 216.763)

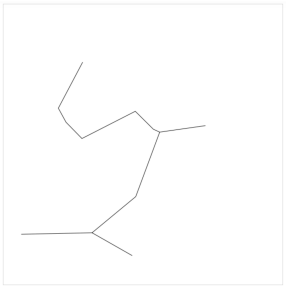
Figure 4.1: Worse than MST



(a) minimal Steiner tree (length 195.810)

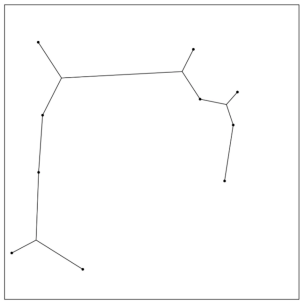


(b) minimal spanning tree (length 202.127)

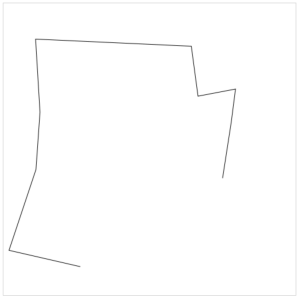


(c) Steiner tree we generate (length 202.426)

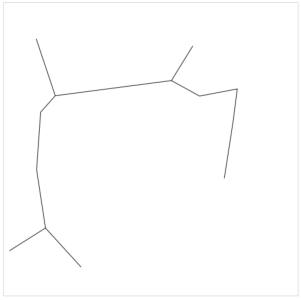
Figure 4.2: No obvious difference



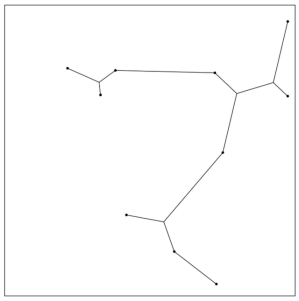
(a) minimal Steiner tree (length 203.425)



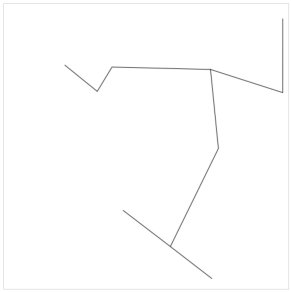
(b) minimal spanning tree (length 213.477)



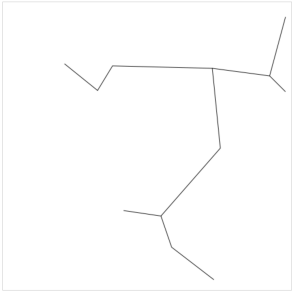
(c) Steiner tree we generate (length 208.934)



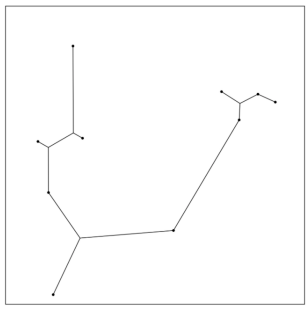
(d) minimal Steiner tree (length 205.996)



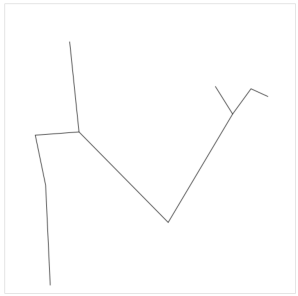
(e) minimal spanning tree (length 217.625)



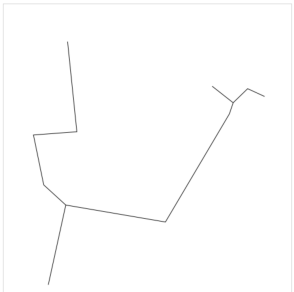
(f) Steiner tree we generate (length 210.584)



(g) minimal steiner tree (length 203.772)



(h) minimal spanning tree (length 214.904)



(i) Steiner tree we generate (length 209.197)

