A Comparative Study for the Minimum Steiner Tree Problem

Luis Flak

Abstract

We present and analyse three heuristics, a 2-APX algorithm and a flow based integer linear program (ILP) for computing an undirected minimum cost steiner tree (ST). We provide two optimisations for the ILP and a parallelisation approach for the 2-APX algorithm. The parallel adaptation of the 2-APX algorithm is then supported by two parallel random access machine analyses. We evaluated and compared the algorithms in extensive experiments for performance and the cost of the produced STs. Furthermore, structural influences of graphs on the respective algorithms are determined.

1 Introduction

Solving the undirected minimum cost steiner tree (UMCST) problem is essential for various networks in different fields. It is critical for applications in telecommunications routing [Kun et al., 2006, Hsieh and Liao, 2007], it finds application in biomolecular structure modeling [Mondaini, 2008], and it is used for chip design [Tang et al., 2020]. The UMCST problem involves creating a spanning tree of minimum cost that includes a certain set of vertices. The optimisation problem, i.e. finding such a minimum cost steiner tree (ST) is NP-Hard. On the other hand, deciding if there exists a ST within a given cost is NP-Complete [Karp, 2009].

In this work, we implement a flow-based integer linear program (ILP) formulation to solve the UMCST problem exactly, develop a 2-APX algorithm that leverages the metric closure of the given graph, and propose three heuristics for computing a minimum cost ST. These heuristics include using a normal and a pruned minimum spanning tree (MST), and the Takahashi-Matsuyama Heuristic [Takahashi and Matsuyama [1980] as cited by Winter [1987]]. We then propose a parallelisation approach for the 2-APX algorithm including a parallel random access memory (PRAM) analysis. Additionally, we also provide optimisations for the ILP formulation. We then conduct comprehensive experiments to evaluate and compare the performance of these methods and improvements.

Section 2 introduces the problem formulation and outlines the implemented approaches. Section 3 discusses the algorithm engineering techniques and improvements applied to these methods. In Section 4, we present the experimental setup and results. Finally, in Section 5 our findings are discussed.

2 Preliminaries

Let $G = (V, E, c)$ be an undirected graph with a cost function $c : E \to \mathbb{R}_{\geq 0}$ that assigns each edge a non-negative value. We denote $R \subseteq V$ as the set of terminals. A ST $T = (V_T, E_T, c)$ is defined as a connected and acyclic subgraph of G, such that $R \subseteq V_T$. Then the UMCST problem asks for a ST with minimum cost, i.e. that $c(E_T) := \sum_{e \in E_T} c(e)$ is minimised [Winter, 1987]. In practice, we can reduce the computation to just E_T as V_T can be determined as $V_T = \{w \ : \ \{w, w'\} \in E_T\}$.

Let $N(v) := \{w \ : \ \{v, w\} \in E\}$ denote the neighborhood of a vertex $v \in V$ in G. A path $P \in E^k$ of length $k \in \mathbb{N}$ is a sequence of edges in G. We denote $P_{v,w}$ as a path where the end points are v and w, and $P^{opt}_{v,w}$ as the shortest (i.e. minimum cost) path between v and w.

For the theoretical analysis, we assume that all algorithms are implemented using a Fibonacci heap [Fredman and Tarjan, 1987, Cormen et al., 2009] as priority queue.

2.1 Pruned Minimum Spanning Tree Heuristic

By definition every MST is also a ST as it spans all vertices V and $R \subseteq V$. Since it spans all vertices, a MST possibly includes a much larger set of vertices and edges than needed to only span R. Thus, the resulting cost $c(E_T)$ can be much larger than the cost $c(E_T^{OPT})$ of an optimum (minimum cost) ST.
A straightforward approach to reduce the cost of a normal MST is by pruning leafs and their incident edge [Takahashi and Matsuyama [1980] as cited by Winter [1987]]. Since the costs are non-negative, removing edges can only reduce the total cost of the tree. Since all terminals must be included in the final tree, removing a leaf l is only possible if l is not a terminal, i.e. $l \notin R$.
The creation of a MST can be done in $\mathcal{O}(|E| + |V| \log |V|)$ using Pirm's algorithm Prim's algorithm [Prim, 1957]. Then for pruning the MST, the next leaf to be removed can be determined in constant time by utilising a first-in, first-out (FIFO) queue data structure of edges. We first enqueue the only incident edge of all non-terminal leafs in the MST (i.e. vertices $v \in V$ with $|N_{MST}(v)| = 1$ where N_{MST} denotes the neighborhood in the MST). Since the approach only removes leafs, the resulting structure remains a tree. Also since the method does not remove connections to terminal leafs, all terminals remain connected. The superfluous leafs and their edges can thus be removed one after the other, until no valid leafs to be removed remain. When a leaf l and its incident edge $\{l, w\}$ is removed, the other end w of the edge might become a leaf itself. Therefore, w is tested for being a non-terminal leaf after the removal of $\{l, w\}$. If this is the case, w's only incidet edge is enqueued to be removed later. This processes at most $|E|$ edges and stops after no non-terminal leaf exists after $\mathcal{O}(|E|)$ time.
Thus a pruned MST representing a ST can also be constructed in $\mathcal{O}(|E|+|V| \log |V|)$ total time. The approach is also depicted in Algorithm 1. This approach is said to have a $|V| - |R| + 1$ error ratio [Takahashi and Matsuyama [1980] as cited by Winter [1987]].

Algorithm 1: PrunedMST(G, R)

```
1  (V_MST, E_MST) = ComputeMST(G)
2  Q ← CreateQueue()
3  for {v, w} ∈ E do
4      if |N_MST(v)| = 1 ∧ v ∉ R ∨ |N_MST(w)| =
         1 ∧ w ∉ R then
5          Q.Enqueue({v, w})
   // Prune leafs
6  while ¬Q.isEmpty() do
7      {v, w} ← Q.Dequeue()
8      E_MST ← E_MST \ {{v, w}}
9      u ← v if |N_MST(v)| < |N_MST(w)| else w
10     if u ∉ R ∧ |N_MST(u)| = 1 then
11         e ← {u, u'} ∈ E_MST // e is unique
12         Q.Enqueue(e)
13 return (V_MST, E_MST)
```

