Chair of Operations Research
Department of Mathematics
TU Munich

Classical and Modern Approaches for Solving the Steiner Tree Problem

Interdisciplinary Project (IDP) by Teodora Dobos

Examiner:	Prof. Dr. rer. nat. Andreas S. Schulz
Advisor:	Dr. rer. nat. Daniel Vaz
Submission Date:	June 21, 2022

I hereby confirm that this is my own work, and that I used only the cited sources and materials.

München, June 21, 2022

Teodora Dobos

Abstract

Steiner Tree is a combinatorial optimization problem in which a weighted undirected graph and a subset of nodes, called *terminals*, are given. The objective is to find a tree that spans all terminals and has minimal weight. This problem is NP-hard and, therefore, an optimal solution cannot be computed in polynomial time. Approximation algorithms have been designed to solve the Steiner Tree problem, providing provable guarantees on the distance of the computed solution to the optimal one. Recently, modern techniques that rely on machine learning have been applied to find approximate solutions. For this Interdisciplinary Project (IDP), we implemented and evaluated four classical and modern approaches for solving the Steiner Tree problem. We implemented three classical techniques, namely the *Repetitive Shortest Path Heuristic*, the *Mehlhorn Algorithm* and the *Primal-Dual Method*, and a modern approach based on deep reinforcement learning, called *Cherrypick*.

Zusammenfassung

Steinerbaum ist ein kombinatorisches Optimierungsproblem, bei dem ein gewichteter ungerichteter Graph und eine Teilmenge von Knoten, die so genannten Terminals, gegeben sind. Ziel ist es, einen Baum zu finden, der alle Terminals miteinander verbindet und minimales Gewicht hat. Dieses Problem ist NP-schwer und daher kann eine optimale Lösung nicht in polynomieller Zeit berechnet werden. Zur Lösung des Steinerbaumproblems wurden Approximationsalgorithmen entwickelt, die nachweisbare Garantien für den Abstand der berechneten Lösung zur optimalen Lösung bieten. In der letzten Zeit wurden moderne Techniken, die auf maschinelles Lernen basieren, angewandt, um gute Lösungen zu finden. In diesem interdisziplinären Projekt (IDP) haben wir vier klassische und moderne Verfahren zur Lösung des Steinerbaumproblems implementiert und evaluiert. Wir haben drei klassische Algorithmen implementiert, nämlich die *Repetitive Shortest Path Heuristic*, die *Primal-Dual-Methode* und den *Mehlhorn-Algorithmus*, und ein modernes Verfahren namens *Cherrypick*, das auf Deep Reinforcement Learning basiert.

Contents

Chapter 1

Introduction

In the Steiner Tree problem (STP), the input consists of a graph $G = (V, E)$ with non-negative edge costs $c_e \geq 0$ for all $e \in E$ and a subset of nodes $R \subseteq V$ called *terminals*. The objective is to find a subset of edges $F \subseteq E$ with minimum cost such that $G = (V, F)$ contains a path between each pair of nodes in R. The graph T induced by F is called a *Steiner tree* and the vertices $V \backslash R$ which are included in T are named *Steiner vertices*.

STP is related to the *shortest path problem* in the sense that, when there are only two terminals, the STP and shortest path problems are identical. Also, if the set of terminals contains all nodes of the input graph (i.e. $R = V$), STP is reduced to the *minimum spanning tree problem*. Although both shortest path and minimum spanning tree problems are solvable in polynomial time, the decision version of STP is known to be NP-complete and, hence, the optimization variant of STP is NP-hard. In fact, the decision version of STP is included in Karp's original 21 NP-complete problems [Kar72].

STP has been extensively studied in the research community and classical approximation techniques have been applied to solve this problem. Simple 2-approximation algorithms can be achieved by applying the primal-dual scheme, LP rounding or by constructing the minimum spanning tree of the subgraph of the metric closure of the graph induced by the terminal vertices. Approximation guarantees better than 2 can be obtained by considering the directed component relaxation of STP (e.g. 1.694-approximation by randomized rounding [Byr+10]). Currently, the best known guarantee is a 1.39-approximation [BC14]. An important fact is that it is NP-hard to approximate STP within 96/95, as shown in [CC02].

In the last years, modern approaches that use graph neural networks [Ahm+21] and deep reinforcement learning [Yan+21; Wan21] were developed, in an attempt to compute good Steiner tree solutions. Although these methods might find solutions with a smaller weight compared to the classical approximation techniques, they typically do not provide any approximation guarantees. Moreover, depending on the graph instance that is solved, the modern approaches are computationally expensive and require training complex models for many hours.

For this Interdisciplinary Project (IDP), we implemented classical and modern

approaches that solve the Steiner Tree problem and evaluated them on graph instances of varying sizes. We implemented three classical 2-approximation algorithms, namely the Repetitive Shortest Path Heuristic, the Primal-Dual Method and the Mehlhorn Algorithm, and a modern approach based on deep reinforcement learning, which extends the idea presented in a recent paper [Yan+21]. The algorithms were evaluated against each other and also against an algorithm implemented in a Python package. The solvers were benchmarked on small graph instances and on a well established dataset for the Steiner tree problem (PACE 2018 dataset [BS19]). The evaluation criteria that we considered include the running times of the algorithms and the ratio between the weight of the computed solutions and the optimal value.

This document is structured as follows: in Chapter 2 we present the theoretical and implementation details of the algorithms that we considered. In Section 2.1 we introduce the classical approaches, while in Section 2.2 we discuss the deep reinforcement learning based algorithm. In Chapter 3 we describe the results that we obtained in the evaluation of the implemented approaches. Finally, in Chapter 4 we summarize the main results of our work.

Chapter 2

Implementation

In this chapter we introduce four algorithms for solving STP that we implemented. Three of them are approximation algorithms (Section 2.1) and one is based on deep reinforcement learning (Section 2.2). The programming language that we chose for the implementation of the algorithms is Python. We used the NetworkX package [HSS22] for the creation and basic manipulation of the graphs. The input for our algorithms is a graph in the PACE 2018-challenge [Del20] format.

2.1 Approximation Algorithms

In this section we present the implemented approximation methods, namely the Repetitive Shortest Path Heuristic (Section 2.1.1), the Primal-Dual Method (2.1.2) and the Mehlhorn Algorithm (Section 2.1.3). All algorithms provide approximation guarantees of $2\left(1-\frac{1}{|R|}\right)$. However, their running times are different, as will be discussed in the following.

2.1.1 Repetitive Shortest Path Heuristic

The Repetitive Shortest Path Heuristic (RSPH) or Minimum Path Heuristic [Ple92] iteratively constructs a solution by adding to the current tree the path corresponding to the uncovered terminal that has the minimum shortest path to a covered terminal. We implemented the algorithm in the method `repetitive_shortest_path(graph, terminals)`. The corresponding pseudocode is presented in Algorithm 1.

In our implementation, we first compute the shortest paths (and their lengths) from each terminal to all nodes in the graph using Dijkstra's algorithm. We ensure that the stored shortest path from a terminal r_1 to another terminal r_2 is identical to the stored shortest path from r_2 to r_1 (since multiple shortest paths can exist between two nodes). Then, we identify the uncovered terminal that is closest to an already covered terminal and add the shortest path connecting it to the current tree. Finally, in a post-processing step, we successively remove from the tree all terminals that have degree 1 (together with their connecting edges), since they are not necessary and increase the cost of the solution.

Algorithm 1 is a $2\left(1-\frac{1}{|R|}\right)$ approximation for STP. A proof of the approximation guarantee can be found in [TM80]. In terms of the running time, the following observations hold: we apply Dijkstra's algorithm to compute the shortest path from each terminal to the other nodes, a step which has complexity $O(|R|(|E|+|V|\log|V|))$. There are $O(|R|)$ iterations and each can be computed in $O(|V|+|E|)$ time (select the closest uncovered terminal to an already covered terminal and add the connecting path to the solution). The post-processing step is done in $O(|V|)$ time. Therefore, the total running time of RPSH is $O(|R|(|E|+|V|\log|V|))$ and is dominated by the computation of the shortest paths.

