Solving the Node-Weighted Steiner Tree Problem using Reinforcement Learning

Zongbo Yang

Department of Big Data Statistics, Guizhou University of Finance and Economics, Guizhou

Received: 09 August 2024/ Revised: 16 August 2024/ Accepted: 20 August 2024/ Published: 31-08-2024

Abstract**—** *The node-weighted Steiner tree problem is a significant issue in network design, with broad applications including telecommunications network construction, offshore oilfield development planning, and wireless ad-hoc networks. The objective of the node-weighted Steiner tree problem is to find a subtree within a given undirected graph that has the minimum total weight and includes all specified terminals. This problem is NP-hard, typically requiring complex algorithm design and exponential time. We have combined graph neural networks and deep reinforcement learning techniques to propose a new solution method. By encoding the structural information of the graph into vector form through graph neural networks and self-attention networks, and optimizing decisions using a reinforcement learning strategy network, we efficiently construct the desired node-weighted Steiner tree. To validate the model's effectiveness, we conducted extensive experiments on various types and sizes of generated graphs, covering different numbers of endpoints. The results show that in scale-free networks (BA), our algorithm performs equivalently to SCIP, with both achieving a Gap value of 0. In small-world networks (WS) and complete graphs (K), our algorithm closely matches SCIP's performance, with the highest Gap values being 0.3% and 1.76%, respectively, indicating that our algorithm can closely approximate SCIP's performance in handling these network structures., In random graphs (ER) and random regular graphs (RR), while there is a performance discrepancy between our algorithm and SCIP, the maximum gap does not exceed 5.76%. These findings demonstrate that our algorithm performs well in solving the node-weighted Steiner tree problem.*

Keywords**—** ***Node-Weighted Steiner Tree; Reinforcement Learning; Graph Neural Networks; Combinatorial Optimization.***

I. INTRODUCTION

Network design problems are prevalent across various practical applications, with the Node-Weighted Steiner Tree Problem (NWSTP) being a particularly significant one. The NWSTP was first introduced by Segev A[1] in 1987 and has found extensive applications in real-world scenarios. For example, in logistics and transportation, companies can use NWSTP to optimize delivery routes by identifying the most efficient transfer points and routing paths, thereby reducing transportation time and costs. In offshore oil field development[2], the problem can be modeled where edges represent pipelines and nodes represent potential platform locations. Here, the rewards indicate the net difference between expected revenue from a site and the construction cost, while edge costs represent installation and maintenance expenses. In the field of network communications[3], multicast communication often employs tree structures known as multicast trees. When modeling a network as a graph, a multicast tree can be seen as a node-weighted Steiner tree. In this model, the multicast sender and receivers correspond to the endpoints of the Steiner tree, while other nodes involved in constructing the tree are referred to as Steiner nodes. These diverse applications underscore the importance of the NWSTP in optimizing complex networks and resources, highlighting the need for effective solutions and advancements in algorithmic strategies.

The definition of the Node Weighted Steiner Tree Problem (NWSTP) is as follows: In an undirected graph $\mathrm{G} = (\mathrm{V}, \mathrm{E}, \omega)$, each node $v \in \mathrm{V}$ and each edge $\mathrm{e} \in \mathrm{E}$ are assigned a non-negative weight $\omega(v)$ and $\omega(\mathrm{e})$), respectively. Given a graph G and a set of terminal nodes T, where $\mathrm{T} \subseteq \mathrm{V}$, the task is to find a subtree G' with the minimum total weight such that G includes all the terminal nodes in T, $T \in \mathrm{G}'$. The formula representation is as follows:

$$C(S, G)_{min} = \sum_{e \in s} \omega(e) + \sum_{v \in s} \omega(v) \quad (1)$$

Where S represents the set of nodes used to construct the Node Weighted Steiner Tree.

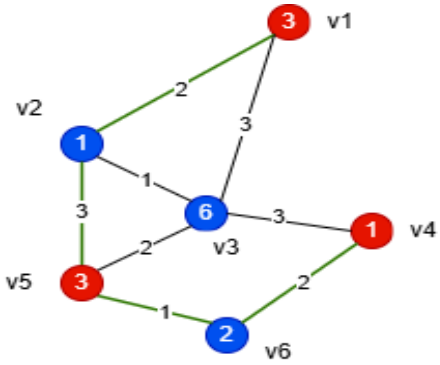

FIGURE 1: Node weighted Steiner Tree

The optimal solution diagram for the Node Weighted Steiner Tree, with endpoints represented by red nodes, and the optimal solution connected by green edges, connecting nodes {v1, v2, v5, v6, v4} with a total weight of 1+2+2+1+3+3+1+2+3 = 18, where {v2, v6} are Steiner points.