2.2 Takahashi-Matsuyama Heuristic

A more sophisticated heuristic is also described by [Takahashi and Matsuyama [1980] as cited by Winter [1987]]. This heuristic grows a ST by starting with an arbitrary vertex r as tree $T = (V_T, E_T, c)$ with $V_T = \{r\}$ and $E_T = \emptyset$. Then iteratively the shortest path P between any vertex v in the current tree V_T and any terminal t not yet included in V_T is computed and added to T (i.e. all edges on P are added to E_T and all vertices incident to an edge on P are added to V_T).
Since these paths P have only one vertex in V_T as endpoint, the edges on P are disjoint to any edges already in the tree, i.e. $\{e \ : \ e \in P\} \cap E_T = \emptyset$. Therefore, the created structure remains a tree throughout computation and no pruning afterwards is needed.
The authors are also said to have shown that the worst-case error ratio of the resulting tree T is bounded by $2 - 2/|R|$ [Takahashi and Matsuyama [1980] as cited by Winter [1987]].
The algorithm computes in each step a multi-source shortest path computation for each terminal $t \in R \setminus \{r\}$. A straightforward approach for this would be to run Dijkstra's algorithm [Dijkstra, 1959] for each vertex currently in V_T. In the worst-case, there

are all $|V|$ vertices in V_T. Since we add at least one vertex in each iteration, this results in $\sum_{i=1}^{|V|} i$ computations of Dijkstra's algorithm. Using Gauss formula, this results in $\mathcal{O}(|V|^2)$ shortest path computations. Dijkstra's algorithm has a running time of $\mathcal{O}(|E| + |V| \log |V|)$. Thus the running time of the overall algorithm is in $\mathcal{O}(|V|^2 \cdot (|E| + |V| \log |V|))$ time. When using a multi-source shortest path algorithm, this running time can be improved. When using Dijkstra's algorithm, a dummy vertex can be added with a connection to all desired sources. Then Dijkstra's algorithm is executed as normal starting from the dummy vertex. When ignoring the dummy vertex in the found path, this approach will find a shortest path starting from either vertex the dummy has been connected to. This is exactly what the Takahashi-Matsuyama heuristic needs for each iteration: A shortest path starting from any vertex in the set V_T. Since this approach only adds constant overhead, the improved algorithm only has to run Dijkstra's algorithm once per iteration. There are $|R|$ iterations (one for each terminal added to the tree) which results in a total reduced running time of $\mathcal{O}(|R| \cdot (|E| + |V| \log |V|))$. The approach is also depicted in Algorithm 2.

Algorithm 2: TakahashiMatsuyama(G, R)

```
V_T ← {r} ⊆ R // Start with any terminal
E_T ← ∅
R' ← R \ V_T
while R' ≠ ∅ do
    P ← ShortestPath(V_T, R', G)
    V_T ← V_T ∪ {w : {w, w'} ∈ P}
    E_T ← E_T ∪ {e : e ∈ P}
    R' ← R' \ (V_T ∩ R)
return (V_T, E_T, c)
```

2.3 2-APX Algorithm

Another method for computing an UMCST is the 2-APX algorithm [Kou et al., 1981].

Let $E_{MC} := \{\{v, w\} : v, w \in V\}$. Then the metric closure is defined as a derived complete graph $MC := (V, E_{MC}, c_{MC})$ with a cost function

$$c_{MC}(\{v, w\}) := \begin{cases} c(\{e : e \in P^{opt}_{v,w}\}), \text{ if } \exists P_{v,w} \\ \infty, \text{ otherwise.} \end{cases}$$

In other words, the cost of the new edges in MC are defined as the cost of a shortest path (i.e. the accumulated cost of all edges on the path) in G between the same vertices and ∞ if no such path exists in G. For this, the cost of all shortest paths for all pairs of vertices $(v, w) \in V^2$ with $v \neq w$ have to be computed. To do this, Dijkstra's algorithm can be run from each vertex $v \in V$ once to get the shortest paths to all $w \in V \setminus \{v\}$. Thus, the computation of the metric closure of a graph G can be done in $\mathcal{O}(|V| \cdot (|E| + |V| \log |V|))$ total time using Dijkstra's algorithm. However, for the 2-APX algorithm only the metric closure induced by R is needed. Since only paths between distinct $v, w \in R$ need to be created. Thus, the running time for computing the induced metric closure is $\mathcal{O}(|R| \cdot (|E| + |V| \log |V|))$.

Next, a MST is computed on the metric closure in $\mathcal{O}(|R|^2 + |R| \log |R|) = \mathcal{O}(|R|^2)$ time using Prim's algorithm. Then, the edges of this metric closure MST need to be expanded to a tree in the origianl graph. To do this, for each edge $\{v, w\}$ in this MST of the metric closure, the shortest path $P^{opt}_{v,w}$ between v and w is computed in the original graph G. This path $P^{opt}_{v,w}$ is then added to the final ST of the original graph G. This results in the same running time as for creating the metric cosure as the MST of the metric closure includes $|R| - 1$ edges and for each of these edges $\{v, w\}$ the shortest path between v and w is determined. Thus, the expansion of the MST edges in the metric closure have a running time of $\mathcal{O}(|R| \cdot (|E| + |V| \log |V|))$ when using Dijkstra's algorithm.

However, the computed set of edges E_T may contain cycles as the shortest paths could have common edges. Thus, the computed structure (E_V, E_T, c) must be pruned to form a valid tree. This can be done by computing another MST on just (E_V, E_T, c) as this creates a tree by removing the most expensive edges in the structure.

Since the running time of creating the metric closure dominates all steps, the total running time becomes

$\mathcal{O}(|R| \cdot (|E| + |V| \log |V|))$. The approach is also depicted in Algorithm 3.

By induction it can be proven that there must exist a loop $L \in V^k$ of length k in T (with $|E| \geq 1$), such that each edge in E_T appears twice in L, i.e. $c(L) = 2E_T^{OPT}$. Also each leaf in E_T appears exactly once in the loop L and between two subsequent leafs in L there must be a simple path (i.e. no loops). Note that in a minimum ST, every leaf must be a terminal as the leaf could be removed otherwise without invalidating the ST. Then L consists of l simple paths between two leafs in T. Remove the simple path with highest cost from the loop L to create a path P. Note that at least one of the l paths must have a cost of at least $c(L)/l$ by pigeonhole principle. Then P covers every edge $e \in E_T$ at least once and has a total cost $c(P)$ of at most $(1-(1/l)) \cdot c(L) = 2 \cdot (1-(1/l)) \cdot E_T^{OPT}$. Since P spans all $t \in R$, $c(P) \geq c(E_{MC}) \geq c(E_{MST})$ where E_{MST} are the edges of a MST on G. Since the final ST edges are possibly pruned with the same cost of the edges in the metric closure, we get that $c(E_T) \leq c(E_{MC})$. In total, $c(P) \geq c(E_{MC}) \geq c(E_{MST}) \geq c(E_T)$

Thus $c(E_T) \leq 2 \cdot (1 - (1/l)) \cdot E_T^{OPT} \overset{(l \geq 2)}{<} 2 \cdot E_T^{OPT}$ and therefore, Algorithm 3 is in 2-APX [Kou et al., 1981].

