Emit As You Go: Enumerating Edges of a Spanning Tree

Katrin Casel
Humboldt Universität zu Berlin
Berlin, Germany
Katrin.Casel@hu-berlin.de

Stefan Neubert
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
Stefan.Neubert@hpi.de

ABSTRACT

Classically, planning tasks are studied as a two-step process: plan creation and plan execution. In situations where plan creation is slow (for example, due to expensive information access or complex constraints), a natural speed-up tactic is interleaving planning and execution. We implement such an approach with an enumeration algorithm that, after little preprocessing time, outputs parts of a plan one by one with little delay in-between consecutive outputs. As concrete planning task, we consider efficient connectivity in a network formalized as the minimum spanning tree problem in all four standard variants: (un)weighted (un)directed graphs. Solution parts to be emitted one by one for this concrete task are the individual edges that form the final tree.

We show with algorithmic upper bounds and matching unconditional adversary lower bounds that efficient enumeration is possible for three of four problem variants; specifically for undirected unweighted graphs (delay in the order of the average degree), as well as graphs with either weights (delay in the order of the maximum degree and the average runtime per emitted edge of a total-time algorithm) or directions (delay in the order of the maximum degree). For graphs with both weighted and directed edges, we show that no meaningful enumeration is possible.

Finally, with experiments on random undirected unweighted graphs, we show that the theoretical advantage of little preprocessing and delay carries over to practice.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**.

KEYWORDS

solution part enumeration; preprocessing vs. delay; spanning tree

ACM Reference Format:
Katrin Casel and Stefan Neubert. 2025. Emit As You Go: Enumerating Edges of a Spanning Tree. In *Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Detroit, Michigan, USA, May 19 – 23, 2025*, IFAAMAS, 9 pages.

1 INTRODUCTION

In many complex planning settings, such as path finding for robots in a storage facility, the combined time of planning and execution determines the overall efficiency. A natural way of optimizing this is to start execution before fully finishing planning [33, 45]. This approach generalizes to all kinds of multi-step processes, in which the output of an early step is needed to start working on a later one. In fact, with large input instances it can even be beneficial to employ an algorithm with worse total-time, if that algorithm produces the solution in the form of many *solution parts* and (while still running) emits them for following steps to process [28].

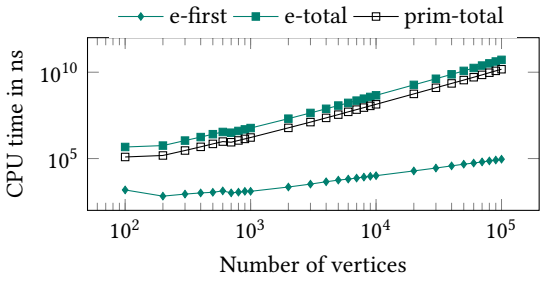

Figure 1: Time to first output and total time of MST enumeration (e) compared to the runtime of Prim's MST algorithm.

But how much speedup can actually be gained by this approach? We tackle this question for one of the most fundamental problems on networks: cost-efficient connectivity. Be it networks for communication, electricity, transport or water supply – both for planning problems and for the analysis of existing infrastructure it is a central task to connect every node to all other nodes by a Minimum Spanning Tree (MST). Besides being useful as output on their own, MSTs are also used as input for many complex algorithmic tasks, e. g. for Coverage Path Planning [40], or in data analytics for graph-based clustering [19, 44], and for image segmentation [15]. In this work, we consider MSTs in general and discuss in Section 5 next steps to bringing our results closer to the motivating applications.

Naturally, there is an extensive history of research on the complexity of computing a *(minimum) spanning tree* for (un)directed and (un)weighted graphs – that is if by *complexity* one refers to the *total-time complexity* of the problem: How many computation steps does it take in total to produce a complete solution as output? With the above motivation of *starting next steps early*, recently several core algorithmic problems have been analyzed using terminology from *enumeration complexity* [39]: How much *preprocessing* time does it take at most to produce a first part of the solution, and what is the worst case *delay* in-between two consecutive solution parts that are emitted? (Note that in contrast to classical enumeration this question does not ask to enumerate all solutions to an input instance, but to enumerate all parts of a single solution.) Ideally, an algorithm does not need to spend any time on preprocessing but is able to immediately emit solution parts with little delay. On the other end of the spectrum are problems for which the time to compute even the first few parts of a solution is in the same order as the time required to solve the problem completely. The goal

Proc. of the 24th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2025), Y. Vorobeychik, S. Das, A. Nowé (eds.), May 19 – 23, 2025, Detroit, Michigan, USA. © 2025 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org).

therefore is to understand possible tradeoffs between preprocessing time and delay.

Among the problems studied with this perspective are incremental sorting [31, 35], shuffling [6], computation of all pairs shortest distances in a graph [7] and finding a topological ordering in a directed acyclic graph for solving scheduling problems [33]. Interestingly, one of the applications of incremental sorting is speeding up total-time MST algorithms; improved also as *Filter Kruskal* in [34]. While these algorithms *employ* the incremental idea for sub problems, we go one step further and discuss MST *itself* in terms of enumerating solution parts.

Our Contribution. We present a family of efficient algorithms that enumerate edges of a spanning tree for a given input graph. For MSTs of weighted undirected graphs, we experimentally confirm in Section 4 that our enumeration approach leads to a smaller time to a first emitted solution part and small delay in practice, with only a constant overhead in the total time (see Figure 1).

We complement the algorithmic results with lower bounds to almost completely characterize the complexity of enumerating the edges of a spanning tree for a connected graph with n vertices and m edges. Table 1 summarizes our results for all typical edge types:

- **Undirected unweighted** edges allow for a delay in the order of the average degree $\overline{\Delta}$; we derive a matching lower bound from the best possible total-time algorithm (Section 3.1).
- For **undirected weighted** edges we show a lower bound in the order of the maximum degree Δ on either preprocessing time or delay. We also present algorithms that, for small Δ or with $O(n)$ preprocessing time, homogeneously spread out the computation time $\mathcal{T}$ of a total-time algorithm for minimum spanning trees and emit the $n-1$ solution parts with delay in $O(\frac{\mathcal{T}}{n})$ (Section 3.2).
- Given a **directed unweighted** graph along with the root for a spanning tree, computing a shortest path tree with delay in the order of the maximum out-degree Δ^+ [7] solves the problem with optimal delay, as we show with a matching lower bound (Section 3.3)
- Without given root or when the graph has **directed weighted** edges, no algorithm can emit any part of a solution after less than $\Omega(m)$ time. For unweighted edges this already matches the total time needed to solve the instance. But also the small remaining gap to existing total time algorithms for the weighted case (see below for related work) rules out any meaningful enumeration (Section 3.4).

We build upon the model and techniques introduced in [7]. We formalize these in Section 2 where we also describe the general form of the algorithms we develop. For some theorems (marked with *) we only present proof ideas due to space constraints. The omitted proofs along with all experiment code can be found in the full version of the paper [8].