In recent years, numerous algorithms have been proposed to solve the Node Weighted Steiner Tree Problem (NWSTP). For instance, Y Sun and S Halgamuge[4] introduced a Physarum-inspired heuristic algorithm for solving NWSTP; S Angelopoulos[5]proposed an approximation algorithm with a ratio of $o(min\{\log \alpha, \log k\})$, where α is the ratio of the maximum to minimum node weights, and k is the number of terminals; E D Demaine [6] developed an approximation algorithm with a ratio of 3; T Erlebach and A Shahnaz[3] offered an approximation algorithm with a ratio of 0.775d, where d is the maximum degree; A Buchanan et al. [7] extended the Steiner tree algorithm by Dreyfus and Wagner[8], proposing an exact algorithm with a provably worst-case runtime of $O(n^3)$ in instances with a finite number of terminals and n nodes; R Cordone and M Trubia [2] suggested an exact algorithm, however, their experiments did not involve graphs with degrees greater than 2. Despite the progress made by existing methods, they have limitations in dealing with highly complex graphs, providing approximate solutions only under specific conditions, and longer computational times. With the advancement of graph neural networks and reinforcement learning technologies, new solution strategies have emerged. This paper will explore an efficient and precise method to solve the NWSTP by leveraging the strengths of these two technologies, aiming to break through the limitations of traditional algorithms.

II. DEEP REINFORCEMENT LEARNING

Due to the NP-hard nature of the Node Weighted Steiner Tree Problem (NWSTP) and the challenges in obtaining accurate node labels, we have chosen to employ the Actor-Critic algorithm from deep reinforcement learning[9] to address the NWSTP. In this algorithm, the agent's decision-making for the next step does not depend on information about future actions because all necessary information is already contained in the current state, which is directly determined by previous actions. In reinforcement learning, the Actor-Critic algorithm features two key roles: the Actor selects actions based on the current state, while the Critic evaluates these actions and provides feedback to optimize the choices. This cooperative mechanism is central to many successful reinforcement learning algorithms and enables the agent to continuously make wise decisions and improve performance.

When addressing the NWSTP, we treat it as a sequential decision-making problem, starting from a node and incrementally building the tree with the minimum weight sum while ensuring all specified endpoints are included. To handle different types and sizes of graphs, we define six core elements: state space, state transitions, action space, rewards, policies, and termination conditions.

State Space: At time t, the state is a tensor describing the current environment, including the graph G, the partial solution set S, the set of unadded endpoints, and node features. When an endpoint is added to the solution set, the related node features change, and each addition of a node updates the state, collectively forming the state space.

Action Space: At time t, the actions that the agent can execute are the set of candidate nodes at that moment. Candidate nodes are required to connect with nodes in the partial solution set and not be part of it, ensuring the tree structure does not form a cycle. The action space at the initial moment $t = 0$ is set as the endpoint set T, thus defining the action space representation at time t as:

$$A_t = \begin{cases} \{v \in T | T \in V\}, \text{if } t = 0 \\ \{v | v \notin S \wedge \exists u \in S: (u, v) \in E\}, \text{otherwise} \end{cases} \quad (2)$$

State Transitions: The state transitions for the NWSTP on the graph are deterministic, without any probabilistic elements. When a node v is added to the partial solution S, the state changes accordingly.

Rewards: The reward mechanism in our model is designed based on the current state and actions taken, aiming to guide the agent toward finding the path with all endpoints included and the minimum weight. When a new node v is added to the partial solution set S, moving the state from s_t to s_{t+1}, rewards are calculated based on the type of node v(endpoint or non-endpoint) considering, the reward function $r(S, v)$ considers two main factors:(1)The change in the cost function $C(S, G)$, which measures the overall path cost after adding the new node v; (2)The path weights from node v to other endpoints, reflecting the cost from the current node to reach endpoints. The design aims to incentivize the model to achieve the minimum weight tree, with the specific reward function as follows:

$$R(s, v) = \begin{cases} C(S', G) - C(S, G) + |x_v| + N, v \in T \\ C(S', G) - C(S, G) + |x_v| + N + c, v \notin T \end{cases} \quad (3)$$

Where S' represents the partial solution set after adding node v, $|x_v|$ is the sum of distances from the node to the endpoints, N is the weight of the currently added node, and c is a constant.

Policy: $a_t \sim$Categorical $\pi_\theta(v|s;\theta)$, where the policy network computes the probability distribution of actions given the state S, $\pi_\theta(v|s;\theta)$, and θ represents the network parameters. Subsequently, actions are sampled from the discrete categories based on the probability distribution. This method of action selection balances exploration and exploitation during training. Traditional greedy algorithms tend to get stuck in local optima, while the sampling method based on probability distribution introduces randomness, allowing the policy network to explore the state space more effectively. Introducing randomness permits the policy to attempt various possible action paths at different training stages, thereby increasing the likelihood of finding the global optimal solution.