Algorithm 1: Repetitive Shortest Path Heuristic

```
1 Compute the shortest paths from terminals to all nodes
2 Choose an arbitrary terminal v ∈ R and define a tree T = ({v}, ∅)
3 while F is not a feasible solution do
4 |   Choose the closest terminal u ∈ R to T
5 |   Add the minimum path connecting u to the current tree T
6 Successively remove all non-terminals with degree 1 that are in T
7 Return T
```

2.1.2 Primal-Dual Algorithm

In this section we present our implementation of the Primal-Dual Algorithm [Mad19]. The method is a special case of the primal-dual algorithm that solves the Steiner forest problem [WS11].

Before formalizing the algorithm, let us define the cut set $\mathcal{C}$ as all subsets of vertices such that their induced cuts separate at least one terminal from another:

$$\mathcal{C} = \{S \subseteq V : \emptyset \neq S \cap R \neq R\}.$$

Using this notation, we formulate the following LP:

$$\begin{aligned} \min \quad & \sum_{e \in E} c_e x_e \\ \text{s.t.} \quad & \sum_{e \in \delta(S)} x_e \geq 1 \qquad \forall S \in \mathcal{C} \\ & x_e \geq 0 \qquad\qquad \forall e \in E \end{aligned} \tag{2.1}$$

The primal variable x_e encodes which edges are selected in the Steiner tree. The non-trivial constraints guarantee that there exists a path which connects all terminals for a set S that separates R. The dual of this LP is:

$$\begin{aligned} \max \quad & \sum_{S \in \mathcal{C}} y_S \\ \text{s.t.} \quad & \sum_{S: e \in \delta(S)} y_S \leq c_e \quad \forall e \in E \\ & y_S \geq 0 \qquad\qquad \forall S \in \mathcal{C} \end{aligned} \tag{2.2}$$

The dual variable y_s corresponds to a cut S that separates R. Intuitively, the non-trivial constraints ensure that every edge must pay for the cuts that it crosses by having a greater or equal cost. We observe that if $c_e = 1 \ \ \forall e \in E$, then the dual computes the largest collection of edge-disjoint cuts.

Algorithm 2: Primal-Dual ST

```
1  A = {{x} : x ∈ R}
2  T = (R, ∅)
3  y_S ← 0  ∀S ∈ C
4  while |A| > 1 do
5      while Σ_{S:e∈δ(S)} y_S ≤ c_e is not tight for some e = (u, v) do
6          ∀c ∈ A increase  y_c ← y_c + Δt
7      if u ∉ R ∧ v ∈ R then
8          Add u to the component corresponding to v
9          x_e ← 1
10     else if u ∈ R ∧ v ∉ R then
11         Add v to the component corresponding to u
12         x_e ← 1
13     else if ∃C_i, C_j ∈ A,  i ≠ j :  u ∈ C_i ∧ v ∈ C_j then
14         Add C_i ∪ C_j to A
15         Delete C_i, C_j from A
16         x_e ← 1
17         Find the path p connecting T ∩ C_i and T ∩ C_j with x_e = 1  ∀e ∈ p
18         T = T ∪ p (add vertices and edges in p to T)
19 Return T
```

The general scheme of a primal-dual algorithm is the following. We start with an infeasible solution x to the primal LP 2.1 and a feasible solution y to the dual LP 2.2. As long as the primal solution is not feasible, we increase the dual variables y until at least one dual constraint becomes tight and set its corresponding primal variable to 1. Recall that, unlike in other approximation techniques, in a primal-dual algorithm we do not explicitly solve the primal or the dual linear programs.

The pseudocode of the primal-dual method for STP is shown in Algorithm 2. We start by creating a set $\mathcal{A}$ that contains one component for each terminal. The initial solution T is a forest that includes all terminals and no edges. At the end of the algorithm, T is a Steiner tree. Throughout the algorithm, we grow the components in $\mathcal{A}$ by adding edges and vertices, i.e. some variables x_e become 1. However, not all edges e that fulfill $x_e = 1$ are included to the solution. Instead, we only add edges in T when we merge two active components. The algorithm stops when the number of active components is exactly 1, which indicates that all terminals are connected in the solution.

Our implementation of the Primal-Dual Algorithm consists of the following functions:

- `sort_edges(incident_edges)`: sorts the incident edges to the active components in increasing order w.r.t. their growth value, which is the amount that an edge misses to become tight. In our pseudocode, Δt corresponds to the minimum amount that an edge needs such that its constraint becomes tight.

- `iteration(sorted_edges, incident_edges, steiner_tree, covered_edges count_active_sets, x_graph)`: performs one iteration of the algorithm. Given a sorted list with edges which are not yet covered (i.e. e s.t. $x_e \neq 1$), the algorithm selects the tight edge $e = (u, v)$ and distinguishes 3 cases: u and v are in the same active component (trivial case), either u or v (but not both) is part of an active component, and u and v are in two distinct active components. A list with edges incident to at least one active component is maintained, since the tight edge will be chosen from this set. `x_graph` denotes a graph which contains tight edges. When merging two components, the path that connects them will be found by applying depth-first search in this graph.

- `primal_dual(graph, terminals)`: constructs a Steiner tree iteratively.

Algorithm 2 is a $2\left(1 - \frac{1}{|R|}\right)$ approximation for STP. A proof for the approximation guarantee can be found in [Mad19]. The algorithm has $O(|V|)$ iterations and each of them can be performed in $O(|E| \log |E|)$ (which is dominated by sorting the edges that are incident to the active components). Thus, the total running time of our implemented Primal-Dual Algorithm is $O(|V||E| \log |E|)$.

2.1.3 Mehlhorn Algorithm

The Mehlhorn Algorithm was introduced by Kurt Mehlhorn in [Meh88] and is an adaptation of Algorithm Kou-Markowsky-Berman [KMB81]. Before presenting the Mehlhorn Algorithm, we briefly describe Algorithm Kou-Markowsky-Berman, whose pseudocode is shown in Algorithm 3.

Algorithm 3: Algorithm Kou-Markowsky-Berman

1 Metric closure computation: Construct the complete distance graph $G_1 = (V_1, E_1, d_1)$, where $V_1 = R$ and for each $(v_i, v_j) \in E_1$, $d_1(v_i, v_j)$ is equal to the distance of the shortest path from v_i to v_j in G
2 Compute a minimum spanning tree $G_2 := MST(G_1)$
3 Construct a subgraph $G_3 \subseteq G$ by replacing each edge in G_2 by its corresponding shortest path in G
4 Compute a minimum spanning tree $G_4 := MST(G_3)$
5 Delete unnecessary edges in G_4

The idea of Algorithm Kou-Markowsky-Berman is to compute the minimum spanning tree of the subgraph of the metric closure of G which is induced by R. The metric closure of a graph is defined as the complete graph in which each edge $\{v_i, v_j\}, v_i, v_j \in R$ is weighted by the distance of the shortest path between v_i and v_j in G. When analyzing the pseudocode of Algorithm Kou-Markowsky-Berman, we observe that step **1** is performed in $O(|R|(|V|\log|V| + |E|))$, since it requires solving $|R|$ single source shortest path problems, i.e. $|R|$ runs of the Dijkstra's algorithm. As we will see in the following, the Mehlhorn Algorithm is similar to Algorithm Kou-Markowsky-Berman. In fact, the only difference lies in step **1**, which, in the Mehlhorn Algorithm, is computed in a more efficient way.

We concentrate now on the Mehlhorn Algorithm. For every terminal $r \in R$, let $N(r)$ be the set of nodes in V which are closer to r than to any other vertex in R. Naturally, it holds that $r \in N(r)$ for each terminal $r \in R$. We consider a partition of the set of nodes $V = \bigcup_{r \in R} N(r)$, where $N(r) \cap N(r') = \emptyset$, $r, r' \in R$, $r \neq r'$ and $v \in N(r)$ implies that $d(v, r) \leq d(v, r')$ for all $r' \in R$. Note that, from a computational geometry perspective, the set $N(r)$ represents the Voronoi region of a terminal r.

In the following, we define the set

$$E_1' = \{\{r, r'\} : r, r' \in R, \exists\{u, v\} \in E, u \in N(r), v \in N(r')\}$$

and the distance function

$$d_1' = \min\{d_1(r, u) + c(u, v) + d_1(v, r') : \{u, v\} \in E, u \in N(r), v \in N(r')\}.$$

Consider now the graph $G_1' = (R, E_1', d_1')$. Although d_1' is in general not the restriction of d_1 to the set E_1' (as illustrated in figure 2.1), a minimum spanning tree of G_1' is always a minimum spanning tree of G_1, as the following lemma claims.

