Efficiently Computing Directed Minimum Spanning Trees*

Maximilian Böther † Otto Kißig ‡ Christopher Weyand §

Abstract

Computing a directed minimum spanning tree, called arborescence, is a fundamental algorithmic problem, although not as common as its undirected counterpart. In 1967, Edmonds discussed an elegant solution. It was refined to run in $O(\min(n^2, m \log n))$ by Tarjan which is optimal for very dense and very sparse graphs. Gabow et al. gave a version of Edmonds' algorithm that runs in $O(n \log n + m)$, thus asymptotically beating the Tarjan variant in the regime between sparse and dense. Despite the attention the problem received theoretically, there exists, to the best of our knowledge, no empirical evaluation of either of these algorithms. In fact, the version by Gabow et al. has never been implemented and, aside from coding competitions, all readily available Tarjan implementations run in $O(n^2)$. In this paper, we provide the first implementation of the version by Gabow et al. as well as five variants of Tarjan's version with different underlying data structures. We evaluate these algorithms and existing solvers on a large set of real-world and random graphs.

1 Introduction

The minimum spanning tree problem is well studied with various applications [15, 33] and algorithms [19, 25, 28]. The directed version, called the minimum spanning arborescence problem, has received much less attention. For a given root r, it aims at finding a directed spanning tree of minimum weight rooted at r. Applications include infection chain modeling [20] and approximating traveling salesperson instances [29]. Different versions and generalizations were studied [14, 22, 21]. Sometimes multiple roots are given or it is required to find the best root. Historically, the problem was to find a set of non-overlapping trees with maximum total weight, called an optimum branching. As these versions are linear time equivalent [7, 27], we focus on the minimum spanning arborescence problem with given root.

The algorithm to find a minimum spanning arborescence was discovered independently by Edmonds [7], Chu [5], and Bock [2]. Karp [23] was the first to give a combinatorial proof of correctness. Following the literature, we call it Edmonds' algorithm. The algorithm runs in $O(nm)$ and forms the basis for later, more elaborate versions by Tarjan [35, 4] running in $O(\min(n^2, m \log n))$ and Gabow et al. [12] running in $O(n \log n + m)$, which we call the GGST algorithm in the following. There exist parallel algorithms for different settings of distributed computing [26, 9]. They are based on Edmond's Algorithm as well but we will focus solely on the sequential setting. Both Tarjan's versions and GGST have the same complexity for very sparse and very dense graphs while the GGST version beats Tarjan's by a logarithmic factor for the regime in between. GGST likely is optimal since the minimum spanning arborescence problem can be reduced to (s,t)-shortest path [9] and comparison based sorting can be reduced to determining the order of contractions performed during Edmonds' algorithm [12]. However, a time of $O(m \log \log n)$ was obtained in the word RAM model with Tarjan's version [27]. Moreover, Tarjan's version was shown to run in $O(n \log^2 n + m)$ on Erdős-Rényi graphs with random weights [35, 8].

To the best of our knowledge, no experimental evaluation of these algorithms, or even an implementation of GGST, exist. The latter is likely due to the rather technical description and the fact that the algorithm is not the main result of the corresponding paper. On the other hand, there exist some efficient (meaning $O(m \log n)$) implementations of Tarjan's version. The problem is a niche topic in coding competitions such as the International Collegiate Programming Contest (ICPC). Unfortunately, they are hard to find because most of them are only documented as submissions in online judge systems. The only ready-to-use library implementations run in $O(n^2)$. This paper provides accessible descriptions and implementations as well as a detailed evaluation. Our code is open source and can be found in our public repository[1]. The core contributions of this paper include

- five Tarjan implementations with different underlying data structures, one of which beats existing solvers on most instances,
- a high level description of the GGST algorithm with several optimizations/simplifications,

*The full version of the paper can be accessed at `https://arxiv.org/abs/2208.02590`

†ETH Zurich.
‡Hasso Plattner Institute.
§Karlsruhe Institute of Technology.

[1] `https://github.com/chistopher/arbok`

- an efficient implementation of the GGST algorithm,
- and a detailed experimental evaluation on a large number of real-word and synthetic networks.

In Section 2 we describe Edmonds' algorithm along with the versions by Tarjan and Gabow et al. Section 3 describes the existing and new implementations and optimization techniques. The experimental evaluation is presented in Section 4. We conclude in Section 5.

2 Edmonds' Arborescence Algorithm

We discuss Edmonds' algorithm in Section 2.1, Tarjan's version in Section 2.2, and the GGST version in Section 2.3. The latter two yield just the weight of the optimal solution, not the actual edges. Reconstructing the edge set is discussed in Section 2.4.

2.1 Edmonds' Original Version Edmonds' algorithm works as follows. For each vertex $v \neq r$, pick the cheapest incoming edge $\pi(v)$. If the set of these $n-1$ edges contains no cycles, it is an arborescence; otherwise, it is possible to show that there is an optimal solution that contains all chosen edges except one for each cycle. To determine which edge of each cycle to remove, Edmonds' algorithm contracts each cycle. Note that a vertex is part of at most one cycle. The weight of all edges going into a cycle C is reduced as follows. An edge pointing at vertex $v \in C$ is reduced by the weight of $\pi(v)$, i.e., the weight of the cheapest edge incoming into v. We then compute a solution on the contracted graph. The resulting solution has an incoming edge for each cycle C we contracted. This edge corresponds to an original edge (u, v) with $v \in C$, which we use to replace the cycle edge $\pi(v)$ we picked earlier.

The correctness is based on the following fact. Adding a constant Δ to all incoming edge weights of a vertex changes the weight of each arborescence by Δ, since each solution picks exactly one of those. This means that the edge cost changes performed during the algorithm preserve the optimal solution. Moreover, the cycle edges all get a cost of zero. Thus, the final cost is the same, no matter which edge is replaced.