Termination Condition: The process terminates when all endpoints are added to the solution set S, $T \in S$.

III. MODEL

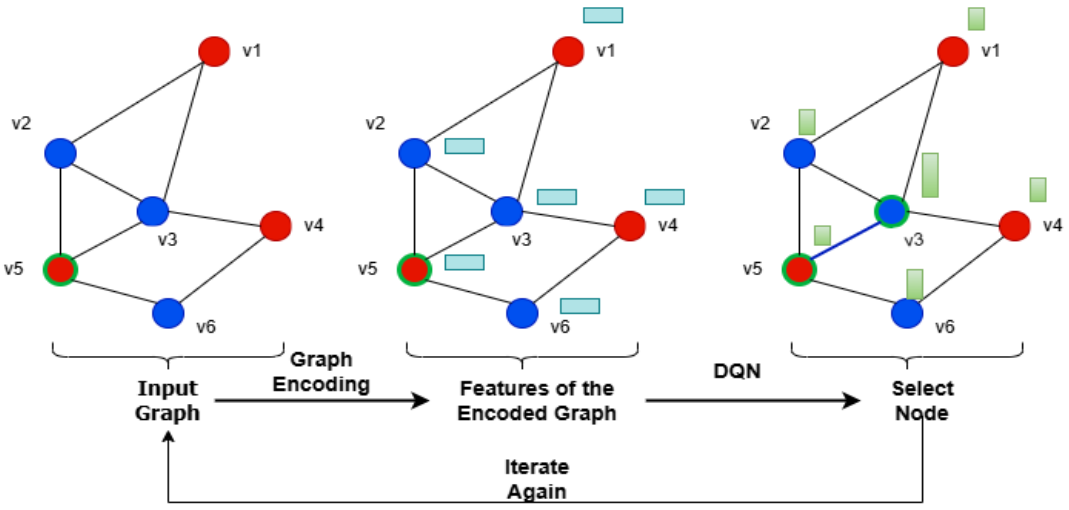

FIGURE 2: MODEL DIAGRAM

3.1 Model overview:

Figure 2 provides an overview of the model. Initially, a graph is input where endpoints are represented by red nodes. Subsequently, the graph's information, the partial solution set, and predefined node feature information are fed into the model. The model employs a novel graph encoding method to compactly encode the input information into node state tensors (as shown by the colored bars next to the nodes in the diagram). The encoded tensor is then input into a DQN, which outputs scores for candidate nodes, represented by green bars. The height of the bar indicates the node's score, with higher bars representing higher scores. Finally, the model incrementally adds the highest-scoring nodes according to a greedy strategy to construct the solution set. The specific construction process is shown in Algorithm 1.

Algorithm 1

Input: In the graph G, a random initial node $v_0 \in$ T.

S = {v_0} , $V(G,t) = S$

While T $\notin S$ **do** :

Obtain the action space A_t.

$a_t \in A_t$: a_t ~Categorical $\pi(v|s;\theta)$, Select an action a_t.

$S = S + a$

end while

return S

3.2 Node feature construction:

Node features have a direct impact on model performance. Therefore, we constructed 20 initial features for each node in the graph. The specific features are detailed in Table 1.

TABLE 1
FEATURES

index	feature description
0	Whether it is in the partial solution: 1 if the node is in the solution, otherwise 0.
1	Whether it is an endpoint: 1 if the node is an endpoint, otherwise 0.
2	Node weight.
3	Edge weight of the node.
4	PageRank: Reflects the importance of the node in the graph based on connection quality and structure.
5	Degree centrality: The number of direct connections of the node.
6	Clustering coefficient: Measures the density of connections between the node's neighbors.
7	Closeness centrality: The reciprocal of the average distance from the node to all other nodes.
8	Betweenness centrality: The number of shortest paths passing through the node.
9	Eigenvector centrality: The influence of the node in the graph.
10	Distance to the nearest endpoint.
11	Distance to the second closest endpoint.
12	Distance to the third closest endpoint.
13	Minimum weight of neighboring nodes.
14	Second smallest weight of neighboring nodes.
15	Edge weight to neighbor node 1.
16	Edge weight to neighbor node 2.
17	Edge weight to neighbor node 3.
18	Weight of second-order neighbor node 1.
19	Weight of second-order neighbor node 2.
20	Weight of second-order neighbor node 3.