Lemma 2.1

1. *There is a minimum spanning tree G_2 of G_1 which is a subgraph of G_1'. Moreover, d_1 and d_1' agree on the edges of this tree.*

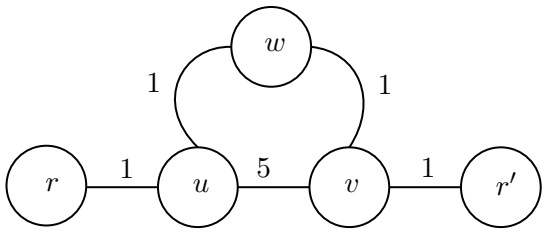

Figure 2.1: Graph with $R = \{r, r'\}$, $d_1(r, r') = 4$ and $d'_1(r, r') = 7$

2. Every minimum spanning tree of G'_1 is a minimum spanning tree of G_1.

The proof of this lemma can be found in [KMB81]. Thus, we can replace step **1** of Algorithm Kou-Markowsky-Berman with the computation of the graph $G' = (R, E'_1, d'_1)$. The final pseudocode of Mehlhorn Algorithm is presented in Algorithm 4. We implemented the algorithm in the method `mehlhorn_algorithm(graph, terminals)`. In the following, we briefly describe our approach.

Algorithm 4: Mehlhorn Algorithm

1 Construct $G'_1 = (R, E'_1, d'_1)$
2 Compute a minimum spanning tree $G_2 := MST(G')$
3 Construct a subgraph $G_3 \subseteq G$ by replacing each edge in G_2 by its correspinding shortest path in G
4 Compute minimum spanning tree $G_4 := MST(G_3)$
5 Delete unnecessary edges in G_4

Clearly, the most computationally expensive part of the algorithm is step **1**. To compute the sets $N(r)$ for all terminals $r \in R$, we first add a vertex s_0 and edges $\{s_0, r\}$ to all terminals $r \in R$ of cost 0 to G. Then, we run Dijkstra's algorithm with source s_0 and obtain the shortest paths from s_0 to all nodes in G. By traversing the shortest path from s_0 to each node $v \in V$ in reverse order, and finding the first terminal in this reversed path, we get the closest terminal $r(v)$ for every node $v \in V$ and also $d_1(v, r(v))$. Eventually, we obtain the sets $N(r)$. This step is done in $O(|V| \log |V| + |E|)$ time. To compute d'_1, we iterate over all edges in E and update $d'_1(r, r')$ any time we find an edge $\{u, v\}$ such that $d_1(r, u) + c(u, v) + d_1(v, r') < d'_1(r, r')$. This is done in $O(|E|)$ time and, thus, step **1** is performed in $O(|V| \log |V| + |E|)$ time (which is dominated by running Dijkstra's algorithm).

For computing the minimum spanning trees G_2 and G_4, we use Prim's algorithm that is implemented in the NetworkX library and has complexity $O(|V| \log |V| + |E|)$. Nevertheless, each of the steps **3** and **5** are done in $O(|E|)$ time. Consequently, Mehlhorn Algorithm has complexity $O(|V| \log |V| + |E|)$, which is a great improvement compared

to Algorithm Kou-Markowsky-Berman, which has complexity $O(|R|(|V|\log|V|+|E|))$. Both algorithms are $2\left(1-\frac{1}{|R|}\right)$ approximations for STP, which is proven in [Meh88].

2.2 Deep Reinforcement Learning

In this section we briefly introduce the main concepts of deep reinforcement learning and present Cherrypick algorithm (Section 2.2.1).

Reinforcement learning is a subfield of machine learning characterized by an agent that uses observed rewards to learn an optimal (or nearly optimal) policy for the environment in which it operates. In Q-learning [RN09], the agent learns an action-utility function, or Q-function, which describes the expected utility of taking a given action in a given state. Unfortunately, in STP, the action space in very large: the number of pairs $(state, next\_state)$ is exponential in the number of nodes, since, for each current state (partial solution) that includes a set of vertices, there exists a large number of possibilities for choosing the node that can be added in order to reach the next state. Thus, creating and updating a Q-table becomes a nearly impossible task, especially for graph instances that contain many nodes.

Deep reinforcement learning provides a solution to the aforementioned problem. It uses a deep neural network (NN) in order to approximate the Q-value function. The current state is given as the input of the NN and the Q-value of all possible actions is generated as the output.

We implemented the Cherrypick algorithm [Yan+21] which uses deep reinforcement learning in order to solve STP. In our implementation, we used the PyTorch [Pas+16] framework.

2.2.1 Cherrypick Algorithm

Cherrypick algorithm is based on the approach presented in a paper by Yan, Du, Zhang and Li [Yan+21]. However, we applied some modifications and improvements to the original idea, as will be discussed in the following. To the best of our knowledge, there exists no public implementation of this approach.

Cherrypick is a greedy algorithm that, in each iteration, adds to the current solution a vertex which is incident to the current solution (i.e. a vertex v such that there exists an edge $\{u, v\}$ with u included in the current solution) and has the maximum Q-value. The algorithm has two main components, namely a **Graph Embedding Model** and a **Deep Q Network**. Given an input graph, the method first constructs a graph embedding after preprocessing the input and obtaining node information. The resulting embedding encodes vertex and path-related information as vertex states. The initial Q-values are computed for all nodes of the graph using these vertex states. Then, the graph embedding and the initial Q-values are fed to a deep Q network that is trained

and updates the Q-values depending on the current partial solution. Before formalizing the algorithm, we introduce some useful notations.

A partial solution is defined as $S = (v_1, v_2, ..., v_{|S|})$, where $v_i \in V$ and $v_1 \in R$. We denote with S^* the incident vertices to the current solution S. For each node v, we define two binary variables: $s_v = 1$ if $v \in S$ and $r_v = 1$ if $v \in R$. In each iteration, we extend the solution S as $S := (S, v^*)$, where $v^* := argmax_{v \in S^*} Q(S, v)$ and $Q(S, v)$ represents the evaluation function value corresponding to v for the current solution S. The cost of a solution is defined as $C(S, G) = \sum_{i=1}^{|S|} \min w(u, v_i^*)$, where $u \in S_i$ and $v_i^* \in S^*$. In the following, we present the two components of Cherrypick.

1. Graph Embedding Model Given an input graph, the algorithm first constructs a graph embedding. In this sense, a similarity matrix $X^{|V| \times k}$ is created, which, for each node in the graph, encodes the inverse of the shortest distance (similarity) to the k closest terminals which are not yet included in the current solution (i.e. entry $X[i, j]$ represents the inverse of the shortest distance from the ith node to the jth terminal). Note that, in the original paper, a distance matrix is created instead of a similarity matrix. We found, through empirical analysis, that a similarity matrix is more appropriate. As suggested in the paper, we set $k = 2$, since it was shown in experiments that this value has the best effect. The matrix is updated any time the model is in the learning phase, which will be discussed later in this section.

Next, an encoder-processor-decoder architecture is constructed. The encoder performs the following step. For each node, the partial solution S (i.e. the value s_v) and the initial vertex weight x_v (i.e. the row associated to v in X) are combined to obtain μ_v as follows:

$$\mu_v = \text{ReLU}(\theta_1[s_v, r_v] + \theta_2 x_v),\ \theta_1 \in \mathbb{R}^{p \times 2}, \theta_2 \in \mathbb{R}^{p \times k}.$$

Thus, we obtain a p-dimensional vertex embedding μ_v. In our implementation, we set $p = 2$. The operation $[\cdot, \cdot]$ is the stacking operation, that is, $[s_v, r_v]$ represents a 2-dimensional vector, since $s_v, r_v \in \{0, 1\}$. ReLU is the rectified linear unit ($\text{ReLU}(a_i) = \max(0, a_i)$ applied to each entry of the vector a). Intuitively, the values μ_v aim to encode global path information and the 'status' of the vertices s_v.