2.2 Tarjan's Version Tarjan proposed a version of Edmonds' algorithm that, given the right data structures, runs in $O(m \log n)$ or $O(n^2)$ [35]. It features two major improvements. First, the cycle expansion and removal of one edge per cycle is detached from the main algorithm and seen as a postprocessing step. The algorithm tracks all chosen edges as a superset of the solution, which can be reconstructed afterwards in linear time. The second change is to formulate the algorithm sequentially in such a way to avoid rebuilding the graph for each contraction. The approach goes as follows. While there is a vertex other than the root that was not processed yet, its cheapest, incoming, edge that is not a self-loop is added to the solution. If this edge forms a cycle with previously chosen edges, the cost of edges into the cycle is changed as in Edmonds' algorithm and the cycle vertices including their incoming edges are merged into a vertex representing the cycle. This vertex is then added to the queue of unprocessed vertices.

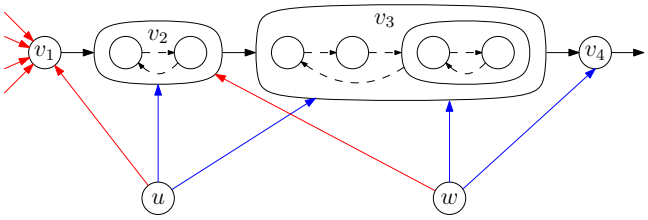

Figure 1: Visualization of the growth path with the first four vertices v_1, v_2, v_3, v_4. Also shown are the exit lists of two arbitrary vertices u, w. Active edges are red and passive edges blue.

The algorithm requires data structures to find the cheapest incoming edge, recognize cycles of chosen edges, and to track contractions. The latter two can be achieved with disjoint set union (DSU) data structures such as a disjoint set forest [13, 34]. To find cycles, a DSU maintains weakly connected components with respect to the chosen edges. Note that each vertex has at most one incoming chosen edge. Thus, an edge closes a directed cycle with previously chosen edges, if and only if, it connects two vertices in the same weakly connected component. A second DSU is used to manage contractions and to map original vertices to contracted vertices. The endpoints of edges are not updated after each contraction. Instead, a DSU lookup is required each time the algorithm handles an edge.

The data structure to maintain incoming edge sets must support four operations: (1) add an element, (2) extract the minimum element, (3) change the weight of all elements in the set by a constant, and (4) merge two sets. If all operations take at most logarithmic time, the algorithm runs in $O(m \log n)$. Most mergeable heaps (e.g., hollow heaps, treaps, skew heaps) support operations (1), (2), and (4) and can be extended with lazy propagation to allow for operation (3). Alternatively, if operations 2-4 run in $O(n)$, e.g., when using an adjacency matrix, the algorithm runs in $O(n^2)$, which is better for dense graphs.

2.3 GGST Version Gabow et al. [12] further refine the version given by Tarjan to reduce the running time to $\mathcal{O}(n \log n + m)$. They use the last remaining degree

of freedom, namely the order in which vertices are processed. The authors suggest to always choose the vertex from where the last incoming edge originated thus forming a path of chosen vertices, called the *growth path*. To avoid special cases for when the path reaches the root, they add dummy edges with cost 0 from the root to all other vertices, making the graph fully connected. These edges do not affect running time but simplify the description since the algorithm becomes oblivious to the root and the additional edges can be removed in the reconstruction phase. The improved running time is achieved by exploiting the structure of the path and clever handling of associated edges. For each vertex, whether on the growth path or not, an *exit list* contains outgoing edges pointing into the growth path. Exit lists are sorted by the position of their target vertex in the growth path, i.e., the first edge points closest to the head of the path. The first edge in each exit list is called *active*; all others *passive*. The active edges are maintained in a data structure we call an *active forest*. Figure 1 shows an example. In the following, we give a concise, yet comprehensive, description of the algorithm that differs from the original discussion in the level of abstraction and simplifies the logic and data structures.

The algorithm starts at an arbitrary vertex and repeatedly picks the cheapest incoming edge of the path head until the path covers the whole graph. Each iteration, the path is either extended or contracted. If the origin of the picked edge is not yet on the growth path, then it becomes the new path head. If it is already on the growth path, then the prefix of the growth path up to this vertex forms a cycle, which is contracted into a single vertex that becomes the new path head. The process is summarized in Algorithm 1. As in Tarjan's version, contractions are tracked with a DSU [13, 34] which handles the $find(u)$ calls.

Growth Path Extension. When the growth path is extended by a new vertex u, all incoming edges of u are introduced to the algorithm and inserted into their respective exit lists. Consider the insertion of an edge, say (x, u), into x's exit list. Since u just became the new head of the growth path, the edge will be inserted at the front of the exit list. It will become active and, if the exit list was not empty, the previously active edge will become passive. Because u just became part of the growth path, it is not a contracted vertex. However x may be on the growth path and therefore may be a contracted vertex. As such, x may have multiple outgoing edges to u, originating from different vertices inside x. To deal with this issue (and with multi-edges in the input), one checks if the first edge in the exit list already points to u and if so, only keeps the cheaper one. This limits the exit list to at most one edge to u maintaining the invariant that an exit list never contains two edges to the same vertex.

Algorithm 1: Minimum arborescence algorithm by Gabow et al. [12]

```
1  initialize growth path with arbitrary vertex;
2  insert its incoming edges into exit lists;
3  while not all vertices on growth path do
4      query min. incoming edge (u, v) of path head
         from active forest;
5      remember (u, v) for reconstruction;
6      if find(u) is not on growth path then
7          insert u's incoming edges into exit lists;
8      else
9          delete prefix of path up to last occurrence
             of find(u);
10         update incoming edge costs for all vertices
             on prefix;
11         delete outgoing edges of prefix from exit
             lists;
12         merge prefix in DSU and Active Forest;
13         limit edges into the cycle to at most 1 per
             origin;
14     insert find(u) at front of path;
```

Growth Path Contraction. When a prefix of the path forms a cycle, it is contracted just as in Edmonds' algorithm. That is, the prefix is removed from the path, incoming edges into the cycle are reduced in cost, edges resulting in self loops are deleted, the cycle vertices are contracted in the DSU as well as in the active forest, and multi-edges are removed.