We constructed features for each node, including commonly used graph theory metrics such as degree centrality, PageRank, and clustering coefficient. When solving the NWSTP problem, we particularly focus on the shortest distance from the nodes to the endpoints. These distances are represented in the matrix $X \in R^{|V|\times|T|}$, where V denotes the number of nodes v, and T denotes the number of endpoints T. To maintain consistent matrix dimensions and to eliminate the influence of endpoints that are too far away from the nodes, we set T in $X \in R^{|V|\times|T|}$ to a constant parameter k. Since constructing the tree is a Markov decision process, once an endpoint is added to the partial solution set, it loses its value. Therefore, k refers to those endpoints that have not been added to the partial solution set. When the number of unadded endpoints is less than k, we pad with zeros. Experimental analysis shows that the model performs best when $k = 3$.

Next, we considered the edge weights of nodes and their first-order neighbors, as these weights are crucial for constructing the solution set. Therefore, we selected the three neighbors with the smallest weights and their edge weights as features, padding

with zeros if there were fewer than three. Additionally, to enhance the model's ability to capture longer path information, we introduced the weights of each node's second-order neighbors. Specifically, we chose the weights of the three second-order neighbors with the smallest weights as features.

Finally, the topology of the subgraph formed by the partial solution set influences the selection of the next node. To enable the model to recognize this structure, we encoded the subgraph information into feature tensors. By integrating the graph tensor and subgraph tensor through a fully connected neural network, we extracted more comprehensive information about the graph.

3.3 Encoding and decoding:

Due to the STP problem involving unstructured data, the probability of selecting candidate nodes is influenced by multiple factors, including the graph's structure, the positions of the terminals, and their weights. Our goal is for the network to accurately understand the current state at any given time and calculate the selection probability for each candidate node. To achieve this, we designed an encoding-decoding architecture aimed at effectively handling these complex factors. This architecture is mainly divided into two parts: the encoder and the decoder.

Encoder: The encoder is responsible for converting the current state information into a representation suitable for processing by a machine learning model. Its role is to transform the graph structure, terminal positions, and other relevant information into tensors. The encoder's input consists of the graph, the partial solution set, and features, and its output is a state tensor that contains all the key information of the current state.

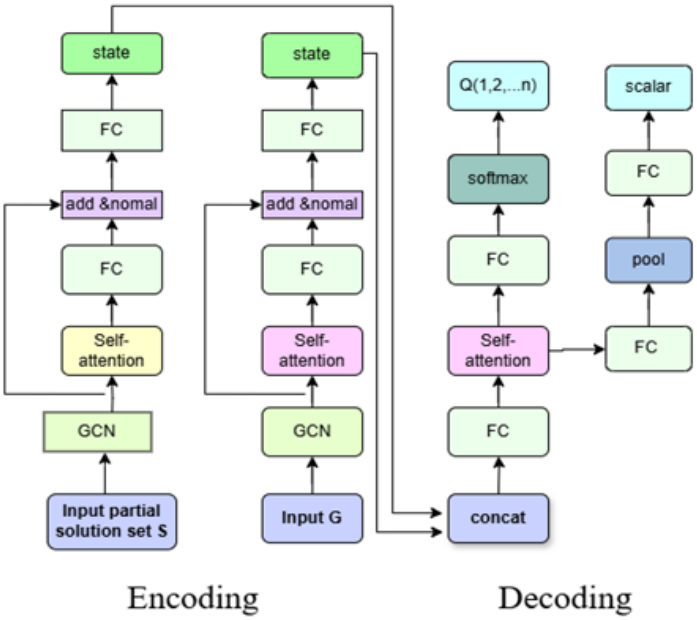

FIGURE 2: Encoding-Decoding architecture diagram

Decoder: The task of the decoder is to learn and understand the state vector generated by the encoder, in order to predict the selection probability and reward for each candidate node. It receives the state tensor provided by the encoder as input and outputs the selection probability and reward for each candidate node.

The encoder-decoder architecture enables deep reinforcement learning models to effectively utilize the complex structures and multiple influencing factors in graph-structured data, thereby enhancing the accuracy and efficiency of solving the node-weighted Steiner tree problem. This approach is not only applicable to addressing the node-weighted Steiner tree problem but also provides a methodology for solving other combinatorial optimization problems.

3.3.1 Encoding process:

Our model takes a graph's feature matrix$X$ and adjacency matrix A as inputs. Initially, we utilize a Graph Convolutional Network (GCN) to propagate features, with the layer-to-layer propagation rule defined as follows:

$$H^{(l+1)} = \sigma\left(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \quad (4)$$

where $H^{(l)}$ is the feature matrix at layer l; $H^{(0)} = X$; $\tilde{A} = A + I$ is the adjacency matrix (including self-connections); $\tilde{D} = \text{diag}\left(\sum_j \tilde{A}_{ij}\right)$ is the degree matrix; $W^{(l)}$ is the weight matrix learned by the convolution layer; σ represents the activation function.