In the processor component, a differentiable message passing framework is implemented. That is, messages (i.e. states μ_u) of the neighbors of a node v are considered in order to update the embedding of v and eventually obtain a new hidden representation $\mu_v^{'} \in \mathbb{R}^p$:

$$\mu_v^{'} = h_\theta \cdot \text{ReLU}([\mu_v, \sum_{u \in N(v)} (\mu_v - \mu_u)]),\ h_\theta \in \mathbb{R}^{p \times 2p}.$$

The neighborhood $N(v)$ is the set containing all adjacent nodes of v. As before, $[\cdot, \cdot]$ is the stacking operation. The multiplication of the weights h_θ and $\text{ReLU}(\cdot)$ reduces the size of $\mu_v^{'}$ to p.

Last but not least, in the decoder component we compute the Q-values for all nodes, given the current solution S:

$$Q(S, v; \theta) = \theta_3^\intercal \cdot \text{ReLU}([\theta_4 \sum_{u \in V} \mu_v^{'}, \theta_5 \mu_v^{'}]), \ \theta_3 \in \mathbb{R}^{2p}, \theta_4, \theta_5 \in \mathbb{R}^{p \times p}.$$

Note that, in order to compute the Q-value of a node v, we use the embeddings of all nodes (their sum) and the embedding of v. The value $Q(S, v; \theta)$ encodes the potential vertex weight of v, given the current solution S and the weights θ. Intuitively, $Q(S, v; \theta)$ represents the 'quality' of v in the current algorithm stage, which is encoded by S and the current weights θ.

The weights $\theta_i, i \in [5]$ and h_θ are initiated using Xavier initialization [Pas+16]. The following methods of our implementation are relevant for the graph embedding model:

- `compute_similarity_matrix(k=2)`: calculates, for all nodes, the similarity $x_{i,r_j} \in [0, 1]$ to the k terminals, where the similarity is defined as the inverse of the shortest distance; we set $x_{r_j,r_j} = 0$.
- `compute_neighborhoods()`: computes, for each node, its adjacent nodes.

2. Deep Q Network The values $Q(S, v; \theta)$ are learned using a Deep Q Network (DQN). In this regard, the DQN updates the weights $\theta_i, i \in [5]$ and h_θ after a predefined number of iterations. Recall that in each iteration, a new node is added to the current solution. In the following, we define six aspects which are specific to any reinforcement learning task:

- **State space**: includes a partial solution S and the adjacent nodes S^*.
- **Transition**: choose a node v from S^* based on the policy and set $s_v = 1$.
- **Action space**: contains the nodes S^* connected to the partial solution S. Cherrypick chooses a node v such that $s_v = 0, v \in S^*$ and adds it to S. Nodes are chosen from S^* in order to maintain the tree structure of the partial solution.
- **Reward**: depends on whether the selected node v is a terminal. $S^{'}$ denotes the new state, i.e. $S^{'} := (S, v)$. The reward is defined as:

$$\text{reward}(S, v) = \begin{cases} C(S^{'}, G) - C(S, G) + |x_v|, v \notin R \\ C(S^{'}, G) - C(S, G) + |x_v| + c, v \in R \end{cases}$$

  where $|x_v|$ represents the sum of the entries in the vector x_v and c is the sum of all edge weights in the input graph. The reward depends on the value of $|x_v|$: if $|x_v|$ is large, then v is 'similar' to the nodes that are already included in the solution (i.e. the combined distance to them is small) and, hence, the reward is high. The purpose of adding the large constant c when v is a terminal is to 'encourage' the network to choose terminals.

- **Policy**: select v such that $\pi(v|S) = argmax_{v \in S^*} Q(S, v)$ (deterministic greedy policy). In order to decrease the chances that the network gets stuck in a local optima, we set the exploration rate ϵ to 0.1, i.e. the algorithm chooses a random node in S^* with probability 0.1 and with probability 0.9 applies the deterministic greedy policy.
- **Termination**: S contains all terminals (the tree structure is implicitly preserved).

Learning As mentioned previously, Cherrypick does not update the weights and the Q-values of the nodes in each iteration. Instead, a learning step is performed after a predefined number of iterations. We use the SGD optimizer [Pas+16] to minimize the square loss:

$$J(\theta) = (y - Q(S_t, v_t; \theta))^2,$$

where

$$y = \gamma \max_{v'} Q'(S_{t+1}, v'; \theta') + \text{reward}(S_t, v_t).$$

In this loss function, γ is the discount rate ($\gamma \in [0, 1]$), which represents the attenuation from the next state information. Thus, $Q'(S_{t+1}, v'; \theta')$ represents the Q-value in the next state S_{t+1} for the vertex v' using the weights θ'. Also, y comes from the *target network*, which we introduce in the following. Figure 2.2 shows the learning process of the function Q, as presented in the original paper [Yan+21]. We observe that the there are two networks that appear in the model, namely the *env-network* and the *target-network*, and a *replay memory* component. These components are specific to any deep reinforcement learning task, and we briefly present them now. An extended discussion can be read in an online post [Tor20].

Recall that in Q-Learning we update a 'guess with a guess', and this can lead to unwanted correlations. The standard Bellman equation [dee18] computes the value $Q(S_t, v)$ using $Q(S_{t+1}, v')$. But the states S_t and S_{t+1} have only one step between them and, clearly, are very similar. Therefore, it is difficult for a neural network to distinguish them and, consequently, the training becomes unstable. We can solve this problem by introducing two deep Q networks, namely the *target-network* and the *env-network*. The Q-values of the *target-network* are used for backpropagation and for training the *env-network*. It is essential that the weights of the *target-network* are not trained, but are periodically synchronized with the parameters of the *env-network*. Thus, using the Q-values of the *target-network* to train the *env-network* improves the training stability.

Let us now concentrate on the *replay memory* component. Recall that in our reinforcement task, we try to approximate a complex, nonlinear function, Q, using a deep neural network. In order to achieve this, we calculate the targets using the Bellman

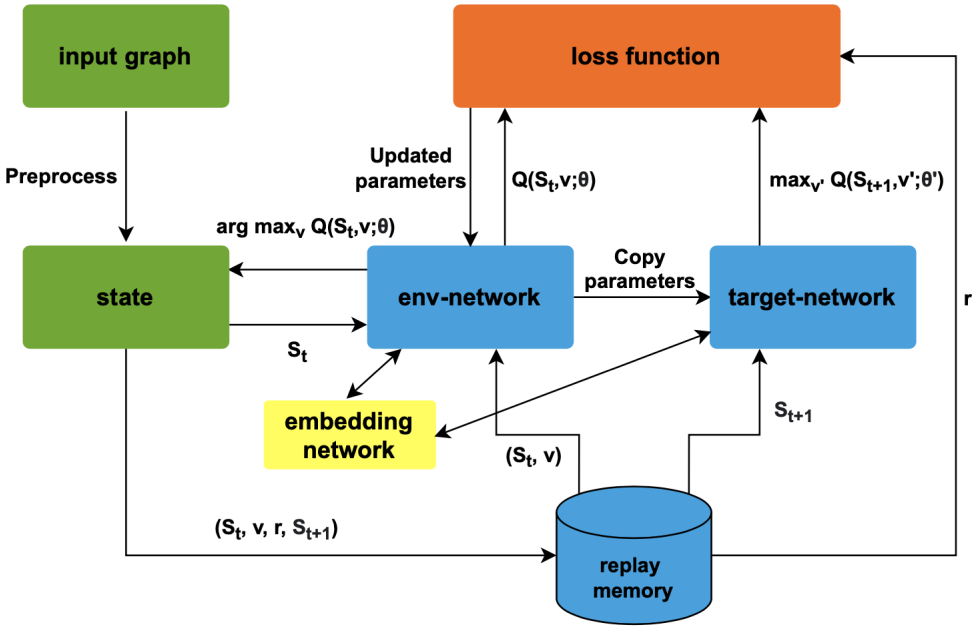

Figure 2.2: Cherrypick learning framework (Inspired from [Yan+21])

equation, and keep in mind that we want to solve a supervised learning problem, whose loss function is optimized using SGD. One important requirement of SGD optimization is that the training data is independent and identically distributed. However, when the agent interacts with the environment, the sequence of experience tuples (i.e. the states defined by the action taken in each iteration) can be highly correlated. Note that the naive Q-learning algorithm that learns from these experiences tuples in sequential order has the risk of oscillating due to the correlations between consequent experiences. To prevent this problem, we use a large buffer of the past experiences and sample training data from it, instead of using the latest experience. This method is called *replay memory*. It contains a set of experience tuples (S_t, v, r, S_{t+1}), which are gradually added to the buffer as the agent interacts with the environment. We implemented a buffer of fixed size using the `deque` data structure from the Python's collections library. A new experience tuple is added to the end of the *replay memory* in any iteration and one tuple is sampled from it in any learning step that updates the *env-network*.