Our algorithms usually start with a targeted search for a set of easy-to-find solution parts. As this is then extended to a full solution by tailored versions of existing algorithms for finding spanning trees, it makes sense to summarize the previous work in this field:

A spanning tree for an unweighted graph can be computed in $O(m+n)$, e. g. with a depth first search from a given root [11]; we show in Theorem 3.9 how to cover the case without given root.

There are numerous approaches to solve the *minimum weight* spanning tree problem for *undirected* edge-weighted graphs (and the equivalent problem of maximizing the tree's total edge weight). While we refer to [14, 22, 32] for a comprehensive account of the many (re)discoveries, we want to mention the approaches for the comparison-based computation model coined by Borůvka ([4]; $O(m\log(n))$), Jarník, Prim, and Dijkstra ([12, 25, 37]; $O(m + n\log(n))$ with Fibonacci heaps [16]), Kruskal ([27]; $O(m\log(n))$), Chazelle ([9]; $O(m\alpha(m,n))$), and Pettie and Ramachandran ([36]; unknown, but optimal complexity). For the RAM model, Fredman and Willard [17] developed a linear-time algorithm.

Previous work on *directed* minimum spanning trees (sometimes called "branchings" or "arborescences"), centers around the so-called Edmonds' algorithm, its (re)discoveries and a series of implementation improvements with a complexity of $O(m + n\log(n))$ in the comparison-based model and $O(m\log\log(n))$ in the RAM model [3, 5, 10, 13, 18, 26, 30, 41].

While the majority of the research has concentrated on the total-time model, there are several other takes on how to provide input data, expect output data, and measure complexity for computing spanning trees. *Dynamic* algorithms have to maintain a (minimum) spanning tree for a graph undergoing edge insertions and deletions [23]. Such algorithms have to make changes to the existing output, meaning they cannot be used for the problem of enumerating spanning tree edges of a static graph. The same applies to the *reconfiguration* model, where an algorithm has to transform one spanning tree into another while preserving the spanning tree properties [24]. Closer to our setting is a formalization as online problem (e. g. [2, 29]). There, same as for enumeration, algorithms have to produce the final solution piece by piece. In the online setting, however, also the input arrives in pieces which only allows for approximate solutions or requires *recourse actions*. In contrast to this, enumeration algorithms have access to the complete input and are able to produce an exact solution.

2 PRELIMINARIES AND TECHNIQUES

We start by introducing graph notation, defining spanning tree problems and the enumeration framework, and explaining the techniques we use for proving upper and lower bounds.

2.1 Graph Notation and Data Structure

Let $G = (V, E)$ be a (directed or undirected) graph with vertices $V = \{1, \dots, n\}$ and $|E| = m$ edges. We will in this work assume that input graphs are connected, as spanning trees only exist for connected graphs. Note that this implies $m \geq n-1$ and allows us to shorten $O(m+n)$ to $O(m)$ in asymptotic complexity analysis. Besides the unweighted case, we also consider graphs with edge weights given as function $w\colon E \to \mathbb{R}$.

We denote the degree of a vertex v by $\deg(v)$, the average degree in the graph by $\overline{\Delta} = \frac{1}{n}\cdot\sum_{v\in V}\deg(v)$, and the maximum degree by Δ. In case G is directed, we write $\deg^+(v)$ and $\deg^-(v)$ for the out- and in-degree of v, Δ^+ and Δ^- for the maximum out- and in-degree, and $\overline{\Delta^+}$ and $\overline{\Delta^-}$ for the average out- and in-degree.

Graphs are always given in the form of adjacency lists. For a graph G and a vertex v we denote the adjacency list of v as $G[v]$. We assume that the length of each adjacency list (thus: the degree

Table 1: Our results for different problem variants. We abbreviate *preprocessing* with prep. The symbol $\mathcal{T}$ represents the runtime of an (optimal) MST total-time algorithm. An ✗ marks variants where lower bounds rule out any meaningful enumeration.

Problem Variant	Lower Bounds	Upper Bounds
Undirected Unweighted	$\Omega(m)$ prep. or $\Omega(\overline{\Delta})$ delay	$O(\overline{\Delta})$ delay w/o prep.
Undirected Weighted	$\Omega(\Delta)$ prep. or $\Omega(\Delta)$ delay	$O(\max(\Delta, \frac{\mathcal{T}}{n}))$ delay w/o prep.
	$\Omega(\mathcal{T})$ prep. or $\Omega(\frac{\mathcal{T}}{n})$ delay	$O(\frac{\mathcal{T}}{n})$ delay w/ $O(n)$ prep.
Directed Unweighted		
w/ given root	$\Omega((\Delta^+)^2)$ prep. or $\Omega(\Delta^+)$ delay	$O(\Delta^+)$ delay w/o prep.
w/o given root	$\Omega(m)$ prep. or delay	✗ $O(m)$ *total time*
Directed Weighted	$\Omega(m)$ prep. or delay	✗ $O(m + n\log(n))$ *total time [18]*

of each vertex) can be queried in constant time, and that edge weights (if present) are stored alongside the respective entry in the adjacency lists.

Our analyses assume the word RAM model with word size in $\Omega(\log(n))$; enough to store a reference to any vertex or edge in a constant number of cells that can be read and written in constant time. For edge weights we only assume that they can be compared in constant time. Note that we use existing algorithms for computing minimum spanning trees as black-boxes; your choice of algorithm there might imply additional requirements on the machine model.

2.2 Spanning Trees

A *forest* of an undirected graph $G = (V, E)$ is a subgraph $F = (V, E')$ of G with $E' \subseteq E$ that is acyclic. A *spanning tree* (ST) of G is a connected forest $T = (V, E')$ of G with $|E'| = n - 1$. With additional edge weights w a *minimum spanning tree* (MST) of G is a spanning tree that minimizes $\sum_{e \in E'} w(e)$.

An *out-branching* of a directed graph $G = (V, E)$ is an acyclic subgraph $B = (V, E')$ of G with $E' \subseteq E$ in which each vertex v has $\deg^-(v) \leq 1$. If an out-branching is weakly connected (thus: it is connected if one ignores edge directions), we have $|E'| = n - 1$ and thus there is exactly one vertex r with $\deg^-(r) = 0$. We call such an out-branching T a *directed spanning tree* (DST) of G rooted in r. With additional edge weights w a *minimum directed spanning tree* (MDST) of G is a directed spanning tree that minimizes $\sum_{e \in E'} w(e)$.

2.3 Enumeration of Solution Parts

Given an (un)directed (un)weighted input graph G, an *enumeration algorithm* has to, for some (minimum) (directed) spanning tree T of G, emit each edge of T exactly once. We analyze the time complexity of such an enumeration algorithm in terms of its worst case *delay*, that is the maximum time the algorithm needs to emit the respective next edge, and additional *preprocessing time* the algorithm can spend before emitting the first edge. The tradeoff between little preprocessing time and small delay is the core of our analyses.

For sufficiently small preprocessing time, an enumeration algorithm might not be able to initialize complex data structures before having to emit the first solution parts. We follow the approach presented in [7] to allow for *lazy-initialized* memory that can be reserved within constant time and specify for all our results how much lazy-initialized memory an algorithm needs along with its total space complexity.