Figure 4.3: Better than MST

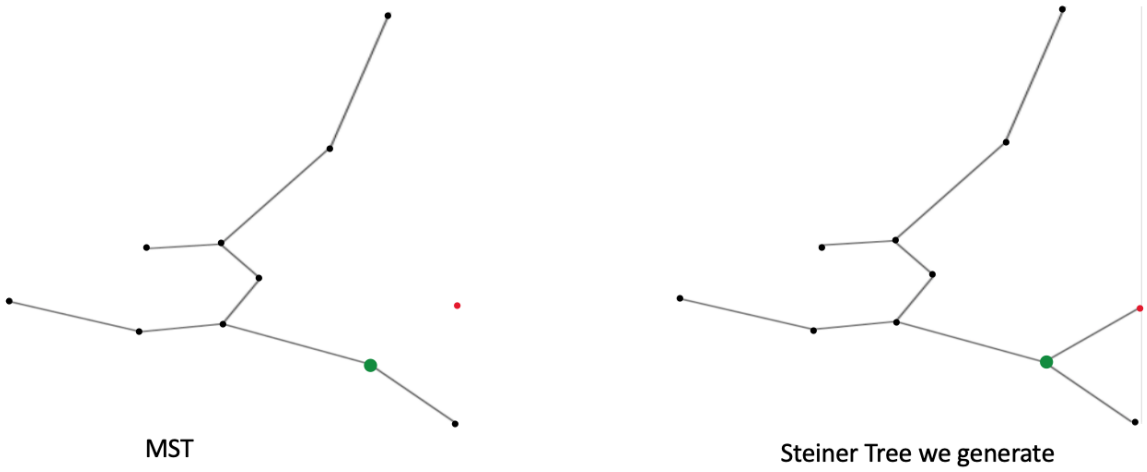


Figure 4.4: Agent chooses an awful points

4.2 Experiment Learning

After our analysis, we find some problems with our method and specific areas that needed improvement. The first Problem is the Graph Embedding. We set the graph embedding as the average nodes embedding over the node set $V_i^{'}$, which leads that each original node has the same weight as each node in the candidate set. However, the number of nodes in the candidate set PSP_i is much larger than the number of nodes in the original node set V_i, which makes the whole candidate set has a larger effect on the graph embedding. In this condition, when we calculate the compatibility between the graph embedding and the node embedding from the candidate set, the whole candidate set has a larger effect on the compatibility than the original node set. The compatibility focuses more on the relationship between nodes in the candidate set rather than the relationship between nodes in the candidate set and the nodes in the original node set. So sometime it may select some points having no relationship with the original node set. An example is illustrated in Figure 4.4, the agent selects the red point as the Steiner points. Another explanation for the agent selecting the red point

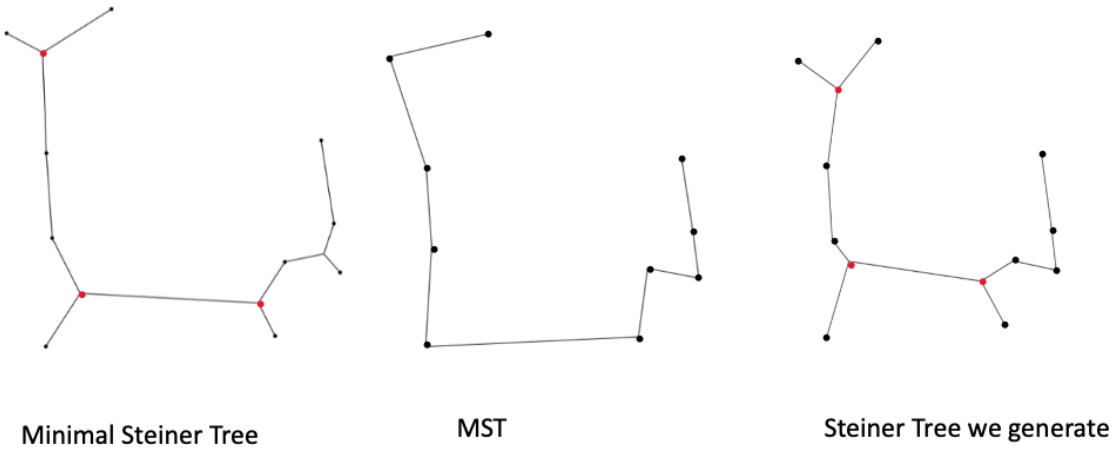


Figure 4.5: Quality of candidate set generated by arc approximate approach: The Steiner points Selected by the agent is near the optimal Steiner points. The model will have a better performance if a preciser arc approximate approach used.

in Figure 4.4 is that the agent takes the red points as the original point and takes the green point as the Steiner points, which may also be caused by the graph embedding. For the reason that the original point shares the same weight as the nodes in the candidate set in the graph embedding, the agent may not distinguish original points from the nodes in the candidate set in the graph embedding. Using the average nodes embedding over the original node set V_i may be helpful to improve the performance. The agent will distinguish original points with the nodes in the candidate set in the graph embedding and focus on the relationship between nodes in the candidate set and the whole original node set.

The second problem is the arc approximate. We only use six points to replace the Steiner Arc. The agent sometimes locates the Steiner Arc and selects the best one generated by arc approximate approach. However, the quality of the candidate set generated by arc approximate approach affects the quality of the Steiner points. An example is illustrated in Figure 4.5. If we can use more points to replace the Steiner Arc, the model may have better performance. We will discuss the number of points we

need in further research and try to make a balance between accuracy and efficiency. We will also decide the number of K that will be used in the K nearest nodes in further research.

Chapter 5

CONCLUSION

In this thesis, we propose a deep reinforcement learning model to solve the Steiner problem. Compared to the existing exact and approximate algorithms based on enumeration methods, our method can produce Steiner trees without incurring high computation and memory costs. Evidenced by experiments, our method can produce Steiner trees that are non-trivially better than the minimum spanning tree, which shows that deep reinforcement learning is a possible and potential approach to solve the Steiner tree problem.