Now that we have explained the main components of the Cherrypick learning framework, we can summarize it in the following. Given an input graph, the algorithm first performs a preprocessing step to obtain the initial Q-values for all nodes and the first state. Then, the state (i.e. solution) is updated in each iteration using the

env-network, by selecting the node v with the highest Q-value and adding it to the current solution. The current state S_t, the node which was added to S_t, the reward that this action brought and the next state S_{t+1} are added to the *replay memory*. Anytime a learning step is performed, the following actions are taken: a tuple (S_t, v, r, S_{t+1}) is sampled from the *replay memory*, the similarity matrix is updated, the Q-values of the *env-network* are recomputed, the loss function is optimized using SGD, obtaining new weights for the *env-network* and the previous weights of the *env-network* are copied to the *target-network*.

Our implementation is based on the following classes and methods:

- `class QNetwork(nn.Module)`: provides all functionalities of a DQN. The main methods are `weight_initialization` and `forward`, the latter updating the Q-values of the nodes.
- `class Agent`: implements the tasks that a learning agent does, such as `transition`, `update_cost`, `compute_reward`, `step`, `act`, `learn` and `hard_update` (copies the weights of the *env-network* to the *target-network*). We also implemented the method `soft_update`, which updates the weights of the *target-network* using a convex combination of the weights of both networks.
- `class ReplayMemory`: provides the methods `add` and `sample`, which are useful in the learning step, when a batch of experiences from the *replay memory* are sampled.
- `dqn(graph, terminals, n_episodes)`: solves the reinforcement learning task. A random terminal is chosen as the initial node and the algorithm terminates after at most `n_episodes`. We set `n_episodes` to be the number of vertices in the original graph, since this represents the maximum number of iterations that are needed for computing a solution. After a feasible solution was found, the unnecessary nodes and edges are removed in a postprocessing step.

Chapter 3

Evaluation

In this chapter we present the results that we obtained during the algorithms evaluation. In Section 3.1 the approximation ratios are discussed, in Section 3.2 the running times are compared and in Section 3.3 the results of an early stopping approach applied on the Primal-Dual Method are presented. In Sections 3.4, 3.5 and 3.6 the results of three evaluation approaches for Cherrypick are discussed.

We compared the approaches against each other and also against an algorithm implemented in the NetworkX package [HSS22], which finds a solution by computing the minimum spanning tree of the subgraph of the metric closure of the initial graph induced by the terminal nodes. Thus, this method corresponds to Algorithm Kou-Markowsky-Berman, which we briefly discussed when we introduced the Mehlhorn algorithm in Section 2.1.3. In the following, we will refer to this algorithm as the 'Metric closure'.

We applied the algorithms on some of the PACE 2018 instances from Track 1 [Del20], which is the 'exact with low number of terminals' track. The optimum value is known for each graph instance from this collection.

3.1 Approximation Ratio

In the following, we present the results that we obtained when we evaluated the approximation ratios achieved by the Mehlhorn, Repetitive Shortest Path Heuristic, Primal-Dual, Metric closure and Cherrypick algorithms.

Let A be an algorithm and $w_I(A)$ the weight of the solution computed by A for a graph instance I. The approximation ratio achieved by A on an instance I is defined as $\frac{w_A(I)}{OPT(I)}$, where OPT is the weight of the optimal solution. Since all classical approaches that we consider are $2\left(1-\frac{1}{|R|}\right)$ approximations, we expect that the approximation ratios are in the interval $[1, 2)$.

Figure 3.1 shows the approximation ratios achieved by the algorithms on graph instances with less than 400 nodes. The instances are sorted by increasing number of vertices. When we created the plot, we rounded the values to 2 decimal points. We observe that all four approximation algorithms achieve very similar (or even identical)

approximation ratios. Also, Cherrypick tends to perform significantly worse compared to the classical approaches and sometimes achieves approximation ratios higher than 2. There are, however, instances for which Cherrypick computes solutions that have a smaller approximation ratio.

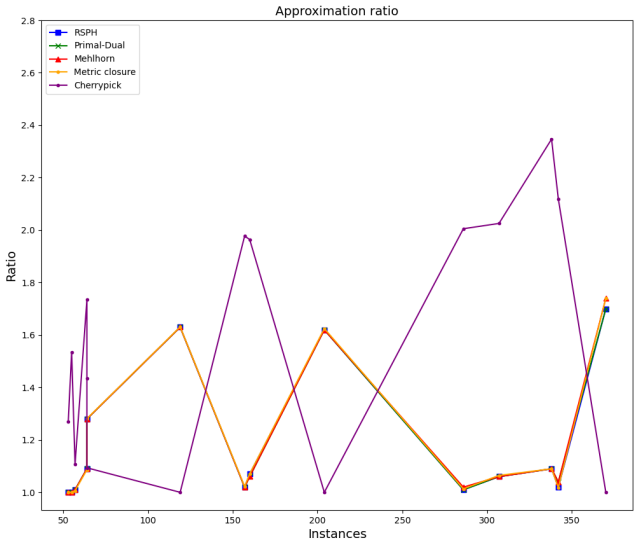

Figure 3.1: PACE Challenge: Approximation ratios of the algorithms. Instances are sorted by increasing number of vertices.

Figure 3.2 illustrates the approximation ratios computed by the approximation algorithms for graph instances with up to 7000 nodes. Although for graphs with a large number of vertices the algorithms do not always achieve the same approximation ratios, we observe that we cannot identify a method that always performs better than the others. Figure 3.3 shows the approximation ratios for all algorithms that we considered, including Cherrypick. We observe that Cherrypick performs considerably badly on graph instances with many nodes.

But what is the performance of Cherrypick on small graphs? To answer this question, we created graph instances with a small number of nodes (up to 50) and with up to 190 edges. We created complete, wheel, ladder and grid graphs, randomly chose the terminal set, calculated the corresponding optimum values and used Cherrypick to solve these instances. The approximation ratios are shown in Figure 3.4. We notice that Cherrypick performs better on small, sparse graphs and that the approximation ratios increase as the number of edges grows.

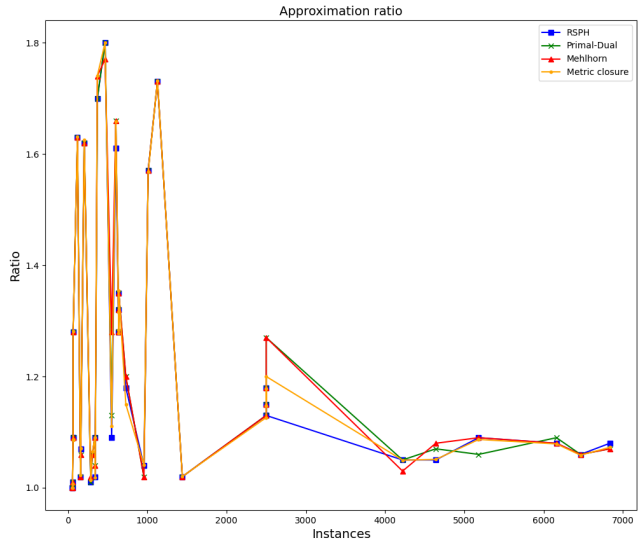

Figure 3.2: PACE Challenge: Approximation ratios of the algorithms. Instances are sorted by increasing number of vertices.