The cost reduction is done with the DSU, which can be modified to track an offset for each vertex [12]. Whenever the current cost of an edge is needed, a DSU lookup analogous to a $find$ is made to get the offset of the target vertex.

Self loops are outgoing edges from the cycle, so by deleting all edges in exit lists of cycle vertices, self loops are avoided. This also deletes edges pointing further down the path but these are irrelevant to the algorithm. They can only become incoming edges of the head if, in the future, the path is contracted up to their target and in this case they would be self loops.

Edges that became multi-edges by the contraction are consolidated. For each vertex with more than one edge into the cycle, the prefix of their exit list that points into the cycle is deleted except for the cheapest of those edges. If a vertex has more than one edge into the cycle, at least one of them is passive. Thus, such vertices can be found by maintaining for each vertex on the growth path a list of incoming passive edges, called a *passive*

list.

Active Forest. The active forest maintains all currently active edges and must be updated accordingly. It stores for each vertex the outgoing active edge and a set of incoming active edges. We associate an active edge with the vertex it originates from. The active forest is able to INSERT an active edge for a vertex that does not yet have one in $O(1)$, REPLACE the active edge of a vertex by another one that points closer to the growth path head or points to the same vertex but has less weight in $O(1)$, DELETE the active edge of a vertex in $O(\log n)$, MERGE the sets of incoming active edges for the first two vertices of the growth path in constant time, and QUERY the minimum incoming active edge of the path head in $O(\log n)$ amortized time.

It is implemented as follows. Each vertex stores its incoming active edges in a Fibonacci heap [11], which enables the operations INSERT, DELETE, MERGE, and QUERY by just mapping them to the corresponding Fibonacci heap operations. The REPLACE operation could be implemented as a DELETE followed by an INSERT. Unfortunately, this results in a running time of $O(\log n)$. Instead, Gabow et al. [12] suggest to reuse the internal heap node representing the old edge. The node is *moved* from the heap the old edge is in to the heap where the new edge should be and receives the new edge as key. This *move* takes $O(1)$ time and is the crucial point where the logarithmic factor over Tarjan's version is saved. The move operation is possible by restricting QUERY, REPLACE, and MERGE to the structure of the growth path. In general, no mergable heap data structure is known that lifts these restrictions and still supports something like a move [27].

However, the *move* has two major problems for which we need to understand some internals about Fibonacci heaps. A Fibonacci heap is a forest whose roots are kept in a list called the *root list* of the heap. Each tree maintains the heap property, i.e., the key of a child node is higher or equal to the key of its parent. The key in our case is the weight of the corresponding active edge. Also, a Fibonacci heap usually maintains the minimum key of nodes in the root list to allow queries in constant time. The first problem of the move operation is that the cached minimum of a root list cannot be updated in constant time if the current minimum is moved out of that list. Therefore, we do not maintain the minimum. Instead, the QUERY operation rebuilds the root list, which is a common operation for Fibonacci heaps usually done upon extraction of the minimum, resulting in an amortized $O(\log n)$ running time. The second problem is that moving an internal heap node actually moves the whole subtree rooted at this node. Descendants of the node are displaced into the wrong

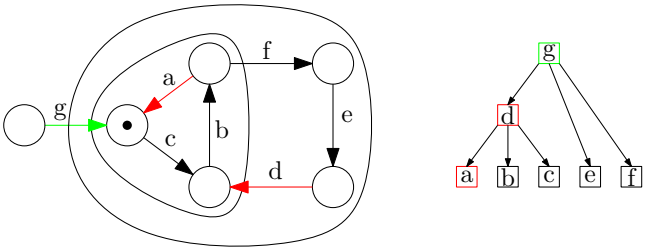

Figure 2: Example graph (left) and corresponding reconstruction forest (right). Assume Gabow starts at the vertex marked with a dot and the edges are labeled alphabetically in the order they were added to the growth path. The first step of the reconstruction process is indicated by colors.

heap and, moreover, changing the key of the moved node can violate the heap property. To fix the displacement, every time a Fibonacci heap operation would put a node into the root list they are returned to the root list of the heap the node actually belongs to, which we call the *home heap* of that node. That is, the *home heap* of a heap node is the heap of the target vertex of the corresponding active edge. Finding the home heap requires a DSU lookup because the target vertex might be contained in a contracted vertex. Gabow et al. [12] prove the following three invariants to address the violated heap property and the correctness of the home heap fix. (1) The root of any tree is always in its home heap. (2) The heaps maintain an additional heap property w.r.t. their home heaps ordered by the position in the growth path. That is, the home heap of a parent node is at least as close to the growth path head as the home heaps of its children. (3) The original heap property is never violated between two nodes that are in their home heap. Only displaced nodes can temporarily violate the heap property.

Time Complexity. The growth path is extended at most n times. Since contracting a cycle of length l reduces the total number of vertices in the graph by $l-1$, there are at most $n-1$ contractions and the summed length of all contracted cycles is less than $2n$. Thus, QUERY, DELETE, and MERGE are called $O(n)$ times on the active forest. Furthermore, the following operations happen at most once per edge and can all be done in constant time. Insertion into an exit list, the active forest, or passive list, REPLACE in the active forest, deletion from an exit list, and deletion from a passive list. The DSU imposes no additional overhead since, if there are at least $n \log n$ calls to $find$, each individual one takes amortized constant time [34]. In total this yields a running time of $O(n \log n + m)$.