2.4 Amortized Analysis for Upper Bounds

When designing and analyzing an enumeration algorithm it is important to separate *computing* solution parts from *emitting* them. While an algorithm might fix parts of the solution after irregular time intervals, it can possibly shrink the maximum output delay by holding back some solution parts to emit later. This transformation was first described in [21] and comes with the downside of having to store the held-back data. However, with the spanning tree problems at hand, an algorithm has to store at most $n - 1$ edge references, which does not increase its asymptotic space complexity.

Our algorithms hold back solution parts for later emission by storing them in a linked list called *solution queue*. The key aspect of an algorithm's description is how this queue is filled with solution parts; complemented by an analysis showing that the solution queue never runs empty when a solution part has to be emitted. Note that we expect an algorithm to actively emit held-back solution parts. While we do not explicitly write out the instructions on when to dequeue and emit a solution part from the solution queue, we do have to explain how an algorithm is able to compute a lower bound on the aspired preprocessing and delay in order to be able to emit the next solution part after a suitable number of computation steps.

We analyze the time complexity of our algorithms with the banker's view on amortization [42]: An algorithm holds a number of *credits*. Assume we claim that an algorithm solves a spanning tree enumeration problem with preprocessing in $O(p)$ and delay in $O(d)$. Initially, for some implementation-specific constant c, the algorithm starts with $c \cdot (p + d)$ credits. For every solution part it emits after $\Theta(d') \subseteq O(d)$ computation steps, it receives additional $c \cdot d'$ credits. Each credit pays for a fixed constant number of computation steps, and every step has to be paid for. To prove our claim we thus have to show that the algorithm's account balance cannot become negative. With this technique in mind, our algorithms and their proofs are roughly structured in three phases as follows:

1. **Credit Accumulation** Fix simple initial parts of the solution within little computation time and compute a lower bound on the aspired delay. Emit only enough solution parts to pay for this initial effort and hold back the rest to save credits for the next step.
2. **Extension** Extend the set of initial solution parts to a complete solution. Pay for the required computation with held-back solution parts from the credit accumulation phase.
3. **Output Finalization** Emit the remaining solution parts from phases 1 and 2 without repeating a solution part.

2.5 Proof Techniques for Lower Bounds

For proving lower bounds on the time complexity of enumerating a spanning tree we analyze how evenly the computational effort for a set of solution parts could possibly be spread over corresponding delays. These kinds of lower bounds come in two different flavors:

Any algorithm enumerating all $n-1$ edges of a spanning tree with preprocessing in $O(p)$ and delay in $O(d)$ yields a total-time algorithm with time complexity in $O(p + n \cdot d)$. Assume it takes $\Omega(\mathcal{T})$ to produce all $n-1$ edges of a spanning tree (either by an unconditional lower bound or by comparing to an algorithm with runtime in $\Theta(\mathcal{T})$ as benchmark). Then every enumeration algorithm has to have preprocessing in $\Omega(\mathcal{T})$ or delay in $\Omega(\frac{\mathcal{T}}{n})$ or it would beat the total-time bound.

For a more fine-grained analysis we observe that we do not need to have a lower bound on the total time to compute all solution parts, but can also make use of a lower bound of $\Omega(\mathcal{T}_k)$ on the required computational effort for finding any first k solution edges. Similarly to before, this yields a lower bound of either $\Omega(\mathcal{T}_k)$ on preprocessing or $\Omega(\frac{\mathcal{T}_k}{k})$ on delay. For this kind of lower bound we construct adversary arguments: We formalize an algorithm's read access to the input data as queries to an adversary. This adversary can freely choose how to answer the queries as long as all answers combined are consistent with at least one actual possible input graph. The adversary uses this freedom to hide essential information on the instance from the algorithm (usually inside densely connected subgraphs) and thereby forces the algorithm to use a lot of queries to fix the first set of k solution parts.

The interface of our adversary mimics providing the input in the form of adjacency lists. Usually, an algorithm has constant time random access to the degree of any vertex and to the head of its adjacency list. It can then iterate through the list to determine the neighbors of this vertex. Accordingly, our adversaries allow for the following queries for any vertex v:

degree query Returns the out-degree of v.
neighbor query Returns the next out-adjacency of v; along with its edge weight if present.

3 THEORETICAL RESULTS

We now apply the introduced techniques to computing (minimum) spanning trees for graphs with and without directions or edge weights.

3.1 Undirected Unweighted Spanning Trees

For connected input graphs with neither edge directions nor edge weights we derive an enumeration lower bound from an unconditional lower bound on the total time required for computing a spanning tree. Recall for the following theorem that m is the number of edges and $\overline{\Delta}$ is the average degree of the graph.

Theorem 3.1. *No algorithm can compute a spanning tree of an unweighted, undirected, connected graph in $o(m)$. No algorithm can enumerate the edges of such a spanning tree with preprocessing in $o(m)$ and delay in $o(\overline{\Delta})$.*

Proof. Consider a graph with two almost-clique components with sizes $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$ connected by a bridge of two edges as shown in Figure 2. In both components all vertices are connected to all others except for two vertices which do not share a common edge but are endpoints to the bridge connecting to the other component.

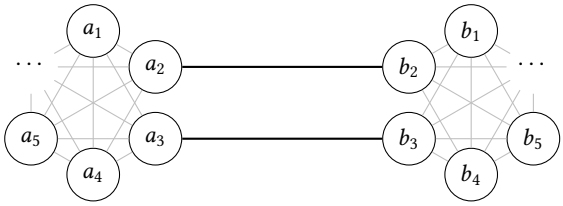

Figure 2: The adversarial graph for Theorem 3.1. An algorithm can be trapped in the cliques before being able to detect one of the two connecting edges; one of which must be part of the tree.

We use the adversarial setup described in Section 2.5. Observe that within each component all vertices have the same degree, thus an algorithm cannot identify the bridge endpoints with degree queries. Also, as long as the algorithm has not queried the complete neighborhood of at least all but 2 vertices in one component, the adversary is free to, for neighbor queries, always return an edge from the queried vertex to an adjacent vertex in the same component.

Assume, an algorithm were to return a spanning tree of the input graph after $o(m)$ queries to the adversary. Clearly, this spanning tree has to connect the two components, so let, w. l. o. g., edge $\{a_2, b_2\}$ be part of the output. But as the algorithm has not yet queried at least $\lfloor \frac{n}{2} \rfloor - 2$ complete neighborhoods of size at least $\lfloor \frac{n}{2} \rfloor - 1$, it has not seen $\{a_2, b_2\}$ and the adversary can swap the bridge endpoints in one clique such that the edge does not actually exist (and the algorithm should have returned $\{a_2, b_3\}$ or $\{a_3, b_2\}$ instead).