Algorithm 3: TwoAPX(G, R)

```
1  V_T ← ∅
2  E_T ← ∅
3  (V_MC, E_MC, c_MC) ←
     InducedMetricClosure(G, R)
4  (V_MC^MST, E_MC^MST, c_MC) ←
     ComputeMST(V_MC, E_MC, c_MC)
5  for {v, w} ∈ E_MC^MST do
       // v, w are in R
6      P ← ShortestPath({v}, {w}, G)
7      V_T ← V_T ∪ {w : {w, w'} ∈ P}
8      E_T ← E_T ∪ {e : e ∈ P}
9  Prune(V_T, E_T, c)
10 return (V_T, E_T, c)
```

2.4 Integer Linear Program

Many ILP formulations for the UMCST exist. A classical formulation uses flow to construct a tree containing all terminals [Goemans and Myung, 1993]. Let the vector

$$x_e := \begin{cases} 1, & e \in E_T, \\ 0, & \text{otherwise.} \end{cases}$$

represent, for each edge $e \in E$ if it is included in the final ST or not.

Since flow normally operates on directed graphs $D = (V, A)$, we need to double each undirected edge $v, w \in E$ to two directed arcs $(v, w), (w, v) \in A$ on which we then apply flow. We define the function æ : $A \to E$ that maps any arc to its respective edge, i.e. $(v, w), (w, v) \in A$ are mapped to $\{v, w\} \in E$. We denote $N^+(v) := \{w : \exists (v, w) \in A\}$ the neighborhood reachable by outgoing arcs and $N^-(v) := \{w : \exists (w, v) \in A\}$ the neighborhood reachable by incoming arcs of a vertex $v \in V$ respectively.

To force the selected edges to connect the terminals, we select any terminal $r \in R$ as root of the final tree $T = (V_T, E_T, c)$. Then r sends one unit of flow to each terminal $t \in R \setminus \{r\}$. It is possible to use a single flow construction (single-commodity) for all terminals together. However, this typically results in weaker bounds. Thus a common improvement is to construct flow for each terminal individually (multi-commodity) [Vanderbeck and Wolsey, 2009]. We denote the flow construction for the terminal $t \in R \setminus \{r\}$ as a vector $f^t \in [0, 1]^{|A|}$. That means, f_a^t refers to the flow on arc $a \in A$ for the flow construction of non-root terminal t. Let the respective demand $b^t \in \{0, 1\}^{|V|}$ of a vertex $v \in V$ for each flow construction be defined as

$$b_v^t := \begin{cases} 1, & v = r, \\ -1, & v = t, \\ 0, & \text{otherwise.} \end{cases}$$

Then f^t is used to enforce for each $v \in V$ the constraint $\sum_{(v,w) \in A} f_{(v,w)}^t - \sum_{(w,v) \in A} f_{(w,v)}^t = b_v^t$ with $w \in V$. In other words, the outgoing flow must equal

the incoming flow. This is known as the "Flow Conservation Constraint" and must apply for every vertex, for a flow to be valid. This constraint must be satisfied for each vertex in every flow construction f^t. To let the flow only use arcs $a \in A$ representing edges $e \in E_T \subseteq E$ that are selected for the final tree, the constraint $f_a^t \leq x_e$ is applied for all $a \in A, e \in E$ with $æ(a) = e$. This can be rearranged to $f_a^t - x_e \leq 0$. We call this constraint "Edge Selection Constraint".
Finally, the goal is to minimise the total cost of the selected edges $\sum_{e \in E} c(e) \cdot x_e$. This lets us define the flow based, multi-commodity ILP as follows [Goemans and Myung, 1993, Vanderbeck and Wolsey, 2009].

$$\begin{aligned}
\text{minimise } & \sum_{e \in E} c(e) \cdot x_e \\
\text{subject to } & f_a^t \leq x_{æ(a)}, \\
& \qquad \forall t \in R \setminus \{r\}, a \in A \\
& \qquad \text{(Edge Selection)} \\
& \sum_{(v,w) \in A} f_{(v,w)}^t - \sum_{(w,v) \in A} f_{(w,v)}^t = b_v^t, \\
& \qquad \forall t \in R \setminus \{r\}, v \in V \\
& \qquad \text{(Flow Conservation)} \\
& x_e \in \{0,1\}^{|E|}, \quad \forall e \in E \\
& f_a^t \in [0,1]^{|A|}, \quad \forall t \in R \setminus \{r\}, a \in A
\end{aligned}$$

3 Algorithm & Implementation

To evaluate the usability and performance of the algorithms, we have to come up with suitable implementations. In this section, we first describe modifications made to the algorithms to improve their practical running times. After this, we provide implementation details for improved performance and memory efficiency.

3.1 Algorithm Engineering

We first propose a parallelisation approach for the 2-APX algorithm and the creation of the metric closure. Then we discuss improvements on the ILP formulation.

3.1.1 Parallelisation

To improve the efficiency of Algorithm 3, we can utilise multiple threads to distribute computation to parallel executions. Each thread is identified by a unique value t. For parallelisation, we then propose the following adaptations.