2.4 Arborescence Reconstruction Although Tarjan [35] proposed to split off the reconstruction phase from main algorithm, the reconstruction method given in the paper is incorrect. A note by Camerini et al. outlines a working method [4]. Consider a new graph called the *reconstruction forest* where the nodes are the edges that are picked by the arborescence algorithm. In the forest, an edge that was picked as an incoming edge to a contracted vertex has directed arcs to the edges that constitute the top-level cycle of the contracted vertex. A leaf in the reconstruction forest corresponds to the first picked incoming edge of a vertex of the original graph. Figure 2 shows an example. The reconstruction process repeatedly selects a root of the forest. The corresponding edge becomes part of the final solution. The target vertex in the original graph of the selected edge has an associated leaf in the reconstruction forest. The process deletes the path from this leaf to the selected root from the forest, then proceeds with the next root.

A very concise implementation is possible by noting that the order in which the main algorithm picks edges is a reverse topological order of the reconstruction forest. Thus, all roots are found by iterating over the picked edges in reverse and skipping already deleted ones. Further required information are the leaf of each original vertex and the parent for each node in the reconstruction forest. The former is computed by iterating over the picked edges to find the first occurrence of each target. The latter must be saved by the main algorithm each time it contracts a cycle.

3 Implementation

This section introduces different solvers for the minimum arborescence problem, highlights their key points as well as optimizations and deviations from the abstract description in Section 2. We compare 11 solvers; our five versions of Tarjan's approach using different data structures, five external Tarjan-based solvers, and our Gabow implementation (see Table 1). External solvers fall into two categories, coding competition code and library solvers.

Competition Codes. Coding competitions occasionally feature arborescence tasks which require an efficient implementation for sparse graphs. These implementations are often not as maintainable or usable as library solvers, but written with a high focus on performance. The online judge platform Library Checker[2] contains a test set for the minimum arborescence problem. We include the jury solution by the maintainer Kohei Morita as well as the fastest submission by David Stangl. We denote them by YOSUPO[3] and FELERIUS[4] according to their pseudonyms on popular contest websites. Furthermore, there is a competition-style implementation by Takanori Maehara[5], which we denote by SPAGHETTI. With around 130 lines of code, it is the most concise implementation. However, it lacks the reconstruction phase and proper memory management.

Library Solvers. The two library implementations we consider are LEMON and ATOFIGH, both running in $O(n^2)$. The former is part of the LEMON library for graph algorithms[6]. We use the latest release 1.3.1 from 2014. They save incoming edges in arrays. The merge is done by iterating over all incoming edge lists of cycle vertices while collecting the cheapest edge into the cycle for each origin. They reuse the same collecting array each time and clear the used entries afterwards, such that the merge is not $O(n)$ but linear in the number of merged edges. Thus, the solver is faster the less edges are involved in each contraction. The second library implementation, ATOFIGH, was written by Ali Tofigh and Erik Sjölund[7] using the Boost Graph Library [30]. They also represent incoming edge sets in dynamically growing arrays. However, the arrays are sorted by origin vertex and the merge is done with the linear time merge routine usually known from merge sort. It was modified to remove multi-edges by only keeping the cheapest one for each origin. The same performance considerations apply. Note that there exist sparse networks where these merge strategies yield quadratic running time.

Our Tarjan-based Solvers. Our Tarjan code shares the logic for the algorithm and reconstruction and differs only in the data structure to manage the sets of incoming edges (see Section 2.2). The MATRIX solver maintains an adjacency matrix and performs the operations in linear time. The HOLLOW and TREAP solvers use our implementations of Hollow heaps [17] and Treaps [31], respectively, which both support lazy propagation to update weights. The hollow heap is not required to implement the usual decrease key operation, as it is not required by the algorithm, which allows for implementing the merge operation efficiently, by simplifying some bookkeeping tasks. The SET and PQ variants use the `std::set` and `std::priority_queue` data structures from the C++ standard template library. They are typically implemented as a red-black tree [16] and binary heap [6], respectively. Since the set and priority queue interface do not support a fast merge operation, we use the well known *smaller into larger*

[2] `https://judge.yosupo.jp/problem/directedmst`

[3] `https://codeforces.com/profile/yosupo`

[4] `https://codeforces.com/profile/Felerius`

[5] `https://github.com/spaghetti-source/algorithm`

[6] `https://lemon.cs.elte.hu/trac/lemon`

[7] `https://github.com/atofigh/edmonds-alg`

Table 1: Overview of arborescence algorithms. Tarjan+Path means they implement Tarjan's version but adjust the order in which vertices are processed to form a path as in Gabow's version.

Solver	Author/Source	Variant	Data Structure	Runtime
FELERIUS	David Stangl	Tarjan	Skew Heap [32]	$O(m \log n)$
SPAGHETTI	Takanori Maehara	Tarjan+Path	Skew Heap [32]	$O(m \log n)$
YOSUPO	Kohei Morita	Tarjan+Path	Pairing Heap [10]	$O(m \log n)$
LEMON	LEMON 1.3.1	Tarjan+Path	Adjacency List	$O(n^2)$
ATOFIGH	Ali Tofigh	Tarjan	Adjacency List	$O(n^2)$
MATRIX	this paper	Tarjan	Adjacency Matrix	$O(n^2)$
TREAP	this paper	Tarjan	Treap [31]	$O(m \log n)$
HOLLOW	this paper	Tarjan	Hollow Heap [17]	$O(m \log n)$
SET	this paper	Tarjan	Red Black Tree [16]	$O(m \log^2 n)$
PQ	this paper	Tarjan	Binary Heap [6]	$O(m \log^2 n)$
GGST	this paper	GGST	Fibonacci Heap [11]	$O(n \log n + m)$

technique. That is, for a merge we iterate over the smaller of the two sets and add the elements individually to the larger set. An element switches sets at most $O(\log n)$ times, each time into a set that is at least twice as large, and a switch takes $O(\log n)$. This sums up to $O(m \log^2 n)$ for all merges combined. Since the elements are moved individually, weight updates do not need lazy propagation but are handled by an offset for each set that is applied when an element enters or leaves the set.