Thus, no algorithm can compute a complete spanning tree for a graph in $o(m)$. This implies a lower bound of $\Omega(m)$ on the total time of an enumeration algorithm for the same problem. It follows that no algorithm can enumerate the $n-1$ solution edges of a spanning tree with preprocessing in $o(m)$ and delay in $o(\frac{m}{n-1}) = o(\overline{\Delta})$. □

A simple depth first search (DFS) started on any vertex produces in $O(m)$ a spanning tree in the form of a DFS-tree, matching the lower bound in the total-time setting. The lower bound is tight for the enumeration variant as well, as we now design an algorithm that enumerates a spanning tree without preprocessing and with delay in $O(\overline{\Delta})$. Roughly, the three phases of this algorithm work as follows. In the credit accumulation phase, it collects $\frac{n}{2} \leq k \leq n-1$ edges with constant delay that form a forest. These edges give enough head start to, in the extension phase, run a modified version of Prim's MST algorithm in $O(m)$ that selects additional $n-1-k$ edges to extend the forest to a spanning tree. The output finalization phase emits the remaining edges from the first two phases.

Phase 1: Credit Accumulation. All undirected edges selected in the first phase as part of the final spanning tree are stored in a graph data structure F in adjacency lists representation; initially F is a graph with n isolated vertices. For each vertex u with empty $F[u]$, the algorithm selects the first edge $\{u, v\}$ in the adjacency list $G[u]$ and adds the edge to F (both u to $F[v]$ and v to $F[u]$ are added). Simultaneously, the algorithm computes the average degree $\overline{\Delta}$ by summing up all vertex degrees and dividing by n.

Recall from Section 2.4 that, in addition to storing edges in F, our algorithm appends all edges selected for emission to a *solution queue* from which the algorithm has to emit edges with delay in $O(\overline{\Delta})$. The first $\frac{n}{4}$ selected edges from the credit accumulation phase are emitted with constant delay (as the algorithm needs time to compute $\overline{\Delta}$). All following edges are emitted with delay in $\Theta(\overline{\Delta})$.

Phase 2: Extension. The algorithm runs a modified version of Prim's algorithm [12, 25, 37]: Starting from an arbitrary vertex, an MST is constructed by repeatedly selecting from a priority queue an edge of minimum weight that connects a previously unconnected vertex to the growing spanning tree. In our unweighted case we prioritize pre-selected edges over all other edges to make sure all edges from phase 1 are picked by the algorithm. We give a complete pseudo-code description of our modifications in Algorithm 1.

The algorithm stores the tree in the form of predecessor attributes in an array T of length n, where a vertex v connected to the tree via an edge $\{u, v\}$ has $T[v] = u$. Additionally, the algorithm maintains two queues of directed edges that are to be used next to connect vertices to the growing tree; one list for pre-selected edges and one list for all edges, where the former is prioritized in edge selection over the later. Pre-selected edges are added to both queues as filtering all adjacent edges according to whether they are part of F or not is too expensive. Skipping unneeded edges later on the other hand is cheap by checking T in line 8.

Algorithm 1: Extension-Prim

Input: undirected graph $G = (V, E)$, corresponding forest F from phase 1
Output: spanning tree encoded as predecessor links in array T of length $|V|$

```
1  preSelectedEdges = ∅, allEdges = ∅;
2  foreach v ∈ V do T[v] = NIL;
3  ENQUEUE(allEdges, (1, 1)); // fictitious start edge
4  while preSelectedEdges ≠ ∅ or allEdges ≠ ∅ do
5      if preSelectedEdges ≠ ∅ then
6          (u, v) = DEQUEUE(preSelectedEdges);
7      else (u, v) = DEQUEUE(allEdges);
8      if T[v] == NIL then
9          T[v] = u;
10         foreach w ∈ F[v] do
11             ENQUEUE(preSelectedEdges, (v, w));
12         foreach w ∈ G[v] do ENQUEUE(allEdges, (v, w));
13 T[1] = NIL; // remove fictitious start edge
14 return T;
```

Phase 3: Output Finalization. The algorithm iterates over all edges in the forest F from the first phase, sets the corresponding entries in the tree T from the second phase to NIL and adds for all remaining non-NIL-entries $T[v] = u$ the edge $\{u, v\}$ to the solution queue.

Theorem 3.2. *Enumerating the edges of a spanning tree of an unweighted, undirected, connected graph G can be done without preprocessing, with delay in $O(\overline{\Delta})$, with $\Theta(n)$ lazy-initialized memory and with space complexity in $\Theta(n)$.*

Proof. We first prove that the presented algorithm produces a valid spanning tree and emits each edge exactly once, before analyzing the algorithm's time and space complexity.

Correctness. In the credit accumulation phase each vertex is adjacent to at least one selected edge, thus $k \geq \frac{n}{2}$ edges are selected. If edge $\{u, v\}$ was selected from $G[u]$, we call u the *origin* of this edge. If there were a cycle of length c in F, each of the c vertices in the cycle would have to be the origin of one of the c cycle edges. However, after selecting the first cycle edge $\{u, v\}$ with origin u, the adjacency list $F[v]$ is non-empty and thus v cannot be the origin of any edge selected in this phase. Therefore F is acyclic and consists of $k \leq n - 1$ edges which are all enqueued for later output.

Assume for the extension phase, that pre-selected edges in F have weight 0 and all other edges in G have weight 1. Then, using the two queues and skipping edges to vertices already connected to the growing tree as shown in Algorithm 1 effectively mimics a priority queue over vertices with the priority of a vertex being the minimum weight of an edge that connects the vertex to the growing tree. Thus, correctness of Prim's algorithm implies that Algorithm 1 indeed produces a spanning tree T. As F is a forest and can be extended to a spanning tree, every min-weight spanning tree (with weights defined as assumed above) must include all edges from F, which proves $F \subseteq T$.

As the output finalization phase appends the missing edges $T \setminus F$ to the solution queue, each edge of the spanning tree T is emitted exactly once by the enumeration algorithm.

Time Complexity. We apply the accounting method to analyze the algorithm's time complexity. Each credit can be used to perform a constant number of steps. The initial balance is $\overline{\Delta} > 0$, as we are aiming for a delay of $O(\overline{\Delta})$ without preprocessing time.

Note that initially the algorithm does not know $\overline{\Delta}$, so it cannot assume a higher lower bound on the delay than $\overline{\Delta} \in \Omega(1)$. The first $\frac{n}{4}$ edges selected in the credit accumulation phase receive 6 credits each. One credit pays for the selection itself, one pays for skipping the second edge endpoint in the iteration over F. Thus, selecting and emitting each edge takes amortized constant time. The remaining 4 credits per edge accumulate to n credits that are used to compute $\overline{\Delta}$. Now that the algorithm knows a proper lower bound on the delay, the next $\frac{n}{4}$ edges receive $2 + 8\overline{\Delta}$ credits each. Again two credits pay for the selection and for skipping the second endpoint. The remaining $8\overline{\Delta}$ credits per edge accumulate to $2m$ credits that pay for all computation in phases 2 and 3 which each run in $O(m)$ total time:

In the extension phase, each undirected edge of the graph is appended to each queue at most twice (once per direction), so the loop in line 4 of Algorithm 1 runs $O(m)$ times and the queue sizes cannot exceed $O(m)$. The `if`-block in line 8 runs once per vertex, so the algorithm iterates over each vertex neighborhood at most twice, resulting in $O(m)$ steps.