To enhance the nonlinear transfer capability, several fully connected layers (FC) are added after the convolution to more effectively learn the graph's feature embeddings.

$$H^{(l+1)} = H^{(l)} W + b \tag{5}$$

where W is the weight matrix of the linear layer, b is the bias vector, and $H^{(l)}$ is the feature matrix output from the convolution layer.

To better capture the complex relationships between nodes and their neighbors, we employ a multi-layer, multi-head attention network. In this network, we take the output of the previous layer $X = H^{(l)}$ as input, and process it through multiple attention heads, each with its own query, key, and value matrices $W_{Q_i}, W_{K_i}, W_{V_i}$. For the input features X, where X is the output from the previous layer, the attention computation for each head is as follows:

$$\mathrm{Z}^i = \text{softmax}\left(\frac{\mathrm{Q}^i(\mathrm{K}^i)^{\mathrm{T}}}{\sqrt{\mathrm{d_k}}}\right)\mathrm{V}^i \tag{6}$$

where $\mathrm{Q}^i = \mathrm{XW}_{\mathrm{Q}^i}$, $\mathrm{K}^i = \mathrm{XW}_{\mathrm{K}^i}$, $\mathrm{V}^i = \mathrm{XW}_{\mathrm{V}^i}$, and $\mathrm{d_k}$ represents the dimension of the key. Next, we concatenate the outputs from all heads and then apply a linear transformation to obtain the final multi-head attention output:

$$\mathrm{H}^{(\mathrm{l})} = \text{Concat}\left(\mathrm{Z}^1, \mathrm{Z}^2, \ldots, \mathrm{Z}^{\mathrm{h}}\right)\mathrm{W} \tag{7}$$

To address the problems of vanishing and exploding gradients in deep neural networks, we introduce residual connection technology. The residual connection formula is as follows:

$$y = F(X, W) + X \tag{8}$$

where $F(X, W)$ represents the features computed through several layers, and X is the input feature.

To enhance the stability and training efficiency of the model, we use layer normalization. The layer normalization process includes calculating the mean and variance of the input features, normalizing the input, and finally scaling and shifting:

$$\hat{\mathrm{x}} = \frac{\mathrm{x} - \mu_{\mathrm{L}}}{\sqrt{\sigma_{\mathrm{L}}^2 + \epsilon}} \tag{9}$$

$$\mathrm{y} = \gamma\hat{\mathrm{x}} + \beta \tag{10}$$

where μ_L and σ_L^2 are the mean and variance of the input features, ϵ is a small constant to prevent division by zero, andγ and β are learnable parameters.

3.3.2 Decoder process

Our decoding section employs two network structures and shares some parameters between them. This design aims to improve computational efficiency by reducing the number of model parameters and to lower the risk of overfitting when handling similar tasks or inputs. The specific structure is shown as Figure 4.

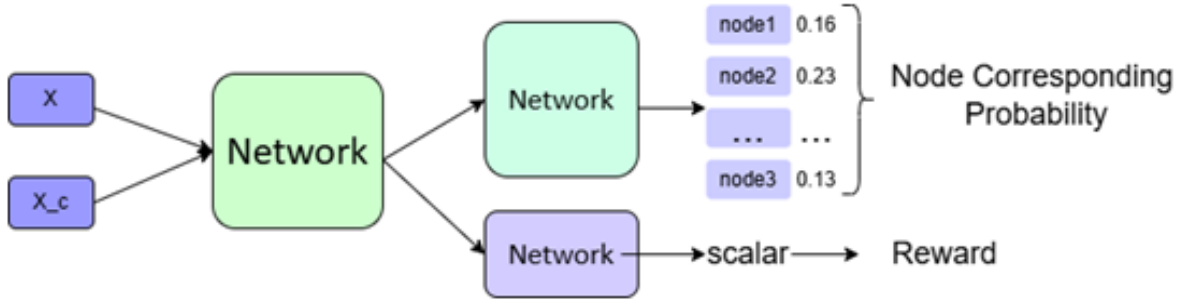

FIGURE 3: Structure of shared parameters in the decoding network

First, we input the graph G and its adjacency matrix A, along with the graph G_c composed of partial solution sets and its corresponding adjacency matrix A_c into the decoder, thus obtaining the encoded feature representations$X$ and X.

Next, we combine the feature information of graph G and graph G_c to obtain the current state information of the graph:

$$\mathrm{X} = \mathrm{Concat}(\mathrm{X}, tile(mean(X_c, axis = 0), (\mathrm{N}, D)) \tag{11}$$

where N is the number of nodes, D is the dimension of features, Concat represents concatenation, and $tile$indicates replicating the feature N times, resulting in $\in R^{N\times 2D}$.