Parallel 2-APX The edges of a given metric closure can be expanded to paths in parallel. For this, we supply each thread with a separate array E_T^t to store its computed shortest paths for each of its considered edges in the metric closure. These paths are not merged directly as in the sequential algorithm but after all shortest paths are computed in parallel. This post processing then merges the sets E_T^t sequentially for all threads without duplicates. This results in a similar set E_T as for the sequential version before pruning. The parallel approach is also presented in Algorithm 4.
Using the parallel random access machine (PRAM) model, we can achieve a more accurate performance analysis for the parallel approach [Fortune and Wyllie, 1978]. For the sequential computation of the MST edges E_{MC}^{MST} on the metric closure MC, the work and depth are $W_{MC} = D_{MC} = \mathcal{O}(|R|(|E| + |V| \log |V|) + |R|^2) = \mathcal{O}(|R|(|E| + |V| \log |V|))$ as for the sequential running time. When collecting the edges in parallel, the work $W_{\mathcal{E}} = \mathcal{O}(|R|(|E| + |V| \log |V|))$ is the same as for the sequential execution. The depth becomes $D_{\mathcal{E}} = \mathcal{O}(|E| + |V| \log |V|)$ for a single execution of Dijkstra's algorithm. Next, the collected edges in $\mathcal{E}$ are merged sequentially. This results in the work and depth being the sequential running time of considering all collected edges. In the worst case, each shortest path between terminals consists of $|V|$ edges which results in a total of $|V| \cdot |R|$ edges. Thus $W_{merge} = D_{merge} = \mathcal{O}(|V| \cdot |R|)$. Afterwards the pruning using Prim's algorithm also has $W_{prune} = D_{prune} = \mathcal{O}(|E| + |V| \log |V|))$. Thus the total work becomes $W_\infty = \mathcal{O}(|R|(|E| + |V| \log |V|) + |V| \cdot |R|)$. Additionally, the total depth is $D_\infty = \mathcal{O}(|R|(|E| + |V| \log |V|))$. By Brent's principle [Brent, 1974, Gustafson, 2011] with p threads, the parallel

running time is bounded by

$$D_p \leq \mathcal{O}(|R|(|E| + |V|\log|V|)) + \frac{\mathcal{O}(|R|(|E| + |V|\log|V|) + |V| \cdot |R|)}{p}.$$

In theory this is a worse running time as for the sequential algorithm, even with a high number of threads. However, simple parallelisation can still perform better than their sequential counterparts in practice.

Algorithm 4: ParallelTwoAPX(G, R)

```
(V_MC, E_MC, c_MC) ← InducedMetricClosure(G, R)
(V_MC^MST, E_MC^MST, c_MC) ← ComputeMST(V_MC, E_MC, c_MC)
// Create a set for each thread
E ← (E_T^1, ..., E_T^k) ∈ P(E)^k
for {v, w} ∈ E_MC^MST in parallel do
    // v, w are in R
    t ← id of current thread
    P ← ShortestPath({v}, {w}, G)
    E_T^t ← E_T^t ∪ {e : e ∈ P}
E_T ← MergeSets(E)
V_T ← {w : {w, w'} ∈ E_T}
Prune(V_T, E_T, c)
return (V_T, E_T, c)
```

Parallel Metric Closure A similar approach can also be applied for computing the induced metric closure itself. Here the shortest paths again are computed in parallel and later merged. The work for the parallel computation is $W_{\mathcal{E}^{MC}} = \mathcal{O}(|R| \cdot (|E| + |V|\log|V|))$ while the depth becomes $D_{\mathcal{E}^{MC}} = \mathcal{O}(|E| + |V|\log|V|)$ as it is represented by a single shortest path computation. Then the sequential merging has a work and depth of $W_{\mathcal{E}^{MC}} = D_{\mathcal{E}^{MC}} = \mathcal{O}(|R|^2)$ as the metric closure is a complete graph with $|R|$ vertices. Thus, the total work becomes $W_\infty^{MC} = \mathcal{O}(|R| \cdot (|E| + |V|\log|V|))$ and the total depth becomes $D_\infty^{MC} = \mathcal{O}(|E| + |V|\log|V| + |R|^2)$. Then, using Brent's principle [Brent, 1974, Gustafson, 2011] with p threads, the parallel running time is bounded by

$$D_p \leq \mathcal{O}(|E| + |V|\log|V| + |R|^2) + \frac{\mathcal{O}(|R| \cdot (|E| + |V|\log|V|))}{p}.$$

In theory, this provides a significant improvement to the sequential computation, when using enough processors. This parallel approach is also presented in Algorithm 5. Additionally, the parallel metric closure can be combined with the overall parallel 2-APX method. However, since the edge collection in the 2-APX algorithm remains sequential, the new parallel bound for the metric closure does not change the previous result for the overall 2-APX algorithm.

Algorithm 5: ParallelInducedMetricClosure(G, R)

```
c_MC : R → R_≥0
// Create a set for each thread
E ← (E_MC^1, ..., E_MC^k) ∈ P(E)^k
// Let each vertex v ∈ V have a unique id id(v)
id : V → N
for v ∈ V in parallel do
    t ← id of current thread
    P ← ShortestPath({v}, V \ {v}, G)
    for w ∈ V \ {v} with id(v) < id(w) do
        if P ≠ () then
            E_MC^t ← E_MC^t ∪ {{v, w}}
            c_MC({v, w}) ← c({e : e ∈ P})
E_MC ← MergeSets(E)
return (V, E_MC, c_MC)
```

3.1.2 Improving the ILP

Additionally, we can optimise the ILP computation. A first improvement is the already discussed use of the multi-commodity ILP instead of the single-commodity version [Vanderbeck and Wolsey, 2009]. Additionally, in order to enable the ILP solver to detect suboptimal solutions earlier, an additional constraint can be added to the overall constraint matrix.

This constraint is $\sum_{e \in E} c(e)x_e \leq u$ where $u \in \mathbb{R}_{\geq 0}$ is the result of a faster algorithm, providing an upper bound for the overall cost of the ST. For this, the 2-APX algorithm could be utilised.

Another improvement can be achieved by adding a precomputation to reduce the number of edges and thus reducing the overall size of the ILP to solve. This reduced ILP then can be solved faster due to less constraints. For this, Dijkstra's algorithm can be run for each edge $\{v, w\} \in E$ to determine if there is a cheaper path $P_{v,w}$ with length $|P_{v,w}| > 1$. In this case, the edge $\{v, w\}$ would always be replaced by the cheaper path $P_{v,w}$ in any minimum ST (since all costs are non-negative). Thus the edge can be removed from E entirely without invalidating the result of the ILP. The approach takes in total $\mathcal{O}(|E| \cdot (|E| + |V| \log |V|)) = \mathcal{O}(|E|^2 + |E| \cdot |V| \log |V|)$ time and is also described in Algorithm 6. Additionally, the algorithm can also be parallelised by executing the shortest paths computations in parallel and flagging the respective edges. Afterwards the flagged edges can be removed sequentially.