The output finalization phase iterates over F once which takes $O(n)$ time. The additional edges appended to the solution queue in this phase receive no credit.

As every edge appended to the solution queue receives credit in $O(\overline{\Delta})$ and all computation steps are paid for by credit accumulated before, the delay is in $O(\overline{\Delta})$ as claimed.

Space Complexity. The credit accumulation phase uses $\Theta(n)$ lazy-initialized memory for F and puts $O(n)$ edges into the solution queue. Algorithm 1 in the extension phase uses up to $\Theta(m)$ memory for the queue of all edges. Note however that, instead of expanding the neighborhood of vertex v in lines 10ff. into the two queues immediately, we can append only the vertex to the respective list. By expanding the vertex to the edges in its neighborhood lazily in lines 5f., we thereby shrink the length of both queues and thus the overall space complexity of Algorithm 1 to $O(n)$. The output finalization phase does not require additional memory on top of the provided F and T and the solution queue of at most $n-1$ edges. □

The analysis of the space complexity also shows why the extension phase does not simply contract all pre-selected edges to find the remaining tree via a depth first search: Storing the contracted graph needs $\Theta(m)$ space in the worst case, while our Prim-based extension can be easily implemented to only use $\Theta(n)$ memory.

3.2 Undirected Minimum Spanning Trees

Any minimum weight spanning tree for an edge-weighted graph is also an unweighted spanning tree for the same graph. Therefore the bound from Theorem 3.1 carries over.

Corollary 3.3. *No algorithm can enumerate a minimum spanning tree of an edge-weighted connected graph with preprocessing in $o(m)$ and delay in $o(\overline{\Delta})$.*

Given that no deterministic comparison-based MST algorithm with complexity in $O(m)$ is known, we can formulate a higher lower bound conditioned on the runtime of total-time MST algorithms, that generically applies to different machine models:

Corollary 3.4. *Let $\mathcal{T}$ be the runtime of an optimal MST algorithm. No algorithm can enumerate an MST of an edge-weighted connected graph with preprocessing in $o(\mathcal{T})$ and delay in $o(\frac{\mathcal{T}}{n})$.*

For smaller preprocessing time, we can use an adversary argument to give an unconditional and even higher lower bound on the delay: As illustrated in Figure 3, an adversary can force any algorithm to inspect either the degree of many vertices in a clique or the complete neighborhood of a single clique vertex before finding the first solution edge.

Theorem 3.5 (*). *No algorithm can enumerate a minimum spanning tree of an edge-weighted connected graph with at least two different edge weights, maximum degree Δ and average degree $\overline{\Delta} \in o(\Delta)$ with both preprocessing and delay in $o(\Delta)$.*

Without preprocessing, we can match the combination of the delay lower bounds of Theorem 3.5 and Corollary 3.4 with a corresponding upper bound; assuming the upper bound on the total time of the referenced algorithm in Corollary 3.4 is computable. Our enumeration algorithm essentially runs one iteration of Borůvka's algorithm [4] in the credit accumulation phase, which yields at least $\frac{n}{2}$ solution edges. These initial solution edges give enough head start to, in the extension phase, run a total-time MST algorithm of our choice as black box to complete the tree. As we have to make sure, that the black box algorithm produces a minimum spanning tree that includes the pre-selected edges, we set their weight to a new minimum weight (or answer edge weight comparisons accordingly in the comparison model).

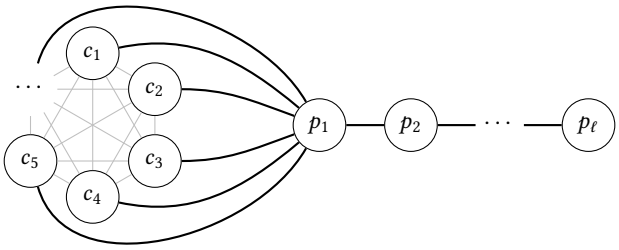

Figure 3: The adversarial input for the proof of Theorem 3.5 to trap an algorithm in the clique (many edges, no solution parts) before it can process the path (few edges, all are solution parts).

Theorem 3.6 (*). *Let $\mathcal{T}$ be an upper bound on the runtime of an MST algorithm such that $\mathcal{T}$ is computable in $O(\Delta n)$ steps and let $\mathcal{S}$ be its space complexity. Enumerating the edges of an MST of an edge-weighted connected graph G can be done without preprocessing, with delay in $O(\max(\Delta, \frac{\mathcal{T}}{n}))$, and with space complexity in $\Theta(n+\mathcal{S})$.*

For graphs with large Δ one can trade in $\Theta(n)$ preprocessing time to match the smaller delay lower bound of $O(\frac{\mathcal{T}}{n})$: The algorithm goes through the vertices in order of increasing degree to build enough head start before dealing with high-degree vertices later.

Theorem 3.7 (*). *Let $\mathcal{T}$ be an upper bound on the runtime of an MST algorithm such that $\mathcal{T}$ is computable in $O(\overline{\Delta} n)$ steps and let $\mathcal{S}$ be its space complexity. Enumerating the edges of an MST of an edge-weighted connected graph G can be done with preprocessing in $\Theta(n)$, with delay in $O(\frac{\mathcal{T}}{n})$, and with space complexity in $\Theta(n+\mathcal{S})$.*

Figure 4 summarizes the tradeoff between preprocessing time and delay for the enumeration of edges of a minimum spanning tree for an edge-weighted graph. The figure highlights a gap for preprocessing in $\Omega(\Delta) \cap o(n)$ and delay in $\Omega(\frac{\mathcal{T}}{n}) \cap o(\Delta + \frac{\mathcal{T}}{n})$, where we do not yet know whether enumeration is possible or not.

3.3 Directed Unweighted Spanning Trees

We analyze two variants of computing a directed spanning tree (DST) for a graph that is guaranteed to have one: First, if a suitable root vertex r for such a DST is given, second without given root.

Given r, the search tree of a depth first search started in r is a directed spanning tree [11].

Corollary 3.8. *Given an unweighted directed graph G that has a directed spanning tree rooted in a given vertex r, such a DST can be computed in $O(m)$ by a DFS.*

Without given root this runtime is still achievable, as one can find a suitable root in $O(m)$: Build the acyclic graph G^{SCC} in which each strongly connected component is contracted to a single node [11]. Pick any vertex in the unique source of G^{SCC} as root.

Theorem 3.9 (*). *Given an unweighted directed graph G that has a directed spanning tree, such a DST can be computed in $O(m)$.*

The lower bound from Theorem 3.1 for undirected graphs extends to the directed case:

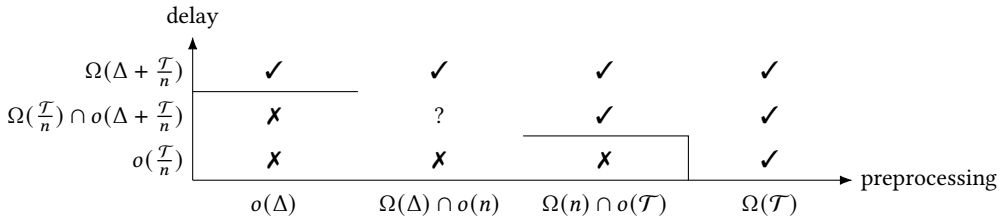

Figure 4: Tradeoff between preprocessing and delay for MST enumeration with a black-box MST algorithm with total time $\mathcal{T}$.