Subsequently, X is passed through a fully connected layer and a multi-head self-attention layer to further extract features, concluding the sharing of network parameters.

Following this, the feature matrix obtained from the previous step undergoes various processes to produce different outputs. First, it outputs the probability of a node being selected:

$$Q(S, \theta) = softmax(X \cdot W + b) \tag{12}$$

Next, it outputs the predicted expected reward:

$$\mathrm{R_{total}} = \mathrm{sum}(\mathrm{Pool}(\mathrm{X} \cdot \mathrm{W_1} + \mathrm{b_1}) \cdot \mathrm{W_2} + \mathrm{b_2}) \tag{13}$$

Where Pool represents average pooling.

3.4 Strategy optimization:

After sampling N node-weighted Steiner trees using strategy π_θ, we obtained N sets of trajectories:

$$\{P_t, R_{t-1}, r_t, m_t\}_{\mathrm{t}=0}^{\sum_0^n T_n} \tag{14}$$

where T_n is the final time step of the nth trajectory, R_t is the expected reward at time t, r_t is the actual reward received after executing an action at timer_t, and m_t indicates whether a solution set has been found at time t, if found, $m_t = 0$; otherwise, $m_t = 0$.

The gradient of the average reward $J(\theta)$ can be approximated using the policy gradient theorem:

$$\nabla \mathrm{J}(\theta) = \sum_{t=0}^{\sum_0^n T_n} (r_t + m_t * R_t - R_{t-1}) * P_t \tag{15}$$

We use the Adam optimizer, and the training process is detailed in Algorithm 2.

Algorithm 2

For _ in 1 to N do:
Initialize three empty lists
$\mathrm{e_n}$ = [], Record the logarithm of the probability of choosing each action, $log\ p_t$
rewards = [], Record the rewards obtained from executing each action
mask = [] ,Record whether a Steiner tree is completed, with the mask for the last action set to 0 and the rest set to 1.
For episode in range (batch, batch + N):
Read graph G, randomly initialize node $v_0 \in T$ and construct subgraph
$V(G\_t) = S$
While $T \notin S$ **do:**
Obtain the action space A_t
$a_t \in A_t$: a_t~Categorical $\pi(v|s;\theta)$, Select a node and record the output R of the evaluation network.
Execute action a_t, record the actual reward r_t, and the probability p_t of choosing action a_t.
Record the expected reward $V^\theta(s_t)$ output by the network
s = s + a
end while
end for
batch + = N, N as a parameter

$$\mathrm{Loss} = \sum_{t=0}^{\sum_0^n T_n} \left(r_t + m_t * V^\theta(s_t) - V^\theta(s_{t-1})\right) * P_t$$

$\theta \leftarrow \mathrm{Amda}(\theta, \nabla_{Loss})$
end for

IV. EXPERIMENT AND ANALYSIS

4.1 Dataset:

In evaluating our model, we used a variety of graph generation rules, including Random Regular graphs (RR)[11], Erdős-Rényi graphs (ER)[12], small-world networks (WS)[13], scale-free networks (BA)**Error! Reference source not found.**, and complete graphs (K). F or each generation rule, we generated instances with a node count of n. These generation rules are widely applied in solving various graph combinatorial optimization problems, such as Minimum Vertex Cover (MVC) and Maximum Cut (MAXCUT). For the Node Weighted Steiner Tree problem, the number and positions of terminals in the graph significantly influence the instances; thus, we introduced a parameter mmm, representing the probability of each vertex becoming a terminal. We set m values at 0.3, 0.6, and 0.9, and node counts n n at 20, 30, 40, and 50. By varying m and n, we generated different instances to increase the complexity of the generated graph instances. Lastly, we randomly assigned a weight between 1 and 3 to each edge and node in the graph. Through this method, we were able to generate test instances of certain complexity, effectively evaluating the performance of the model, as shown in Table 2.

TABLE 2
EXPERIMENTAL DATA

Type of Graph	Number of Nodes	Node Weight Range	Edge Weight Range	Terminal Ratio
RR/WS/BA/ER/K	20	1,3	1,3	0.3, 0.6, 0.9
	30	1,3	1,3	0.3, 0.6, 0.9
	40	1,3	1,3	0.3, 0.6, 0.9
	50	1,3	1,3	0.3, 0.6, 0.9

4.2 Comparison methods:

SCIP-Jack[10] is a software package designed for the classic Steiner tree problem in graphs. It is also capable of solving

fourteen related Steiner tree problems, such as prize-collecting Steiner trees, node-weighted Steiner tree problems, etc. Currently, SCIP-Jack is the optimal solver for the Node-Weighted Steiner Tree Problem.