In future work, we will attempt to improve the performance of the model by solving the problems we have observed through experiment learning. Furthermore, motivated by that Graph Neural Network [29] (GNN) can be applied to solve NP-hard combinatorial optimization problems [9], we will attempt to introduce the GNN to our method. Our goal will not be limited to generate a solution better than the minimum spanning tree. We aim to propose a method that can produce a Steiner tree comparable to the optimal Steiner tree.

REFERENCES

[1] Jarník et al. "O minimálních grafech, obsahujících n daných bodů". cze. In: *Časopis pro pěstování matematiky a fysiky* 063.8 (1934), pp. 223–235. URL: http://eudml.org/doc/25703.

[2] Machine Hsie, Ming-Yen Wu, and Chun Yen Huang. "Optimal urban sewer layout design using Steiner tree problems". In: *Engineering Optimization* 51.11 (2019), pp. 1980–1996.

[3] Xiangyu Han et al. "Distribution network planning study with distributed generation based on Steiner tree model". In: *2014 IEEE PES Asia-Pacific Power and Energy Engineering Conference (APPEEC)*. IEEE. 2014, pp. 1–5.

[4] Manki Min et al. "Improving construction for connected dominating set with Steiner tree in wireless sensor networks". In: *Journal of Global Optimization* 35.1 (2006), pp. 111–119.

[5] Michael R Garey, Ronald L Graham, and David S Johnson. "The complexity of computing Steiner minimal trees". In: *SIAM journal on applied mathematics* 32.4 (1977), pp. 835–859.

[6] David M Warme, Pawel Winter, and Martin Zachariasen. "Exact algorithms for plane Steiner tree problems: A computational study". In: *Advances in Steiner trees*. Springer, 2000, pp. 81–116.

[7] Michał Bereta. "Baldwin effect and Lamarckian evolution in a memetic algorithm for Euclidean Steiner tree problem". In: *Memetic Computing* 11.1 (2019), pp. 35–52.

[8] Irwan Bello et al. "Neural combinatorial optimization with reinforcement learning". In: *arXiv preprint arXiv:1611.09940* (2016).

[9] Wouter Kool, Herke van Hoof, and Max Welling. "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations*. 2018.

[10] Chaitanya K Joshi, Thomas Laurent, and Xavier Bresson. "An efficient graph convolutional network technique for the travelling salesman problem". In: *arXiv preprint arXiv:1906.01227* (2019).

[11] Iddo Drori et al. "Learning to solve combinatorial optimization problems on real-world graphs in linear time". In: *arXiv preprint arXiv:2006.03750* (2020).

[12] Mohammadreza Nazari et al. "Reinforcement learning for solving the vehicle routing problem". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018, pp. 9861–9871.

[13] Hao Lu, Xingwen Zhang, and Shuang Yang. "A learning-based iterative method for solving vehicle routing problems". In: *International Conference on Learning Representations*. 2019.

[14] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[15] Marcus Brazil et al. "On the history of the Euclidean Steiner tree problem". In: *Archive for history of exact sciences* 68.3 (2014), pp. 327–354.

[16] Marcus Brazil and Martin Zachariasen. *Optimal Interconnection Trees in the Plane: Theory, Algorithms and Applications*. Vol. 29. Springer, 2015.

[17] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[18] Sebastian Thrun and Michael L Littman. "Reinforcement learning: an introduction". In: *AI Magazine* 21.1 (2000), pp. 54–70.

[19] Kai Arulkumaran et al. "Deep reinforcement learning: A brief survey". In: *IEEE Signal Processing Magazine* 34.6 (2017), pp. 26–38.

[20] Hanjun Dai et al. “Learning combinatorial optimization algorithms over graphs”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 2017, pp. 6351–6361.

[21] Richard Bellman. “A Markovian decision process”. In: *Journal of mathematics and mechanics* 6.5 (1957), pp. 679–684.

[22] Nina Mazyavkina et al. “Reinforcement learning for combinatorial optimization: A survey”. In: *arXiv preprint arXiv:2003.03600* (2020).

[23] Yoshua Bengio, Aaron Courville, and Pascal Vincent. “Representation learning: A review and new perspectives”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8 (2013), pp. 1798–1828.

[24] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. “Neural machine translation by jointly learning to align and translate”. In: *3rd International Conference on Learning Representations, ICLR 2015*. 2015.

[25] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems* 30 (2017), pp. 5998–6008.

[26] Alexander Miller et al. “Key-Value Memory Networks for Directly Reading Documents”. In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 2016, pp. 1400–1409.

[27] Sergey Ioffe and Christian Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”. In: *International conference on machine learning*. PMLR. 2015, pp. 448–456.

[28] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[29] Keylu Xu et al. “What Can Neural Networks Reason About?” In: *ICLR 2020* (2020).