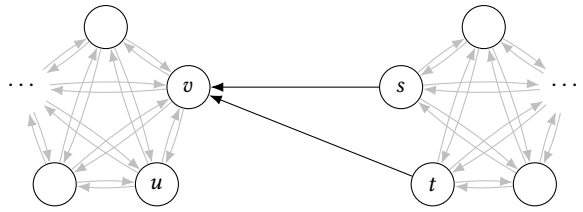

Figure 5: The prototype graph for the adversary for the proof of Theorem 3.11. A DST enumeration algorithm cannot emit an edge (u, v) before having found the crossing edges.

Corollary 3.10. *Given is an unweighted directed graph G that has a DST. No algorithm can compute a DST in $o(m)$. No algorithm can enumerate a DST with preprocessing in $o(m)$ and delay in $o(\overline{\Delta^+})$.*

For the total-time variant this is already a tight bound. We can, however, prove a stronger lower bound for the enumeration of directed spanning trees without given root, that rules out any meaningful enumeration even for dense graphs. The core idea of the proof is that any vertex in a DST is the endpoint of at most one edge. Were an algorithm to emit an edge (u, v) early, the adversary could point key edges (in Figure 5 these are crossing edges between two cliques), to vertex v; and one of these key edges has to be included in any DST, making (u, v) wrong.

Theorem 3.11 (*). *Given is an unweighted directed graph that has a DST. Without given root, no algorithm can enumerate such a DST with both preprocessing and delay in $o(m)$.*

Corollary 3.8 and Theorem 3.9 show that providing a DST root or not does not change the time complexity in the total-time model. This is different in the enumeration variant, as with given root, one *can* enumerate the edges of a DST with delay in $O(\Delta^+)$ by adjusting the BFS variation given in the proof of Theorem 1 in [7]. Starting this shortest distance enumeration with root s, we also store and then emit the edges used to reach a vertex on a shortest path from s. This adjustment outputs a shortest distances tree, thus in particular a DST, and immediately yields the following result.

Corollary 3.12. *Given an unweighted directed graph G that has a directed spanning tree rooted in a given vertex r, such a DST can be enumerated with delay in $O(\Delta^+)$, without preprocessing, with $\Theta(n)$ lazy-initialized memory and space complexity in $\Theta(n)$.*

Note that this leaves a gap to the $\Omega(\overline{\Delta^+})$ delay lower bound from Corollary 3.10. While the $\Omega(\Delta)$ delay lower bound for single source shortest distances from [7] does not transfer to the DST problem, we can use a similar technique to also show a matching lower bound for directed spanning tree enumeration: An adversary can hide the entrance to a bi-directed path behind many edges in a clique. For suitable clique- and path sizes this forces an algorithm to essentially fully explore the clique with many edges and few solution parts before being able to emit one of the many solution parts in the path.

Theorem 3.13. *Given is an unweighted directed graph G that has a directed spanning tree rooted in a given vertex r. No algorithm can enumerate such a DST with preprocessing in $o((\Delta^+)^2)$ and delay in $o(\Delta^+)$, even if $\overline{\Delta^+} \in o(\Delta^+)$.*

Proof. We use the adversarial setup from Section 2.5 with k to be chosen later. Consider a graph consisting of a bi-directed clique of k vertices and a bi-directed path of $n - k$ vertices. The given root r lies in the clique. One other clique vertex c has no edge to r and instead an edge to one of the two end-vertices p or q of the path. (See Figure 6 for an illustration.) Note that this graph has maximum degree $\Delta^+ = k - 1$.

The adversary places each edge to r and also the edge from c to p or q last in the respective adjacency lists. Further, observe that for any degree query in the clique, the adversary answers $k - 1$, so no algorithm can identify c with degree queries.

Assume, after less than $(k-1)^2$ queries to the adversary, an algorithm emits an edge (u, v) among two path vertices or an edge that connects the clique to the path. Even if all the algorithm's queries are adjacency queries on clique vertices, the adversary can up to this point only answer with edges within the clique. Further, there is at least one vertex in the clique where the algorithm has not seen the whole neighborhood. The adversary chooses this vertex as c and connects it either to p or q, picking the endpoint of the path that makes the emitted edge wrong.

Thus, any algorithm has to make at least $(k-1)^2$ queries before emitting a solution edge outside of the clique. As there can only be $k - 1$ tree edges inside the clique, the adversarial setup shows a lower bound on emitting the k-th edge of $\Omega(\mathcal{T}_k) = \Omega(k^2)$. With $\Delta^+ = k - 1$, this shows a lower bound of $\Omega(k^2) = \Omega((\Delta^+)^2)$ for preprocessing or $\Omega(k) = \Omega(\Delta^+)$ for delay.

It remains to pick k to show the desired low average degree. The graph consists of $k \cdot (k-1) + 2 \cdot (n - k - 1)$ edges, so for $k \in o(n)$ the average degree is $\overline{\Delta^+} \in o(\Delta^+)$. □

3.4 Directed Minimum Spanning Trees

Efficient implementations of *Edmonds' algorithm* need little more than linear time to solve any instance completely [18, 30]. A simple

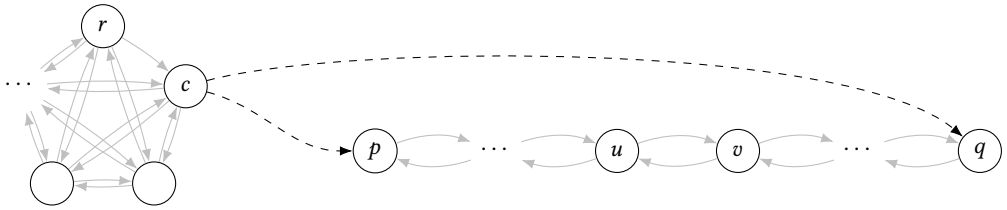

Figure 6: The adversarial graph for Theorem 3.13. A DST enumeration algorithm cannot emit an edge from the bi-directed path before having found the outgoing edge from the clique.

adversary construction rules out essentially any meaningful enumeration, as any algorithm has to inspect almost all of the input graph as preprocessing: Before emitting a first edge (u, v) as part of an MDST for a bi-directed clique, an algorithm has to query $\Theta(m)$ adjacencies to find all edges incident to v to make sure that the single spanning tree edge incident to v is one of minimum weight.