Our GGST Solver. The solver features three optimizations compared to the description in Section 2.3. First, no dummy edges are inserted. Instead a new path is started each time the root is reached. Second, we replace linked lists by dynamic arrays where possible. Exit lists, passive lists, and the growth path are only modified at the front, so an array can be used by saving them in reverse. Actually, the usage of passive lists as previously described requires arbitrary deletions and thus cross references for each edge to the position in the list. Our third optimization is to remove the need for cross references by simplifying the deletion patterns. The only time the algorithm deletes edges is during the contraction of a cycle[8]. Outgoing edges are deleted by clearing complete exit lists and mirroring the deletions across passive lists. Incoming multi-edges are deleted by clearing complete passive lists and mirroring the deletions across exit lists. We modify the two steps to make synchronization between exit and passive lists easier and restrict modifications to the front of the lists.

When outgoing edges of a cycle are deleted, some of these edges are self loops and some point further down the growth path. Instead of mirroring the clearing of the exit lists by deleting corresponding entries from passive lists, we suggest to entirely skip the removal from the passive lists. This, of course, keeps invalid entries in the passive lists. However, a passive list is only read during a contraction to identify multi-edges into the cycle. At this time, the invalid entries point into a prefix of the path but at the time of deletion pointed down the path. Thus, they became self-loops which can be identified and skipped. Since a passive list is cleared after identification of self loops, each invalid entry is seen only once.

We propose to implement the consolidation of multi-edges as follows. For each passive edge into the cycle, compare the first two edges in the exit list of the origin of the passive edge and delete the more expensive one. This "delete one of the first two edges" operation is done for each origin as often as this origin has passive edges into the cycle. Since this origin's exit list starts with an active edge pointing into the cycle, followed by all the passive edges into the cycle, the cheapest edge of this prefix will remain at the front of the exit list. Gabow et al. propose a similar strategy but delete either the first edge or the currently inspected passive edge (instead of the second in the exit list), which requires for each passive edge a way to obtain its handle in the exit list.

4 Experiments

In this section we evaluate the solvers listed in Table 1. The solvers, data preparation scripts, plotting code, execution logs, and the raw timing data are available in our public repository.

Setup. The experiments were performed on a server with two 8-Core Intel Xeon™ Gold 6144 CPUs and 192 GB DDR4 memory on the openSUSE Leap 15.3 operating system. The implementations are written in C++ and adjusted to fit a common interface. The code was compiled with gcc version 10.3.0. Each run had a

[8]The original description by Gabow et al. has more deletions. We simplified the algorithm in this regard.

timeout of 30 minutes. We used a total of 656 networks from the following sources. The number of networks is in parenthesis.

- **konect** (319). All directed networks smaller than 5GB from the KONECT project[9].
- **networkrepository** (75). A selection of sparse networks from the Network Repository project[10]. The project contains mostly undirected networks and does not label directed ones as such. We downloaded all networks (around 3000) and kept the ones that are labeled as directed in their respective file format.
- **girgs** (200). This data set contains Geometric Inhomogeneous Random Graphs (GIRGs), a generative network model closely related to hyperbolic random graphs [3, 24]. We used the efficient generator by Bläsius et al. [1] with default parameters except for n, deg, and seeds. We set $n = 10^4$ and average degrees from 50 to 2000 in steps of 100 with 10 networks per configuration. Edges are directed randomly.
- **antilemon** (5). A sparse family of networks crafted to be difficult for arboresence solvers. They require at least $n/2$ contractions with at least $n/2$ edges pointing into each contracted cycle. We generated networks with $n = 10^i$ for $i \in [2, 6]$.
- **fastestspeedrun** (47). Test cases of a programming task from the ICPC Northwestern Europe Regional Contest 2018[11]. They have up to 2500 vertices and are fully connected.
- **yosupo** (10). Test cases for the Directed MST problem on the Library Checker website. The networks are Erdős-Rényi graphs [8] with a random spanning tree from the root vertex as subgraph. Weights are sampled uniformly at random.

For networks without weights, we sample integer weights uniformly at random. If an instance has no specified root, we restrict us to the largest connected component, and add a root vertex that connects to all original vertices with edges of weight infinity.

General Performance. Figure 3 shows all instances with an untied fastest solver, i.e., a solver that is strictly faster than all others. The major reason for ties is that two or more solvers are faster than 1 ms which is the precision of our measurements. Over all,

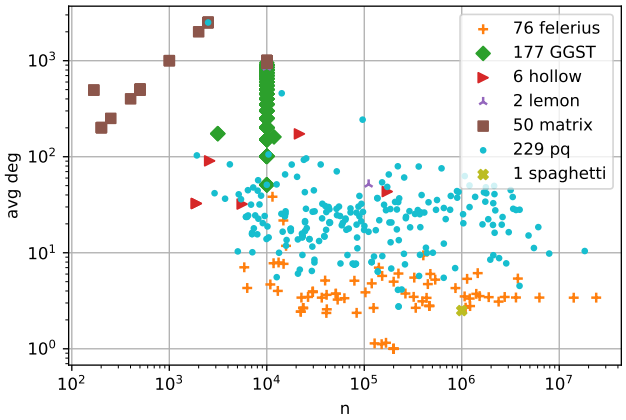

Figure 3: For each instance the untied fastest solver if any. The legend includes the number of wins per solver.

115 instances are tied, 85 instances have at least two algorithms that solve the instance faster than 1 ms, 73 instances are solved in under 1 ms by at least six solvers, and 46 instances are solved in under 1 ms by all solvers.