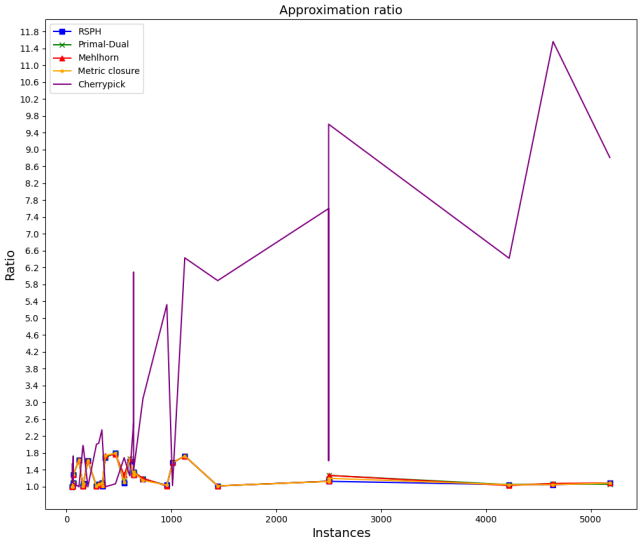

Figure 3.3: PACE Challenge: Approximation ratios of the algorithms. Instances are sorted by increasing number of vertices.

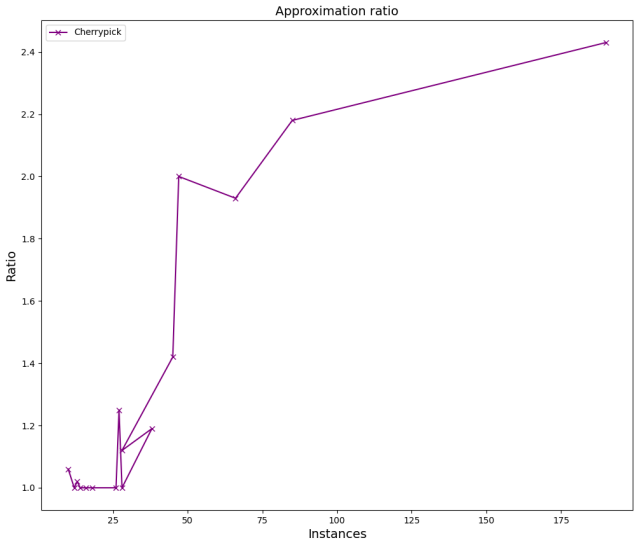

Figure 3.4: Approximation ratios of Cherrypick. Instances are sorted by increasing number of edges.

3.2 Running Time

In this section we present the results that we obtained when we analyzed the running time of the approximation algorithms. To measure the running time, we used the functionalities of the module `time` from Python. We computed the number of seconds that each algorithm needs to compute a Steiner tree for a given graph instance.

Table 3.1 shows the algorithmic complexities of the approximation methods, which we also discussed in Section 2. We observe that the Primal-Dual Algorithm has the highest complexity, while the Mehlhorn Method is, in theory, the fastest.

Algorithm	**Complexity**
Mehlhorn	$O(\|V\| \log \|V\| + \|E\|)$
Repetitive Shortest Path	$O(\|R\|(\|E\| + \|V\| \log \|V\|))$
Primal-Dual	$O(\|V\|\|E\| \log \|E\|)$
Metric closure	$O(\|R\|(\|E\| + \|V\| \log \|V\|))$

Table 3.1: Complexities of the approximation algorithms

Figure 3.5 shows the evolution of the running times of the algorithms on PACE

instances that contain less than 400 nodes. We note that Mehlhorn and RSPH algorithms have the smallest running times. Interestingly, these running times have a small correlation with the number of vertices. This means that applying the Mehlhorn or the RSPH algorithm on an instance with a large number of nodes does not necessarily lead to large running time values. On the contrary, the running times of the Metric closure and Primal-Dual algorithms have a positive correlation with the number of vertices of the instances on which they are applied. Clearly, the highest running time values correspond to the Metric closure method, while the Mehlhorn Algorithm leads to the smallest values of this metric.

Figure 3.6 shows the development of the running times of the algorithms for instances with up to 7000 nodes. We observe a similar trend: the running times of the Metric closure algorithm increase with the number of nodes. We see that, as the number of vertices increases, the Primal-Dual, Mehlhorn and RSPH methods behave similarly, meaning that they achieve comparable running times.

An important observation that we make in the following concerns the Primal-Dual Algorithm. As shown in table 3.1, this method has the highest algorithmic complexity among all approximation algorithms that we considered. However, in our experiments, we saw that the Primal-Dual Algorithm does not lead to the highest running time values. A probable explanation for this fact is related to the sorting algorithm that we used in our implementation. Recall that the complexity of the Primal-Dual Method is dominated by the sorting of the incident edges (to at least one active component), which is an operation that is performed in each iteration. We used Timsort [DH], which is the default sorting algorithm implemented in Python. This algorithm does multiple sorts efficiently since it can take advantage of any ordering already present in a collection. Thus, it is not surprising that Timsort is fast in our setting, since subsets of ordered edges are likely to be present in the set of edges which is sorted in each iteration. This is is due to the fact that the same amount is added to all dual variables in each iteration.

We did not find it appropriate to compute the running times for the Cherrypick algorithm, since it depends on specific hyperparameters, such as the number of iterations after which a Q-values update is performed. Although we did not document this observation, we noticed that the running times of Cherrypick are considerably higher compared to the values corresponding to the classical approaches. This fact was foreseeable, since Cherrypick is an algorithm that relies on machine learning, which is known to be slow in practice.

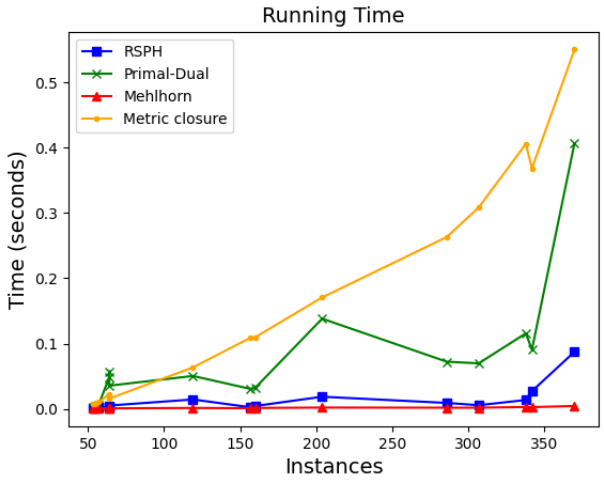

Figure 3.5: PACE Challenge: Running times of the algorithms. Instances are sorted by increasing number of vertices.

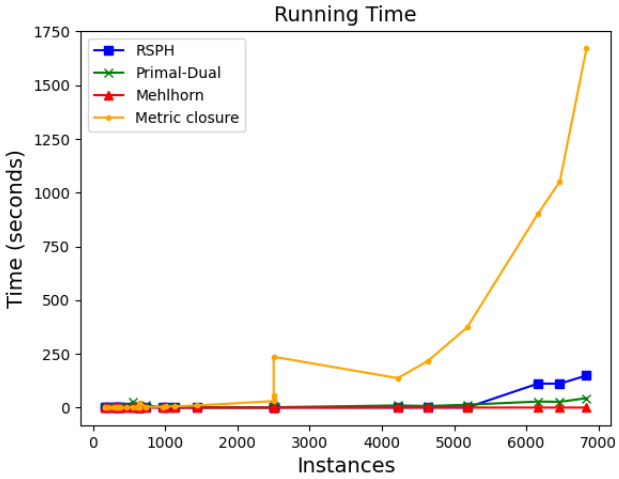

Figure 3.6: PACE Challenge: Running times of the algorithms. Instances are sorted by increasing number of vertices.

3.3 Early Stopping Mechanism (Primal-Dual Algorithm)

In this section we present the results that we obtained when running the Primal-Dual Algorithm with an early stopping mechanism (PD+ES). The idea of such a mechanism is the following. We set a probability of 0.3 and we stop each iteration of the algorithm with this probability if at least two components were already merged (i.e. if at least two terminals were connected in the solution). Since the Primal-Dual Method is not an anytime algorithm, we do not expect that, when an iteration is stopped, the solution computed so far is feasible. Therefore, once the Primal-Dual Algorithm is stopped, we need to adjust the current solution such that, in the end, we obtain a feasible Steiner tree. This is done by merging, iteratively, the largest two components from the current forest. The components are merged by adding to the current solution the shortest path that connects a random terminal from a component with a random terminal from the other component. The reason for always merging the two largest components is that the connecting path is likely to contain nodes which are included in other components. Thus, in one step, we have higher chances to merge more than two components and, therefore, we need few iterations to create a feasible solution. Note that this approach is actually a combination of the 'original' Primal-Dual Algorithm and the Repetitive Shortest Path Heuristic.