Theorem 3.14 (*). *Minimum directed spanning tree enumeration cannot be solved with both preprocessing and delay in $o(m)$.*

4 EXPERIMENTAL EVALUATION

We implemented MST algorithms in Rust and executed the experiments on a compute server with 256 GB RAM and an Intel Xeon Silver 4314 CPU with 2.40 GHz. More specifically, we compared the total time MST algorithms by Prim [37] (with a binary heap [38]), Kruskal [27] (with union-find with union-by-rank [11] and path-halfing [43]), and Boruvka [4], with our enumeration approach of Theorem 3.7 with each of the other three algorithms as black box. Additionally, as Prim's algorithm already fixes edges one at a time, we re-interpreted it as enumeration algorithm; meaning that whenever the algorithm fixes an MST edge, the algorithm is interrupted and the edge is immediately emitted as solution part. Note that for the enumeration variants we did not implement holding back solution parts in a queue, but instead emitted each solution edge as soon as it was fixed.

For the evaluation we generated random graphs in the $G(n, p)$ model [20] with the number n of vertices ranging from 100 to 200 000 and an edge probability p between $n^{-0.75}$ and 0.25. For each input size we ran the algorithms on 10 random instances with 5 runs each and took the average time in nanoseconds for, among others, three values:

- The time until the algorithm produced a first solution part;
- the maximum so-called *incremental* delay, meaning the maximum of $\frac{\text{elapsed time}}{\text{number of emitted parts}}$;
- the total time of the run.

Note that while inspecting the worst case delay gives stronger bounds in the theoretical analysis, looking at incremental delay is the more reasonable measurement in practice. This avoids computational overhead for holding back solution parts and still guarantees the availability of the kth part after $k \cdot$ *incremental delay* time.

For the generated instances, Prim's total-time algorithm and its enumeration variant consistently performed best among their respective algorithm categories. Figure 7 shows the direct comparison of (the re-interpreted) Prim's algorithm with the enumeration approach of Theorem 3.7 with Prim's algorithm as black box. As

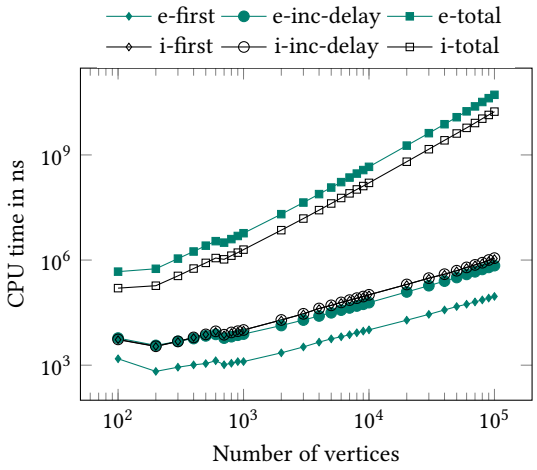

Figure 7: Time to first output, incremental delay and total time for (e) MST enumeration with Prim's algorithm as black box and (i) MST computation with Prim's algorithm, interrupted at each fixed solution part.

expected, the additional effort for enumerating solution parts early leads to a larger total time. However, the enumeration algorithm produces the first solution part one order of magnitude faster and also has the benefit of a smaller incremental delay. This confirms that the theoretical advantage of the enumeration variant transfers to practice if the time to a first solution part and/or the delay are of more importance than the total time.

5 CONCLUSIONS AND FUTURE WORK

In this work we proved upper and lower bounds on the required preprocessing and delay for enumerating the edges of (minimum) (directed) spanning trees. While several of our theoretical results are already tight, there still are gaps to be closed (cf. Figure 4).

An important next step is to incorporate more real world requirements in the model: Most networks admit for many different MSTs, and for some problems, not all of them work equally well [1]. Also, the order in which tree edges are emitted can be relevant to efficiently interleave MST computation with following processing steps. Thus, to bring our theoretical results closer to practical application, additional requirements on the produced output need to be investigated with regard to the possibility of enumerating solution parts with little preprocessing and delay.

REFERENCES

[1] Noa Agmon, Noam Hazon, Gal A. Kaminka, and The MAVERICK Group. 2008. The Giving Tree: Constructing Trees for Efficient Offline and Online Multi-Robot Coverage. *Annals of Mathematics and Artificial Intelligence* 52, 2 (April 2008), 143–168. https://doi.org/10.1007/s10472-009-9121-1

[2] Magnus Berg, Joan Boyar, Lene M. Favrholdt, and Kim S. Larsen. 2023. Online Minimum Spanning Trees with Weight Predictions. In *Algorithms and Data Structures (WADS)*, Pat Morin and Subhash Suri (Eds.). Springer Nature Switzerland, Cham, 136–148. https://doi.org/10.1007/978-3-031-38906-1_10

[3] F. Bock. 1971. An Algorithm to Construct a Minimum Directed Spanning Tree in a Directed Network. In *Developments in Operations Research*. Gordon and Breach, New York, 29–44.

[4] Otakar Borůvka. 1926. O jistém problému minimálním. *Práce Moravské přírodovědecké společnosti* 3, 3 (1926), 37–58.

[5] Paolo M. Camerini, Luigi Fratta, and Francesco Maffioli. 1979. A Note on Finding Optimum Branchings. *Networks* 9, 4 (1979), 309–312. https://doi.org/10.1002/net.3230090403

[6] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. 2022. Answering (Unions of) Conjunctive Queries Using Random Access and Random-Order Enumeration. *ACM Transactions on Database Systems* 47, 3 (Aug. 2022), 9:1–9:49. https://doi.org/10.1145/3531055

[7] Katrin Casel, Tobias Friedrich, Stefan Neubert, and Markus L. Schmid. 2024. Shortest Distances as Enumeration Problem. *Discrete Applied Mathematics* 342 (Jan. 2024), 89–103. https://doi.org/10.1016/j.dam.2023.08.027

[8] Katrin Casel and Stefan Neubert. 2025. Emit As You Go: Enumerating Edges of a Spanning Tree. https://doi.org/10.48550/arXiv.2502.10279 arXiv:2502.10279 [cs]

[9] Bernard Chazelle. 2000. A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity. *J. ACM* 47, 6 (Nov. 2000), 1028–1047. https://doi.org/10.1145/355541.355562

[10] Yeong-Jin Chu and Tseng-Hong Liu. 1965. On the Shortest Arborescence of a Directed Graph. *Scientia Sinica* 14 (1965), 1396–1400.

[11] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms* (4. ed.). The MIT Press, Cambridge, Massachusett.

[12] Edsger W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (Dec. 1959), 269–271. https://doi.org/10.1007/BF01386390

[13] Jack Edmonds. 1967. Optimum Branchings. *Journal of Research of the National Bureau of Standards Section B Mathematics and Mathematical Physics* 71B, 4 (Oct. 1967), 233. https://doi.org/10.6028/jres.071B.032

[14] Jeff Erickson. 2019. *Algorithms.* self-published, Illinois.