On the untied instances, the PQ solver dominates with 229 wins, followed by GGST with 177, then FELERIUS with 76, and MATRIX with 50. Combined, these four solvers win more than 98 % of the untied instances. Moreover, there is a clear trend regarding the type of instance each solver is good at mirroring the theoretical complexities of the algorithms quite closely. The matrix-based Tarjan solver, MATRIX, is best for dense graphs, the heap-based Tarjan solvers, PQ and FELERIUS, are optimal for sparse graphs, and the GGST algorithm wins in between. For the sparse real-world instances, there is a clear cut between the PQ and FELERIUS solvers. The FELERIUS solver was specifically tuned to be fast on the yosupo instances which have barely more edges than vertices and thus wins on instances with average degree below 10. Furthermore, all but three of the 177 GGST wins are on GIRGs. We explicitly generated the GIRGs to fill the gap between the sparse real-world networks and the fully connected fastestspeedrun instances. The most surprising result, however, is that the PQ solver using a binary heap performs exceptionally well although it should scale worse in the number of edges than the competitors by at least a logarithmic factor due to the missing merge operation. We identify three possible reasons for this behavior. First, a binary heap implementation is very efficiently while the more complex logic of GGST and less cache efficient data structures of FELERIUS cause significant overhead. Second, realistic data is easy in the sense that the contractions, which are the theoretical bottleneck of the PQ solvers, occur not as often or involve less edges and

[9] http://konect.cc/
[10] https://networkrepository.com
[11] https://2018.nwerc.eu/

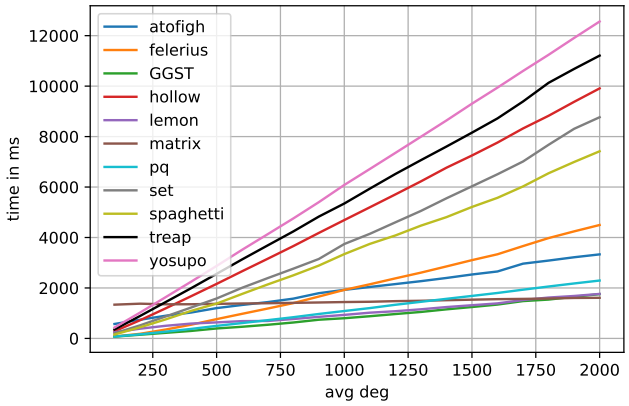

Figure 4: The run time of the solvers on GIRGs with 10^4 vertices over growing density. Each data point is averaged over 10 GIRGs with the same density.

vertices. Finally, realistic networks are sparse and thus $O(n \log n)$ becomes indistinguishable from $O(m \log n)$, which is the remaining complexity of the binary heap implementation when ignoring the cost for contractions. Therefore, on sparse networks with few contractions, the three solvers GGST, FELERIUS, and PQ all have a complexity of roughly $O(n \log n)$ and it comes down to implementation details like memory layout, cache efficiency, and the level of code optimization. For the same reason MATRIX beats GGST on very dense instances where both solvers have a complexity of $O(n^2)$.

Scaling Analysis. To examine the effect of density on solver performance we use the girgs data set. The GIRG model produces realistic networks regarding degree distribution, clustering, and distances that resemble the real-world networks from the networkrepository and konect data sets. Figure 4 shows the results. As expected, the MATRIX solver is not affected by the number of edges. It starts out as the slowest solver but beats all the others by the time the degree reaches 2000. All other solvers exhibit an approximately linear scaling in the number of edges which emphasizes again that logarithmic factors are hardly noticeable for reasonably sized inputs. Most notably, this includes the $O(n^2)$ ATOFIGH and LEMON solvers. These solvers heavily depend on the fact that the instance structure is easy and needs few contractions involving few edges. The LEMON solver is the second fastest solver only slightly outperformed by GGST indicating that GIRGs are even easier to solve than the real-world networks from the other data sets. The reason for this could be the randomized edge direction for the GIRGs. Another interesting fact is that the five solvers that scale the worst with growing density are YOSUPO, TREAP, HOLLOW, SPAGHETTI, and SET. These five have in common that they use pointer-based heap data structures to manage the edges. The other solvers use indices into a preallocated pool (FELERIUS), don't have a heap element for every edge (GGST), or don't use a heap to manage edges (LEMON, ATOFIGH).

Time Per Operation. Figure 5 shows the run times of the solvers divided into initialization, execution, reconstruction, and destructor subroutines as well as timeouts. The ATOFIGH solver crashed by exceeding the available memory on the largest antilemon graph and 8 road networks from the konect data set, which are originally from the 9th DIMACS Implementation Challenge on shortest paths. These crashes are treated as timeout. The MATRIX solver is only executed on graphs with less than 10^5 vertices, and treated as timeout otherwise. The LEMON solver timed out on three DIMACS graphs and the largest antilemon graph.

On the real-world networks from the konect and networkrepository data sets, the quadratic solvers perform much worse than the other algorithms. Of course MATRIX cannot handle large graphs but also ATOFIGH and LEMON occasionally encounter a difficult instance that overshadows their good performance on the many easy instances since we only consider the summed run time here. The LEMON solver performs well on the girg data set and the MATRIX solver dominates the fully connected fastest-speedrun instances. Otherwise the quadratic solvers are never among the fastest. In particular, these three solvers are more than two orders of magnitude slower than the FELERIUS solver on the networkrepository graphs where the FELERIUS solver is the fastest on all instances. The antilemon instances were crafted as worst case instances for LEMON and ATOFIGH which is clearly visible in the results. On this data set, the PQ solver outperforms the others. Unsurprisingly, the FELERIUS solver performs best on the yosupo data which it was optimized for.