For each algorithm run that was 'stopped early', we calculate the cost of the solution before early stopping and the cost needed to ensure the feasibility of the final solution. The final cost of PD+ES is the sum of these two values.

The purpose for implementing an early stopping approach is to observe whether a combination of the Primal-Dual Method and the Repetitive Shortest Path Heuristic leads to better approximation ratios in comparison to the 'original' Primal-Dual Method.

Figure 3.7 shows the approximation ratios achieved by the Primal-Dual Algorithm and PD+ES for graph instances with less than 400 nodes. Interestingly, we observe that there are some graph instances for which PD+ES leads to smaller approximation ratios. Figure 3.8 illustrates the approximation ratios achieved by the same methods for graph instances with less than 7000 nodes. As the plot shows, for instances with many nodes, PD+ES computes solutions that have higher approximation ratios compared to the Primal-Dual Algorithm.

We can conclude that the PD+ES approach is suitable for small graph instances that contain few nodes. However, even for small graphs, there is no guarantee that PD+ES always leads to better solutions in comparison to the Primal-Dual Algorithm. A drawback of PD+ES is that it does not take into consideration the 'progress' made by the Primal-Dual algorithm between the last iteration in which two components were merged and the iteration that is stopped early. In our experiments, we tried different early stopping probabilities, but we found no significant correlation between the number of the iteration that was stopped early and the cost of the final solution.

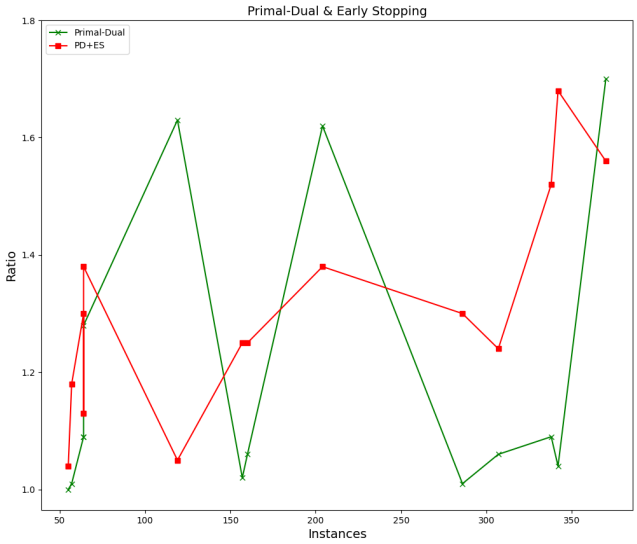

Figure 3.7: PACE Challenge: Approximation ratios of Primal-Dual algorithm and the PD+ES approach. Instances are sorted by increasing number of vertices.

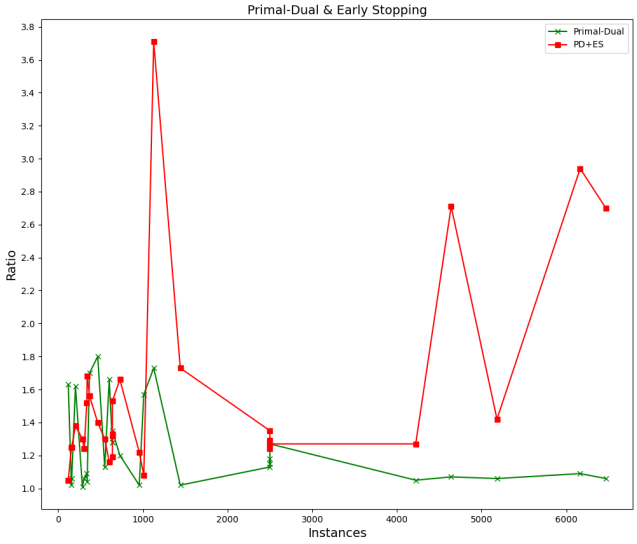

Figure 3.8: PACE Challenge: Approximation ratios of Primal-Dual algorithm and the PD+ES approach. Instances are sorted by increasing number of vertices.

3.4 Transfer Learning (Cherrypick Algorithm)

In the following, we discuss the results that we obtained when applying a transfer learning approach to Cherrypick. The idea of this approach is the following. We use Cherrypick to solve an arbitrary graph instance (given by the graph G). As described in Section 2.2.1, we randomly initialize the weights and compute the Q-values for all nodes in G. Then, we construct a subgraph G' of G and initialize the Q-values of its nodes with the final Q-values computed for G. We compute a solution for G' using these initial Q-values.

The purpose of this approach is to use learned global information (i.e. the Q-values of G) locally. Intuitively, the final Q-values of the original graph reflect the importance of the nodes at the end of the algorithm. That is, the algorithm has learned the 'quality' of a specific node and the 'structure' of the graph. Therefore, when considering a subgraph G' of G, the Q-values of G are likely to be good initial Q-values for G'. Of course, the subgraph G' might have a different set of terminals, and the 'quality' of a node in G' might by different than its quality in the original graph. Thus, we do not simply use the Q-values of G to compute a solution to G'. Instead, the initial Q-values for G' are updated in each learning step.

In our experiments, we created multiple subgraphs for the same graph and applied the transfer learning approach. We compared the solutions obtained for the subgraph using this approach with the solutions obtained when using random initial values. The results are shown in Table 3.2 and indicate that the transfer learning approach tends to produce better results compared to the original Cherrypick implementation (with random initial weights).

Instance	$\|V(G)\|$	$\|V(G')\|$	$\|E(G)\|$	$\|E(G')\|$	**Tr.L.**	**No Tr.L.**
1	17	13	24	16	2.5	2.8
2	17	13	24	13	1	1.6
3	19	14	32	20	1.71	3.23
4	19	14	32	20	3.29	2.52
5	28	28	63	51	2.46	1.82
6	28	20	63	37	2.24	2.71
7	12	8	26	13	1	3
8	12	10	23	15	1	1
9	21	17	27	21	1	1
10	12	10	26	17	1	3

Table 3.2: Results obtained for the transfer learning approach. Tr.L and No Tr.L. represent the approximation ratios obtained with and without the transfer learning approach, respectively.

3.5 Weight Sharing (Cherrypick Algorithm)

We implemented a weight sharing approach for Cherrypick, which we briefly present in this section. The purpose of this approach is to improve the weight of the initial solution computed for an instance in multiple algorithm executions. In the first run, we randomly initialize the weights of the nodes and compute their Q-values as described in Section 2.2.1. In the second run, we use the final Q-values from the previous execution as the initial Q-values and solve the same instance.

In our experiments, we repeated the process four times and chose the best solution. Figure 3.9 depicts the approximation ratios obtained by Cherrypick (blue) and Cherrypick with the weight sharing approach (red), respectively. The latter corresponds to the best solution obtained in the algorithm runs for which the initial Q-values were copied from the first execution. We observe that the results indicate that, for most of the instances, the solution associated with the weight sharing approach is better than the one computed without it. This is not surprising, since the Q-values obtained in the first algorithm execution capture information about the nodes. Therefore, using these values instead of computing the initial Q-values with random initial weights represents a good starting point for the algorithm, which will 'learn' the importance of the nodes in the follow

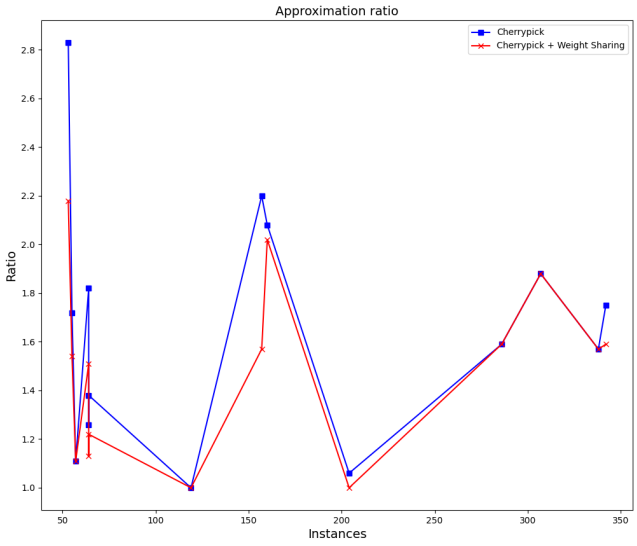

Figure 3.9: PACE Challenge: Approximation ratios of Cherrypick and Cherrypick + Weight Sharing Approach. Instances are sorted by increasing number of vertices.