[15] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. 2004. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision* 59, 2 (Sept. 2004), 167–181. https://doi.org/10.1023/B:VISI.0000022288.19776.77

[16] Michael L. Fredman and Robert E. Tarjan. 1987. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *J. ACM* 34, 3 (July 1987), 596–615. https://doi.org/10.1145/28869.28874

[17] Michael L. Fredman and Dan E. Willard. 1994. Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comput. System Sci.* 48, 3 (June 1994), 533–551. https://doi.org/10.1016/S0022-0000(05)80064-9

[18] Harold N. Gabow, Zvi Galil, Thomas Spencer, and Robert E. Tarjan. 1986. Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs. *Combinatorica* 6, 2 (June 1986), 109–122. https://doi.org/10.1007/BF02579168

[19] Marek Gagolewski, Anna Cena, Maciej Bartoszuk, and Łukasz Brzozowski. 2024. Clustering with Minimum Spanning Trees: How Good Can It Be? *Journal of Classification* (July 2024). https://doi.org/10.1007/s00357-024-09483-1

[20] Edgar Nelson Gilbert. 1959. Random Graphs. *The Annals of Mathematical Statistics* 30, 4 (Dec. 1959), 1141–1144. https://doi.org/10.1214/aoms/1177706098

[21] Leslie Ann Goldberg. 1991. *Efficient Algorithms for Listing Combinatorial Structures.* Ph.D. Dissertation. University of Edinburgh, Edinburgh.

[22] Ronald L. Graham and Pavol Hell. 1985. On the History of the Minimum Spanning Tree Problem. *IEEE Annals of the History of Computing* 7, 1 (1985), 43–57. https://doi.org/10.1109/MAHC.1985.10011

[23] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. 2001. Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity. *J. ACM* 48, 4 (July 2001), 723–760. https://doi.org/10.1145/502090.502095

[24] Takehiro Ito, Erik D. Demaine, Nicholas J.A. Harvey, Christos H. Papadimitriou, Martha Sideri, Ryuhei Uehara, and Yushi Uno. 2011. On the Complexity of Reconfiguration Problems. *Theoretical Computer Science* 412, 12-14 (March 2011), 1054–1065. https://doi.org/10.1016/j.tcs.2010.12.005

[25] Vojtěch Jarník. 1930. O jistém problému minimálním. (Z dopisu panu O. Borůvkovi). *Práce moravské přírodovědecké společnosti* 6, 4 (1930), 57–63.

[26] Richard M. Karp. 1971. A Simple Derivation of Edmonds' Algorithm for Optimum Branchings. *Networks* 1, 3 (1971), 265–272. https://doi.org/10.1002/net.3230010305

[27] Joseph B. Kruskal. 1956. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proc. Amer. Math. Soc.* 7, 1 (1956), 48–50. https://doi.org/10.2307/2033241 arXiv:2033241

[28] Martin S. Lindner, Benjamin Strauch, Jakob M. Schulze, Simon H. Tausch, Piotr W. Dabrowski, Andreas Nitsche, and Bernhard Y. Renard. 2017. HiLive: Real-Time Mapping of Illumina Reads While Sequencing. *Bioinformatics* 33, 6 (March 2017), 917–319. https://doi.org/10.1093/bioinformatics/btw659

[29] Nicole Megow, Martin Skutella, José Verschae, and Andreas Wiese. 2012. The Power of Recourse for Online MST and TSP. In *Automata, Languages, and Programming*, Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer (Eds.). Springer, Berlin, Heidelberg, 689–700. https://doi.org/10.1007/978-3-642-31594-7_58

[30] Ran Mendelson, Robert E. Tarjan, Mikkel Thorup, and Uri Zwick. 2006. Melding Priority Queues. *ACM Transactions on Algorithms* 2, 4 (Oct. 2006), 535–556. https://doi.org/10.1145/1198513.1198517

[31] Gonzalo Navarro and Rodrigo Paredes. 2010. On Sorting, Heaps, and Minimum Spanning Trees. *Algorithmica* 57, 4 (Aug. 2010), 585–620. https://doi.org/10.1007/s00453-010-9400-6

[32] Jaroslav Nešetřil. 1997. A Few Remarks on the History of MST-problem. *Archivum Mathematicum* 33, 1 (1997), 15–22.

[33] Stefan Neubert and Katrin Casel. 2024. Incremental Ordering for Scheduling Problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 34. AAAI Press, Washington, DC, USA, 405–413. https://doi.org/10.1609/icaps.v34i1.31500

[34] Vitaly Osipov, Peter Sanders, and Johannes Singler. 2009. The Filter-Kruskal Minimum Spanning Tree Algorithm. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX) (Proceedings).* Society for Industrial and Applied Mathematics, Philadelphia, PA, 52–61. https://doi.org/10.1137/1.9781611972894.5

[35] Rodrigo Paredes and Gonzalo Navarro. 2006. Optimal Incremental Sorting. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX) (Proceedings).* Society for Industrial and Applied Mathematics, Philadelphia, PA, 171–182. https://doi.org/10.1137/1.9781611972863

[36] Seth Pettie and Vijaya Ramachandran. 2002. An Optimal Minimum Spanning Tree Algorithm. *J. ACM* 49, 1 (Jan. 2002), 16–34. https://doi.org/10.1145/505241.505243

[37] Robert C. Prim. 1957. Shortest Connection Networks and Some Generalizations. *The Bell System Technical Journal* 36, 6 (Nov. 1957), 1389–1401. https://doi.org/10.1002/j.1538-7305.1957.tb01515.x

[38] Hideki Sekine. 2022. Binary_heap_plus. https://docs.rs/binary-heap-plus/0.5.0/binary_heap_plus/

[39] Yann Strozecki. 2019. Enumeration Complexity. *Bulletin of EATCS* 3, 129 (Oct. 2019), 33.

[40] Chee Sheng Tan, Rosmiwati Mohd-Mokhtar, and Mohd Rizal Arshad. 2021. A Comprehensive Review of Coverage Path Planning in Robotics Using Classical and Heuristic Algorithms. *IEEE Access* 9 (2021), 119310–119342. https://doi.org/10.1109/ACCESS.2021.3108177

[41] Robert E. Tarjan. 1977. Finding Optimum Branchings. *Networks* 7, 1 (1977), 25–35. https://doi.org/10.1002/net.3230070103

[42] Robert E. Tarjan. 1985. Amortized Computational Complexity. *SIAM Journal on Algebraic Discrete Methods* 6, 2 (April 1985), 306–318. https://doi.org/10.1137/0606031

[43] Theodorus Petrus van der Weide. 1980. *Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms.* Ph.D. Dissertation. Mathematisch Centrum, Amsterdam.

[44] Charles T. Zahn. 1971. Graph-Theoretical Methods for Detecting and Describing Gestalt Clusters. *IEEE Trans. Comput.* C-20, 1 (Jan. 1971), 68–86. https://doi.org/10.1109/T-C.1971.223083

[45] Yue Zhang, Zhe Chen, Daniel Harabor, Pierre Le Bodic, and Peter J. Stuckey. 2024. Planning and Execution in Multi-Agent Path Finding: Models and Algorithms. *Proceedings of the International Conference on Automated Planning and Scheduling* 34 (May 2024), 707–715. https://doi.org/10.1609/icaps.v34i1.31534