Before evaluating the run times of individual subroutines, we note that LEMON and ATOFIGH perform most of the initialization and reconstruction operations in the main phase of the algorithm while SPAGHETTI has neither reconstruction nor memory management. With that in mind, our experiments show that the reconstruction phase takes only a fraction of the run time independent of solver or data set. Furthermore, the initialization, which includes allocating memory and building internal data structures, takes a considerable amount of time for all algorithms. The high initialization time can be explained by the fact that just inserting the edges into the heap data structures takes $O(m \log n)$ and as such is one of the theoretical bottlenecks of most implementations. There exist linear time constructions for some of the data structures (e.g. treaps, binary heaps, skew heaps) but for consistency across solvers, we build them by repeated in-

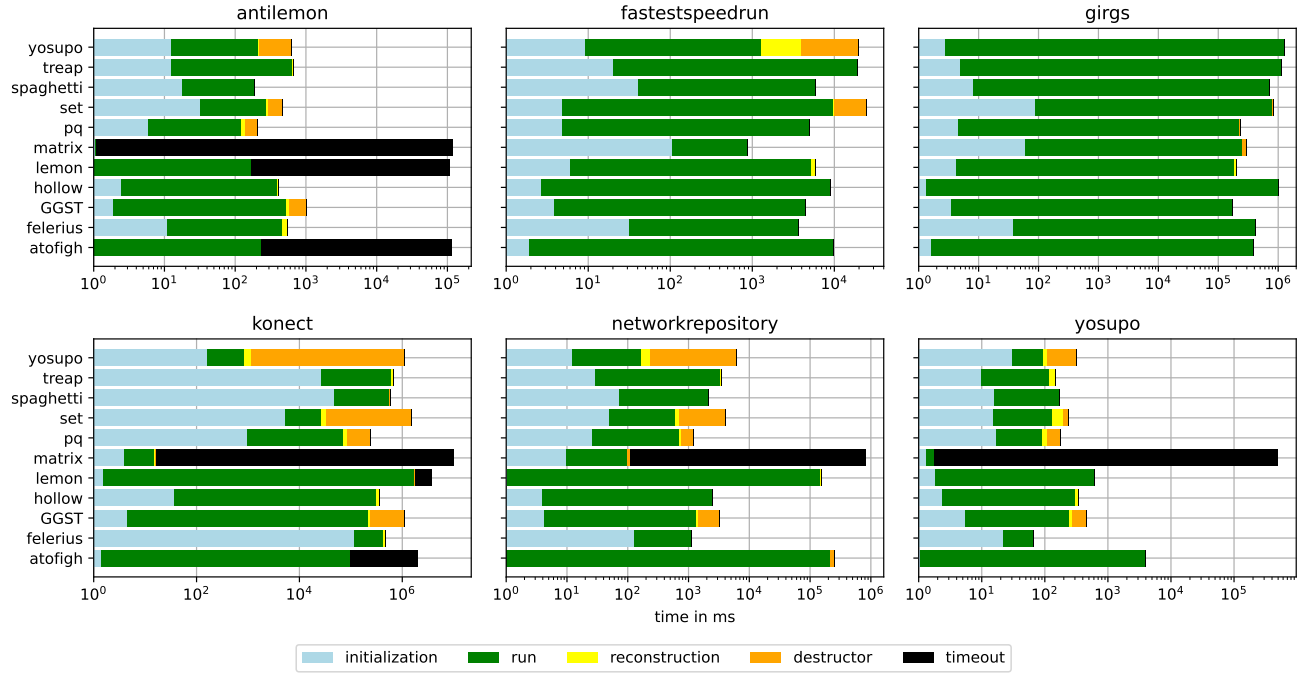

Figure 5: For each data set the summed run time over the contained instances per algorithm. The bars are divided into colored segments to show the fraction of time spend on each subroutine. For timeouts, all 30 minutes are counted as timeout no matter what was done in these 30 minutes. Note that the colored segments inside each bar are completely detached from the logarithmic x-axis.

sertions. Finally, the high destructor time of the YOSUPO solver is due to their use of `std::shared_ptr` instead of manual memory management.

5 Conclusion

In this paper we discussed the Tarjan and GGST versions of Edmonds' algorithm for the minimum spanning arboresence problem. We outlined existing solvers, provide our own implementations, and compare their practical performance. Our implementation of the GGST algorithm is the first public implementation and our description simplifies the original one in several aspects. Our experiments show that the compared solvers perform well on real-world data while scaling experiments suggest that realistic networks are substantially easier than worst-case instances. Even solvers with an $O(n^2)$ worst-case complexity often perform almost linear in the number of edges. However, they are not as consistent. They time out when the instance contains difficult structures, which occasionally happens even on real-world networks. Furthermore, we find that differences in complexity by logarithmic factors are mostly irrelevant in practice. Our $O(m \log^2 n)$ Tarjan implementation using a binary heap beats the other solvers on most real-world networks although our $O(n \log n + m)$ GGST implementation is two logarithmic factors faster asymptotically. This, again, emphasizes that real-world instances often do not force the worst case of an algorithm and complex logic and data structures can produce significant overhead. For future work, it would be interesting to examine what makes realistic instances easy and possibly show a better running time on a random model like hyperbolic random graphs similar to the result for Erdős-Rényi graphs.

References

[1] T. Bläsius, T. Friedrich, M. Katzmann, U. Meyer, M. Penschuck, and C. Weyand, *Efficiently Generating Geometric Inhomogeneous and Hyperbolic Random Graphs*, in Proceedings of the Annual European Symposium on Algorithms (ESA), 2019.

[2] F. C. Bock, *An algorithm to construct a minimum directed spanning tree in a directed network*, Developments in Operations Research, (1971).

[3] K. Bringmann, R. Keusch, and J. Lengler, *Geometric inhomogeneous random graphs*, Theoretical Computer Science, 760 (2019).

[4] P. M. Camerini, L. Fratta, and F. Maffioli, *A note on finding optimum branchings*, Networks, 9 (1979), pp. 309–312.