4.3 Parameter settings:

During the graph embedding phase, we extract features using three layers of Graph Convolutional Networks (GCN), one fully connected neural network layer, and one multi-head self-attention network layer (with eight heads). In the subgraph embedding phase, feature extraction is performed using two layers of GCN, one fully connected neural network layer, and one multi-head self-attention network. Subsequently, the features from these two phases are merged and further processed through one convolutional layer and two fully connected network layers to enhance feature expression. To improve the model's nonlinearity, we have chosen the hyperbolic tangent function (tanh) as the activation function.

Considering the limitations of computational resources, we trained graphs with a node count ranging from 20 to 50, totaling 20,000 graphs. The experimental settings include: a random seed set to 42 to ensure reproducibility; a decay factor γ set at 0.99, used to regulate the learning rate; parameter N set at 5, and parameter$C$ set at 2. The model optimization is managed by the Adam optimizer, with an initial learning rate η of 0.01, which is gradually reduced to 0.00001, implementing a progressively decaying learning rate strategy.

4.4 Comparison of Experimental Results:

In the experiment, we evaluated the gap between the model proposed in this paper and SCIP-Jack. To eliminate random errors, we tested 100 instances for each type of graph generated. We used the formula $\mathrm{Gap} = (\mathrm{RL}/\mathrm{obj} - 1) \times 100\%$ to determine the gap, where RL is the result calculated by our algorithm, and obj represents the result calculated by the $SCIP$ algorithm. We calculated the mean of these instance results as the outcome to measure the performance of each algorithm.

TABLE 3
EXPERIMENTAL RESULTS

Number of Nodes	Terminal Ratio	RR_Gap	WS_Gap	BA_Gap	ER_Gap	K_Gap
20	0.3	2.47%	0.19%	0.00%	0.53%	0.99%
	0.6	2.50%	0.03%	0.00%	0.55%	1.47%
	0.9	3.90%	0.11%	0.00%	3.13%	1.72%
30	0.3	2.74%	0.26%	0.00%	1.51%	1.17%
	0.6	4.07%	0.04%	0.00%	2.51%	1.76%
	0.9	4.94%	0.20%	0.00%	4.20%	1.71%
40	0.3	3.22%	0.07%	0.00%	2.55%	1.51%
	0.6	5.18%	0.02%	0.00%	4.12%	1.76%
	0.9	4.56%	0.01%	0.00%	5.76%	1.47%
50	0.3	3.62%	0.03%	0.00%	3.81%	1.75%
	0.6	5.71%	0.01%	0.00%	5.62%	1.62%
	0.9	4.62%	0.34%	0.00%	0.83%	0.99%

The experimental results from Table 3 show that our algorithm performs on par with SCIP in solving scale-free network graphs (BA), with a Gap value consistently at 0. In small-world network graphs (WS), our algorithm generally performs better, with Gap values very low, the highest being only 0.3%, indicating that our algorithm closely approximates the performance of the SCIP algorithm in handling small-world networks. For random graphs (ER) and random regular graphs (RR), although our algorithm shows some difference from SCIP, the largest gap does not exceed 6%. In complete graphs (K), our algorithm's performance is close to SCIP, with the largest Gap value at 1.76%, which is not significant, and the Gap does not increase notably with more nodes. These comparisons indicate that our algorithm is effective in solving the Node Weighted Steiner Tree problem.