Algorithm 6: ReduceGraph(G)

```
for {v, w} ∈ E do
    P ← ShortestPath({v}, {w}, G)
    if c({e : e ∈ P}) < c({v, w}) then
        E ← E \ {{v, w}}
```

3.2 Implementation Details

We implemented the discussed algorithms in the C programming language [International Organization for Standardization, 2024]. For parallelisation purposes, we utilised OpenMP [OpenMP Architecture Review Board, 2024]. Our ILP implementations were done using the GNU Linear Programming Kit (GLPK) [Makhorin, 2024].

For computation, utilising an adjacency matrix would be infeasible for large graph instances due to the quadratic memory consumption. Thus, we recommend modeling the graph using an adjacency array representation as this improves cache locality and thus overall performance compared to an adjacency list and additionally only has a linear memory consumption. In this representation, all edges are stored together in a dedicated array. Each vertex $v \in V$ receives an array of size $|N(v)|$ where the indices of its incident edges in the edges array are stored.

Additionally, the use of an adjacency array graph representation can directly be integrated with GLPK. For this, doubled arcs are inserted into the same adjacency array structure where the orientation of the arc is determined by the order of its incident vertices. Then each pair of arcs representing an undirected edge are inserted such that the two arcs are consecutive. This approach also enables to directly encode the undirected edges, too, by only considering the first arc of each pair. This way, only the set of arcs needs to be stored and not both a directed and undirected graph, which significantly reduces memory consumption.

Furthermore, due to the poor practical performance of Fibonacci heaps in most szenarios [Cherkassky et al., 1996], we recommend to use a binary min-heap instead for all algorithms. This changes the theoretical running time of Dijkstra's algorithm and Prim's algorithm to $\mathcal{O}((|E| + |V|) \log |V|)$. Thus the theoretical running time of Algorithm 1 becomes $\mathcal{O}((|E| + |V|) \log |V|)$, the running time of Algorithm 2 and Algorithm 3 are now $\mathcal{O}(|R| \cdot ((|E| + |V|) \log |V|))$ and the running time of Algorithm 6 reduces to $\mathcal{O}(|E| \cdot ((|E| + |V|) \log |V|)$. Additionally the PRAM bounds also change accoridingly to $D_p \leq \mathcal{O}(|R|((|E| + |V|) \log |V|)) + (\mathcal{O}(|R|((|E| + |V|) \log |V|) + |V| \cdot |R|)/p)$ for the overall 2-APX algorithm and $D_p \leq \mathcal{O}((|E| + |V|) \log |V| + |R|^2) + (\mathcal{O}(|R| \cdot ((|E| + |V|) \log |V|))/p)$ for the metric closure creation.

Another improvement in implementation concerns the multi-source Dijkstra algorithm. As described in Section 2.2, running Dijkstra from multiple sources at once improves the overall running time of the approach. However, in practice, the insertion of a dedicated dummy vertex is unnecessary as directly inserting the desired sources into the priority queue before the start of Dijkstra's algorithm achieves the same result. This is because inserting a dummy vertex would result in the same state, i.e. that all its neigh-

bors (the set of start vertices) would be inserted into the priority queue at the beginning of the algorithm. The only difference is that inserting the set of start vertices directly avoids the overhead of inserting a dedicated vertex and additionally, the resulting predecessor structure is reduced by one.

4 Experimental Evaluation

In order to evaluate the performance and applicability of the different algorithms, our parallelisation and optimisation approaches, we conducted several benchmarks. We evaluated the algorithms on a multitude of graphs. The results of these evaluations are presented in Section 4.2.

4.1 Data and Hardware

All benchmarks were executed on an GNU/Linux [Free Software Foundation, 2024, Linux Kernel Organization, 2024] Laptop, running Fedora [Fedora Project, 2024] on an AMD Ryzen 7 4800HS processor with 16 GiB RAM.
We ran the algorithms on three tracks with 200 graph instances each, which were provided by PACE Challenge Committee [2018b,a]. The maximum number of edges $|E|$ in all 600 instances (so in all three tracks combined) is $366,093$.
The individual executions on the graph instances had a timeout of five minutes. If an algorithm was estimated to be able to execute multiple times in under 50 seconds, the respective algorithm was executed multiple times (at most $1,000$ times) on the same graph instance and the results were averaged across the multiple runs on the same instance.
Track 1 provides instances with few terminals, Track 2 includes instances with low treewidth, and Track 3 contains instances without such constraints. Across all 600 instances, we evaluated a normal MST using Prim's algorithm (MST), the pruned MST (PMST) heuristic, the Takahashi-Matsuyama (TM) heuristic, and the 2-APX algorithm in both its sequential (2-APX) and parallel (2-APX-P) versions.The 2-APX-P approach utilises both parallel metric closure and parallel edge expansion. In addition, we evaluated two versions of the multi-commodity ILP on the 200 Track 1 instances. First, we implemented the basic multi-commodity ILP without any optimisations. Next, we evaluated an optimised version (OILP) that integrates both an upper bound constraint, derived from the sequential 2-APX algorithm, and the preprocessing step described in Algorithm 6 for reducing the graph. The preprocessing step for OILP has been implemented to compute the shortest paths on the individual edges in parallel and remove the detected edges afterwards sequentially.

4.2 Experimental Results

For all executed experiments, no out-of-memory exceptions occurred. The only reason for an algorithm to not provide a valid result is that the five minute time constraint for a single execution on a graph instance was not sufficient. For the experimental evaluation, we computed the density of the graph instances using the formula $|E|/\binom{|V|}{2}$ [Diestel, 2025].
Table 4.2 shows the number of edges for each smallest tested instance that the used algorithms could not perform on in time. We call this smallest respective instance the problematic instance. The table shows that TM and 2-APX had a similar problematic instance. This underlines the similar theoretical running time of the approaches. Additionally, the problematic instances for both algorithms were two of the largest graphs in the combined data set which supports their usability. On the other hand, ILP and OILP struggled with the same, much smaller instance, which suggests that the improvement of OILP is not significant enough to enable execution on more instances than the basic ILP. Furthermore, the MST and PMST algorithms were executed on all instances in under five minutes which aligns with their efficient running times as they provide the best theoretical bounds among all approaches. The fact that 2-APX-P also finished on all instances in time suggests a significant improvement of our parallelisation approach despite the worse theoretical bound provided by the PRAM analysis of the overall approach.

Table 4.2 shows a similar picture for the average and maximum running time of all algorithms on the respective tested and finished instances. The table

Algorithm	Smallest $\|E\|$ Timeout
MST	-
PMST	-
TM	300,036
2-APX	318,136
2-APX-P	-
ILP	149
OILP	149

Table 1: Smallest number of edges of all 600 instances for which the respective algorithms had a timeout during computation. ILP are only OILP considered on Track 1 instances.