We think that this approach is reasonable only for small graphs, since running the algorithm multiple times for the same instance is computationally expensive.

3.6 Sequential Learning (Cherrypick Algorithm)

In this section we describe the sequential learning approach that we implemented for Cherrypick. The idea of this approach is to apply the algorithm on the same instance multiple times as follows. In the first run, we randomly initialize the weights of the nodes and compute their Q-values as described in Section 2.2.1. In the subsequent runs, we use the final Q-values from the previous run as the initial Q-values for the nodes. The process is repeated for a predefined number of times. Intuitively, the motivation behind this sequential approach is to use the 'learned information' (i.e. Q-values) from the past for a new algorithm execution in order to enhance the chances that a good solution is computed.

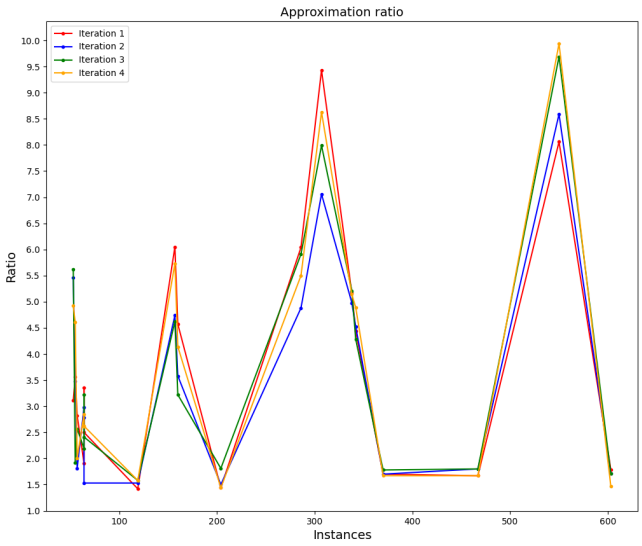

Figure 3.10: PACE Challenge: Approximation ratios of Cherrypick in each iteration using the sequential learning approach. Instances are sorted by increasing number of vertices.

In our experiments, we run the algorithm four times on the same instance. Figure 3.10 shows the evolution of the approximation ratios throughout the iterations. We notice that the approximation ratio does not decrease as the number of iterations grows. Indeed, there are instances for which this approach leads to good results, i.e. the weight of the solution decreases in subsequent iterations. However, in general, we think that this approach is not successful.

Chapter 4

Conclusion

We implemented three approximation algorithms and a deep reinforcement learning based approach that solve the Steiner tree problem. These are the Repetitive Shortest Path Heuristic (Section 2.1.1), the Primal-Dual Method (Section 2.1.2), the Mehlhorn Algorithm (Section 2.1.3) and Cherrypick (Section 2.2.1). We evaluated the algorithms on PACE 2018-challenge instances and on small generated graphs, considering metrics such as the approximation ratio (Section 3.1) and the running time (Section 3.2). Moreover, we implemented an early stopping mechanism for the Primal-Dual Method (Section 3.3) and three evaluation scenarios for Cherrypick, namely Transfer Learning (Section 3.4), Weight Sharing (Section 3.5) and Sequential Learning (Section 3.6).

The evaluation results show that the classical approximation algorithms tend to perform better compared to the modern deep reinforcement learning based method. They achieve smaller approximation ratios and their running times are considerably better. On the other hand, Cherrypick computes good solutions when the weight sharing approach is used, but, in general, the approximation ratios of the returned solutions tend to increase with the number of edges. The results obtained using the transfer learning approach indicate that the final Q-values of a graph represent a good starting point when considering a new instance given by a subgraph of this graph.

List of Figures

Bibliography

[Ahm+21] R. Ahmed, M. A. Turja, F. D. Sahneh, M. Ghosh, K. Hamm, and S. Kobourov. *Computing Steiner Trees using Graph Neural Networks*. 2021. URL: https://arxiv.org/abs/2108.08368.

[BC14] S. Beyer and M. Chimani. "Steiner Tree 1.39-Approximation in Practice". In: *MEMICS*. 2014.

[BS19] É. Bonnet and F. Sikora. "The PACE 2018 Parameterized Algorithms and Computational Experiments Challenge: The Third Iteration". In: *13th International Symposium on Parameterized and Exact Computation (IPEC 2018)*. Ed. by C. Paul and M. Pilipczuk. Vol. 115. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 26:1–26:15. ISBN: 978-3-95977-084-2. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10227.

[Byr+10] J. Byrka, F. Grandoni, T. Rothvo, and L. Sanità. "An Improved LP-Based Approximation for Steiner Tree". In: *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*. STOC '10. Cambridge, Massachusetts, USA: Association for Computing Machinery, 2010, pp. 583–592. ISBN: 9781450300506. URL: https://doi.org/10.1145/1806689.1806769.

[CC02] M. Chlebık and J. Chlebıková. "Approximation Hardness of the Steiner Tree Problem on Graphs". In: *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*. SWAT '02. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 170–179. ISBN: 3540438661.

[DH] A. Dalke and R. Hettinger. *Python Documentation. Sorting HOW TO*. URL: https://docs.python.org/3/howto/sorting.html?highlight=timsort (visited on 05/19/2022).

[dee18] deeplizard. *Reinforcement Learning - Developing Intelligent Agents*. 2018. URL: https://deeplizard.com/learn/video/rP4oEpQbDm4 (visited on 05/19/2022).

[Del20] H. Dell. *PACE-challenge/SteinerTree-PACE-2018-instances*. https://github.com/PACE-challenge/SteinerTree-PACE-2018-instances. 2020.

[HSS22] A. Hagberg, P. Swart, and D. Schult. *NetworkX. Network Analysis in Python*. 2022. URL: https://networkx.org (visited on 05/19/2022).

[Kar72] R. Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations*. Ed. by R. Miller and J. Thatcher. Plenum Press, 1972, pp. 85–103.

[KMB81] L. Kou, G. Markowsky, and L. Berman. "A Fast Algorithm for Steiner Trees". In: *Acta Informatica* 15 (1981), pp. 141–145.

[Mad19] V. Madan. *Primal Dual Method: Approximate Algorithm for Steiner Forest (CS 6550: Randomized Algorithms)*. 2019.

[Meh88] K. Mehlhorn. "A faster approximation algorithm for the Steiner problem in graphs". In: *Information Processing Letters* 27.3 (1988), pp. 125–128.

[Pas+16] A. Paszke, S. Gross, S. Chintala, and G. Chanan. *PyTorch Documentation*. 2016. URL: https://pytorch.org/docs/stable/index.html (visited on 05/19/2022).

[Ple92] J. Plesnik. "Heuristics for the Steiner problem in graphs". In: *Discrete Applied Mathematics* 37/38 (1992), pp. 451–463.

[RN09] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597.

[TM80] H. Takahashi and A. Matsuyama. "An approximate solution for the Steiner problem in graphs". In: *Math. Japon.* 24 (1980), pp. 573–577.

[Tor20] J. Torres. *Deep Q-Network (DQN)-II*. 2020. URL: https://towardsdatascience.com/deep-q-network-dqn-ii-b6bf911b6b2c (visited on 05/19/2022).

[Wan21] S. Wang. "Steiner tree: a deep reinforcement learning approach". MA thesis. University of Delaware, Department of Computer and Information Sciences, 2021.

[WS11] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, 2011.

[Yan+21] Z. Yan, H. Du, J. Zhang, and G. Li. "Cherrypick: Solving the Steiner Tree Problem in Graphs using Deep Reinforcement Learning". In: *2021 IEEE 16th Conference on Industrial Electronics and Applications (ICIEA)*. 2021, pp. 35–40.