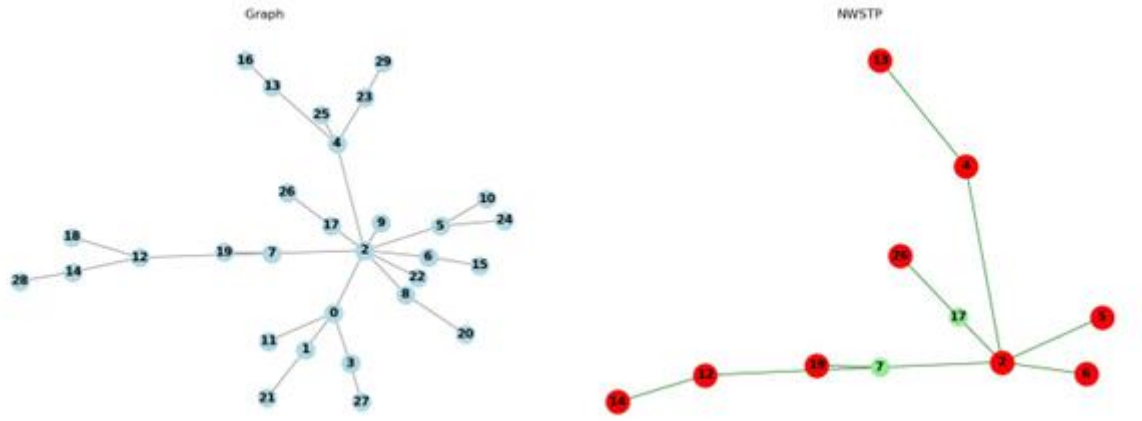

FIGURE 4: Display of solution results in BA graphs

Figure 5 demonstrates the solution of a scale-free Barabási–Albert (BA) network instance with 30 nodes where the terminal ratio is 0.3. In this figure, the red nodes are terminals, and the green nodes are Steiner points. This visualization helps in understanding how the algorithm identifies and utilizes Steiner points to effectively connect all the terminals in the most optimal manner.

V. CONCLUSION

We have proposed an innovative solution method for the node-weighted Steiner tree problem by integrating graph neural networks and reinforcement learning techniques. Initially, we employ graph neural networks and multi-head self-attention networks to encode complex graph structure data into tensor representations, aimed at capturing the deep relationships between nodes in the graph. Subsequently, by combining reinforcement learning techniques, we optimize the decision-making process

through policy networks, progressively constructing node-weighted Steiner trees, thus achieving efficient solutions for the NWSTP.

To validate the effectiveness and robustness of our proposed method, we conducted experiments using five different types of generated graphs. These graphs not only include various types and sizes but also simulate different terminal count scenarios, ensuring the complexity and comprehensiveness of the experimental conditions. The results indicate that our algorithm has achieved satisfactory outcomes.

Looking ahead, although this research has already achieved certain results, there is still room for improvement in the scalability and universality of the algorithm. Future work may include: first, exploring more types of graph neural network architectures to adapt to more complex structural changes in graphs; second, optimizing reinforcement learning strategies to enhance the model's adaptability to different tasks; and third, applying the model to more practical scenarios such as social network analysis and bioinformatics to verify its practicality and effectiveness. Through ongoing research and improvement, we hope to achieve deeper breakthroughs in solving graph optimization problems.

ACKNOWLEDGEMENTS

we would like to extend special thanks to D. Rehfeldt, who provided the SCIP-Jack binary files, allowing us to compare our algorithm with SCIP-Jack.

REFERENCES

[1] Segev A. The node-weighted Steiner tree problem[J]. Networks, 1987, 17(1): 1-17.

[2] Cordone R, Trubian M. An exact algorithm for the node weighted Steiner tree problem[J]. 4OR, 2006, 4: 124-144.

[3] Erlebach T, Shahnaz A. Approximating node-weighted multicast trees in wireless ad-hoc networks[C]//Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing: Connecting the World Wirelessly. 2009: 639-643.

[4] Sun Y, Halgamuge S. Fast algorithms inspired by physarum polycephalum for node weighted steiner tree problem with multiple terminals[C]//2016 IEEE Congress on Evolutionary Computation (CEC). IEEE, 2016: 3254-3260.

[5] Angelopoulos S. The node-weighted Steiner problem in graphs of restricted node weights[C]//Scandinavian Workshop on Algorithm Theory. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006: 208-219.

[6] Demaine E D, Hajiaghayi M T, Klein P N. Node-weighted Steiner tree and group Steiner tree in planar graphs[C]//Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36. Springer Berlin Heidelberg, 2009: 328-340.

[7] Buchanan A, Wang Y, Butenko S. Algorithms for node-weighted Steiner tree and maximum-weight connected subgraph[J]. Networks, 2018, 72(2): 238-248.

[8] Dreyfus S E, Wagner R A. The Steiner problem in graphs[J]. Networks, 1971, 1(3):195-207.

[9] Konda V, Tsitsiklis J. Actor-critic algorithms[J]. Advances in neural information processing systems, 1999, 12.

[10] Rehfeldt D, Koch T. Implications, conflicts, and reductions for Steiner trees[J]. Mathematical Programming, 2023, 197(2): 903-966.

[11] Molloy, M.; and Reed, B. 1995. A critical point for random graphs with a given degree sequence. Random Structures and Algorithms, 6(2): 161–180.

[12] Erdos, P.; and R ′ enyi, A. 1960. On the evolution of random graphs. Publ. Math. Inst. Hung. Acad. Sci, 5(1): 17–60.

[13] Watts, D. J.; and Strogatz, S. H. 1998. Collective dynamics of ‘small-world’networks. nature, 393(6684): 440.

[14] Barabási, Albert-László, and Réka Albert. "Emergence of scaling in random networks." science 286.5439 (1999): 509-512