Algorithm	Time (ms)	
	AVG	MAX
MST	15.93	290.84
PMST	16.52	300.28
TM	4,568.41	222,133.01
2-APX	4,659.29	280,852.81
2-APX-P	1,241.66	88,852.65
ILP	79,391.85	290,512.31
OILP	54,822.18	239,433.09

Table 2: Average and maximum running time over all 600 instances for the different algorithms. ILP are only OILP considered on Track 1 instances.

suggests that MST and PMST have a similar running time in both the average and maximum cases. The same applies for TM and 2-APX, underlining the results of Table 4.2. However, they also show that compared to 2-APX, the parallel implementation 2-APX-P has a significant decrease of 68.36% in running time in the respective worst-case and even a 73.35% decrease on average. This contradicts the overall PRAM analysis for 2-APX. However, the metric closure creation does provide an improved bound on its own, in theory. Additionally, using multiple threads can still achieve a speedup, even though the theoretical bounds do not fully support it, because larger workloads can be distributed among multiple threads. The table also shows the difference in magnitude of ILP and OILP compared to the other algorithms, supporting their exponential running time. Additionally, OILP is an improvement in both average and maximum running time, providing a speedup of 17.58% for the respective slowest run and even a 30.95% improvement on average, compared to the basic ILP. Overall, except for 2-APX-P, the practical performance mirrors the theoretical expectations.

Next, Table 4.2 shows the average and maximum cost percentage difference to the cost of an optimum ST. These values only consider instances from Track 1 and Track 2 as Track 3 does not provide exact minimum cost solutions. For ILP and OILP only Track 1 instances are considered, again. The table suggests a significant improvement of PMST over MST in both measurements with a percentage difference of $15,831.5\%$ on the respective worst-case instances, highlighting the efficiency of the pruning approach on a MST. Since the parallel implementation might select a different subset of edges from its collected edges, the resulting cost can vary slightly compared to the sequential 2-APX. However, the table suggests that the two approaches still result in similar STs in general. Furthermore, on average the two approximations of 2-APX and 2-APX-P are shown, where the results suggest a significantly better approximation than the theoretical bound imply. The 2-APX and 2-APX-P algorithms also provide a much tighter approximation than MST and PMST which aligns with the theoretical results as MST has no approximation guarantee and PMST is said to only provide a $|V|-|R|+1$ approximation ratio [Takahashi and Matsuyama [1980] as cited by Winter [1987]]. Additionally, the best approximation in both measurements is provided by TM which also aligns with its supposed $2 - 2/|R|$ approximation guarantee [Takahashi and Matsuyama [1980] as cited by Winter [1987]] which is better than two approximations. This means, the practical results depicted in Table 4.2 perfectly align with the theoretical results.

Figure 1 shows the absolute cost difference to an optimum solution for each instance in Track 1 and depicts in general a similar picture as Table 4.2. However, it also shows that on some sparse graphs (where $|V| \approx |E|$), the 2-APX and 2-APX-P algorithms perform worse than all other approaches, including the basic MST. Prim's algorithm does not work across

Algorithm	Cost Diff. to $c(E_T^{OPT})$ (%)	
	AVG	MAX
MST	552.57	16,626.67
PMST	80.54	795.17
TM	3.59	36.48
2-APX	25.66	91.64
2-APX-P	25.59	91.64
ILP	0.00	0.00
OILP	0.00	0.00

Table 3: Average and maximum difference in percent of the computed cost to the minimum cost of an optimum ST over all 600 instances for the different algorithms. ILP are only OILP considered on Track 1 instances.

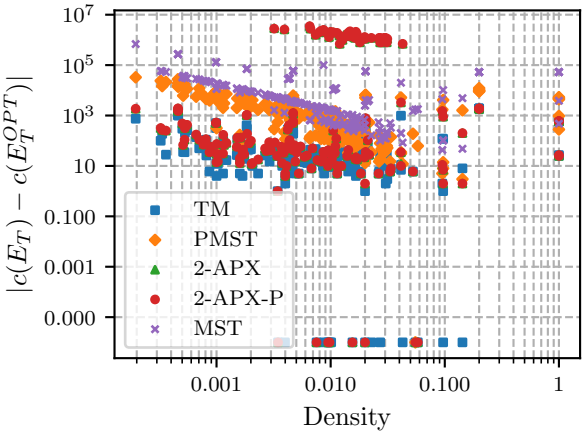

Figure 1: Absolute cost difference compared to cost of a minimum ST for all non-exact algorithms across density on Track 1 instances. Both axes in log scale.

disconnected components. Since it starts at a terminal and the graphs are sparse, the resulting MST might result in a better selection of edges than the two approximations. The 2-APX and 2-APX-P approaches utilise the metric closure which could potentially add higher cost edges to the ST from the computed shortest paths. In general, the two approximation algorithms provide a slightly worse approximation than TM but better than MST and PMST. On the other hand, PMST performs much better than MST on nearly all instances as it only removes edges from the same structure and thus cannot perform worse. The other tracks demonstrate a similar picture.

In Figure 2 the performance in running time on individual instances is depicted. It shows the exponential running times of the exact linear programming computations and demonstrates the immense time difference in magnitude of these approaches compared to the other algorithms. It also shows the advantage in running time of the OILP compared to the basic ILP across nearly all tested instances. An exception is the instance with the smallest number of edges. For this small instance, reducing the graph using Algorithm 6 in parallel is adding a greater time overhead and ends up requiring more running time than solving the small ILP directly. In general however, the time difference between ILP and OILP is similar to the described performances in Table 4.2.

Additionally, MST is the fastest approach across all instances, where PMST has a similar running time. The two approximation approaches perform slightly worse than the MST based ones. Across most instances with less edges (up to $|E| \approx 10^4$), 2-APX-P ends up performing worse than the sequential 2-APX. These results suggest that the overhead of creating multiple threads, the overall parallelisation overhead, and the separate merging step for the collected edges increase the overall running time and result in a slightly worse performance on most smaller instances. This aligns with our theoretical results. However, for larger, denser graphs, the benefits of parallel computation can surpass the extra overhead by distributing the substantial workload across multiple threads, despite the theoretical PRAM bound.

Additionally, TM performs for instances with less edges significantly better than 2-APX, despite them having the same theoretical running time. For the instances with the fewest edges, it even performs similarly to MST. On sparse graphs, shortest paths are computed very efficiently due to the lack of alternative connections for a path. Thus the allocation and computation of a quadratic induced subgraph (i.e. the metric closure) for 2-APX can be a significant ad-

ditional overhead compared to only computing shortest paths on a sparse graph. The strong increase of time for the 2-APX and 2-APX-P algorithms on four instances at $|E| \approx 10^4$ is also likely due to significant larger terminal sets as this increases the running time for creating the metric closure drastically.

Figure 3 suggests that 2-APX-P performs in general slightly worse for smaller maximum degrees than its sequential counterpart. However, for larger maximum degrees, the parallel algorithm overtakes 2-APX in performance. This aligns with our findings in Figure 2. Such instances with a large maximum degree can be dense which enables the parallel threads to distribute the graph and its work evenly. This increases performance compared to one thread covering the entire well connected graph despite a similar (or even worse) theoretical bound. Also for sparse graphs with a large maximum degree, the heavy workload around the well connected vertices can also be given to multiple threads, leading to a faster processing of such hubs.

Other than that, Figure 3 demonstrates a similar picture as Figure 1 suggesting that the maximum degree does not impact the other algorithms significantly.

Next, Figure 4 demonstrates the running time across

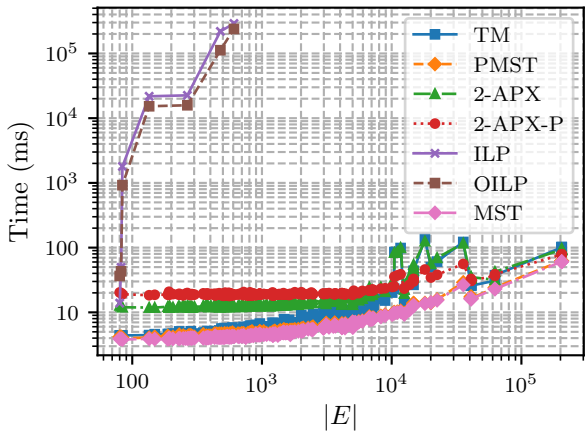

Figure 2: Running time for all algorithms across the number of edges on Track 1 instances. Both axes in log scale.

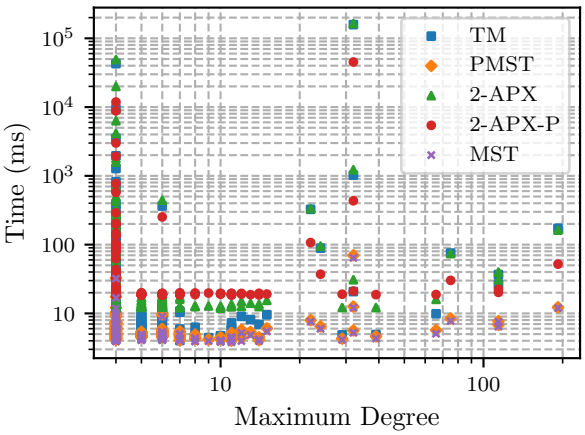

Figure 3: Running time for all non-exact algorithms across maximum degrees on Track 2 instances. Both axes in log scale.

density without additional conditions. In contrast to Figure 2, it shows that 2-APX-P a performs better than 2-APX for almost all instances. Since the instances on Track 3 can have larger sets of terminals or more complex graphs, this aligns with the previous assumption that our parallelisation approach 2-APX-P performs better for heavier work as this can be distributed among threads to result in a better overall performance in practice. Additionally, 2-APX-P even outperforms TM on some instances. This supports the earlier findings for the performance of TM. The algorithm is superior when confronted with simpler and sparse graphs but becomes similar to 2-APX as the instances become more complex. Overall, a trend of better performance for tree-like structures can be examined for all algorithms. This is due to Prim's algorithm and Dijkstra's algorithm performing better on such structures due to the lack of alternative routes the algorithms have to consider.

Figure 6 demonstrates that our parallelisation approach performs better for larger sets of terminals as for those instances the metric closure becomes larger and thus the overall computation more expensive which then can be distributed better between multiple threads. This is also underlines the im-

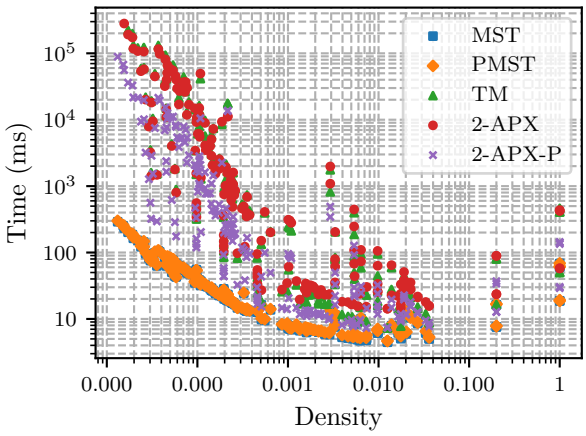

Figure 4: Running time for all algorithms across density on Track 3 instances. Both axes in log scale.

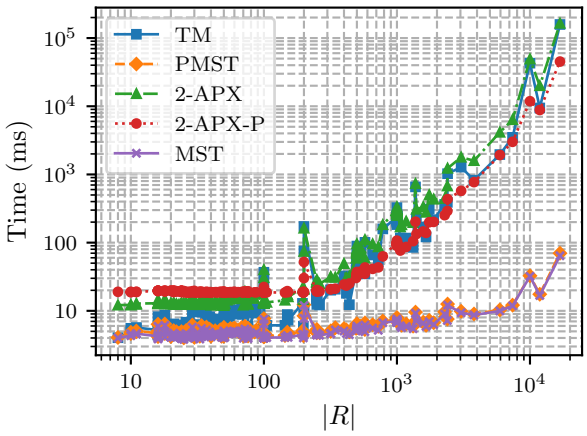

Figure 5: Running time for all non-exact algorithms across the number of terminals on Track 2 instances. Both axes in log scale.

proved PRAM bound for the parallel metric closure algorithm. The overall growth in running time for larger terminal sets reflects the respective theoretical bounds.

Finally, Figure 5 shows that for many instances with smaller terminal sets, the parallelisation is worse than the 2-APX algorithm. However, there are heavy hitters (which are likely complexer graphs with more terminals), instances where the sequential running time is significantly worse than for the parallel algorithm. This aligns with our previous findings as the parallelisation performs generally better on larger, more complex instances with many terminals.

5 Discussion and Conclusion

The experimental evaluation has highlighted the immense running time for the ILP approaches while showing significant improvement in running time for OILP compared to the basic ILP. Additionally, we identified that the parallelisation for our 2-APX only has an advantage on complex instances, i.e. larger graphs with a large set of terminals or graphs with a large maximum degree. This suggests that the theoretical PRAM analysis did not fully capture the overall performance benefits of the parallel approach. On other instances, the sequential approach outperformed our parallel implementation. However, it is noteworthy that some heavy hitter instances exist for which the parallelisation provides a significant improvement. On average 2-APX-P also performed better than 2-APX, despite its worse theoretical PRAM bound. This makes the parallelisation suitable for more complex problems. Moreover, the measured running times of all non-parallel algorithms showed the expected results and a similar trend as the theoretical bounds implied. Our experiments also highlighted the performance gain of TM on sparse graphs. Considering the memory consumption, none of the approaches tested had a problem with the amount of memory provided.

Additionally, we verified for most instances even better approximation results than the respective theoretical bounds imply. The cost differences between the considered algorithms for the resulting STs also mirrored the theoretical results. Some exceptions were noted, where the two approximations performed worse than all other approaches, likely due to structural benefits for MST computations.

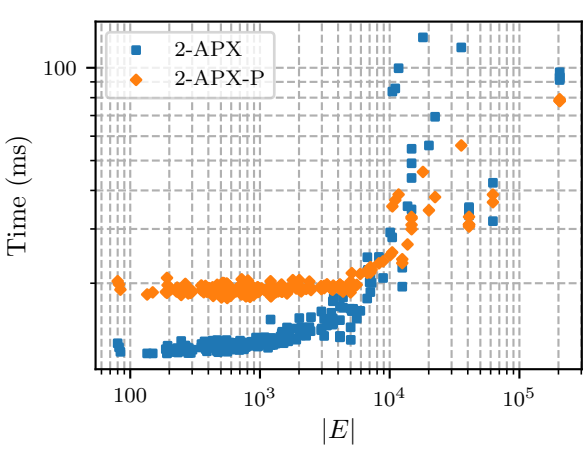

Figure 6: Running time for 2-APX and 2-APX-P across the number of edges on Track 1 instances. Both axes in log scale.

We identified TM as providing the overall best approximations, followed by the two approximation algorithms. The PMST approach created a significantly better approximation than the basic MST, while not surpassing the 2-APX on most instances. Additionally, PMST had a similar running time to MST throughout all instances, making it a superior algorithm to the basic MST in every aspect. Since TM provides the best approximation and a balanced running time, it is recommended for most practical applications.

References

R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974. doi: 10.1145/321812.321815.

B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73: 129–174, 1996. doi: 10.1007/BF02592101.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009. ISBN 978-0-262-03384-8.

R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer Berlin, Heidelberg, 6 edition, 2025. ISBN 978-3-662-70107-2. doi: 10.1007/978-3-662-70107-2.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. doi: 10.1007/BF01386390.

Fedora Project. Fedora linux, 2024. URL `https://fedoraproject.org/`. Accessed: 2025-03-22.

S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 114–118, 1978. doi: 10.1145/800133.804339.

M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987. doi: 10.1145/28869.28874.

Free Software Foundation. Gnu operating system, 2024. URL `https://www.gnu.org/`. Accessed: 2025-03-22.

M. X. Goemans and Y.-S. Myung. A catalog of steiner tree formulations. *Networks*, 23:19–28, 1993. doi: 10.1002/net.3230230104.

J. L. Gustafson. *Brent's Theorem*, pages 182–185. Springer, Boston, MA, 2011. ISBN 978-0-387-09765-7. doi: 10.1007/978-0-387-09766-4_80.

C.-Y. Hsieh and W. Liao. All-optical multicast routing in sparse splitting wdm networks. *IEEE Journal on Selected Areas in Communications*, 25:51–62, 2007. doi: 10.1109/JSAC-OCN.2007.021305.

International Organization for Standardization. Iso/iec 9899:2024 information technology - programming languages - c, 2024. URL `https://www.iso.org/standard/82075.html`. Accessed: 2025-03-22.

R. M. Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*, pages 219–241. Springer, Berlin, Heidelberg, 2009. doi: 10.1007/978-3-540-68279-0_8.

L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15: 141–145, 1981. doi: 10.1007/BF00288961.

Z. Kun, Q. Yong, and Z. Hong. Dynamic multicast routing algorithm for delay and delay variation-bounded steiner tree problem. *Knowledge-Based Systems*, 19:554–564, 2006. doi: 10.1016/j.knosys.2006.04.012.

Linux Kernel Organization. The Linux Kernel, 2024. URL `https://www.kernel.org/`. Accessed: 2025-03-22.

A. Makhorin. Gnu linear programming kit (glpk), 2024. URL `https://www.gnu.org/software/glpk/`. Accessed: 2025-03-22.

R. P. Mondaini. *The Steiner Tree Problem and Its Application to the Modelling of Biomolecular Structures*, volume 102, pages 199–219. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-76784-8_6.

OpenMP Architecture Review Board. Openmp: Open multi-processing api, 2024. URL `https://www.openmp.org/`. Accessed: 2025-03-22.

PACE Challenge Committee. Pace challenge 2018: The steiner tree problem, 2018a. URL `https://pacechallenge.org/2018/`. Accessed: 2025-03-22.

PACE Challenge Committee. Pace 2018 steiner tree instances, 2018b. URL `https://github.com/PACE-challenge/SteinerTree-PACE-2018-instances`. Accessed: 2025-03-22.

R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36:1389–1401, 1957. doi: 10.1002/j.1538-7305.1957.tb01515.x.

H. Takahashi and A. Matsuyama. An approximate solution for the steiner problem in graphs. *Math. Japonica*, 24:573–577, 1980.

H. Tang, G. Liu, X. Chen, and N. Xiong. A survey on steiner tree construction and global routing for vlsi design. *IEEE Access*, 8:68593–68622, 2020. doi: 10.1109/ACCESS.2020.2986138.

F. Vanderbeck and L. A. Wolsey. Using compact extended formulations. In *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*, chapter 13.5.2 Variable splitting I: multi-commodity extended formulations, pages 473–475. Springer Berlin, Heidelberg, 2009. doi: 10.1007/978-3-540-68279-0.

P. Winter. Steiner problem in networks: A survey. *Networks*, 17:129–167, 1987. doi: 10.1002/net.3230170203.