[5] Y.-J. Chu, *On the shortest arborescence of a directed graph*, Scientia Sinica, 14 (1965).
[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, MIT press, 2009.
[7] J. Edmonds, *Optimum branchings*, Journal of Research of the National Bureau of Standards, 71B (1967), p. 233.
[8] P. Erdős and A. Rényi, *On random graphs, i*, Publicationes Mathematicae (Debrecen), 6 (1959).
[9] O. Fischer and R. Oshman, *A Distributed Algorithm for Directed Minimum-Weight Spanning Tree*, in 33rd International Symposium on Distributed Computing (DISC 2019), vol. 146 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2019, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 16:1–16:16.
[10] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, *The pairing heap: A new form of self-adjusting heap*, Algorithmica, 1 (1986).
[11] M. L. Fredman and R. E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM, 34 (1987).
[12] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (1986).
[13] B. A. Galler and M. J. Fisher, *An improved equivalence algorithm*, Communications of the ACM, 7 (1964).
[14] L. Georgiadis, *Arborescence optimization problems solvable by edmonds' algorithm*, Theoretical Computer Science, 301 (2003).
[15] R. L. Graham and P. Hell, *On the history of the minimum spanning tree problem*, IEEE Annals of the History of Computing, 7 (1985).
[16] L. J. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, in Proceedings of the Annual Symposium on Foundations of Computer Science (SFCS), 1978.
[17] T. D. Hansen, H. Kaplan, R. E. Tarjan, and U. Zwick, *Hollow heaps*, Transactions on Algorithms, 13 (2017).
[18] ISO, *ISO/IEC 14882:2020 Programming languages — C++*, International Organization for Standardization, sixth ed., 2020.
[19] V. Jarník, *O jistém problému minimálním [about a certain minimal problem]*, Práce moravské přírodovědecké společnosti, 4 (1930).
[20] T. Jombart, R. M. Eggo, P. J. Dodd, and F. Balloux, *Reconstructing disease outbreaks from genetic data: a graph approach*, Heredity, 106 (2010).
[21] N. Kamiyama, *Arborescence problems in directed graphs: Theorems and algorithms*, Interdisciplinary Information Sciences, 20 (2014).
[22] N. Kamiyama, N. Katoh, and A. Takizawa, *Arc-disjoint in-trees in directed graphs*, Combinatorica, 29 (2009).
[23] R. M. Karp, *A simple derivation of edmonds' algorithm for optimum branchings*, Networks, 1 (1971).
[24] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá, *Hyperbolic geometry of complex networks*, Physical Review E, 82 (2010).
[25] J. B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proceedings of the American Mathematical Society, 7 (1956).
[26] L. Lovasz, *Computing ears and branchings in parallel*, in 26th Annual Symposium on Foundations of Computer Science (sfcs 1985), 1985, pp. 464–467.
[27] R. Mendelson, R. E. Tarjan, M. Thorup, and U. Zwick, *Melding priority queues*, Transactions on Algorithms, 2 (2006).
[28] R. C. Prim, *Shortest connection networks and some generalizations*, Bell System Technical Journal, 36 (1957).
[29] Y. V. Salii and A. S. Sheka, *Improving dynamic programming for travelling salesman with precedence constraints: parallel morin–marsten bounding*, Optimization Methods and Software, (2020), pp. 1–27.
[30] B. Schäling, *The Boost C++ libraries*, 2011.
[31] R. Seidel and C. R. Aragon, *Randomized search trees*, Algorithmica, 16 (1996).
[32] D. D. Sleator and R. E. Tarjan, *Self-adjusting heaps*, Journal on Computing, 15 (1986).
[33] M. Suk and O. Song, *Curvilinear feature extraction using minimum spanning trees*, Computer Vision, Graphics, and Image Processing, 26 (1984).
[34] R. E. Tarjan, *Efficiency of a good but not linear set union algorithm*, Journal of the ACM, 22 (1975).
[35] ———, *Finding optimum branchings*, Networks, 7 (1977).

A Appendix

In this section, we briefly describe some further details on the external solvers.

Integration Issues and Bug Fixes. The LEMON solver does not compile with C++20 upwards because it uses allocator methods that were deprecated in C++17 and removed in C++20. It performs reconstruction during the main algorithm. In Figure 5, its reconstruction time is the time to obtain the solution from their internal data structures.

The ATOFIGH solver contains a programming error in a radix sort subroutine where a right shift equal to the size of the left hand operand type (`int` in our template instantiation) is performed. The C++ standard[12] states in Section 7.6.7 concerning shift operators "The behavior is undefined if the right operand is negative, or greater than or equal to the width of the promoted left operand" [18]. Most compilers give a warning (if enabled) and default to 0, which actually works with the given implementation. Nevertheless, we fixed this error by changing $\leq$ to $<$ in the loop that iterates over the radix. The atofigh solver performs reconstruction during the main algorithm. In Figure 5, its reconstruction time is the time to obtain the solution from their internal data structures.

The YOSUPO solver uses `std::shared_ptr` for memory management. On very large instances this crashes due to a stack overflow caused by deep recursion in the destructor. On Linux machines, one can increase the stack limit to circumvent this problem, which is what we do in our experiments.

The SPAGHETTI solver does not free allocated memory which gives it an advantage over the other solvers. We decided to keep the leak since a proper cleanup would require considerable changes to their code. The SPAGHETTI solver is the only solver that does not support reconstruction.

Alternative Reconstruction by Stangl. The FELERIUS solver features an alternative method for reconstruction more closely related to the original idea of Edmonds' algorithm. Recall that Edmonds' algorithm contracts each cycle C and when picking an incoming edge into the contracted vertex, it replaces one of the cycle edges. That is, the edge into the contracted vertex corresponds to an original edge (u, v) and replaces the cycle edge incoming to v. The difficulty when adapting this to Tarjan's version is that endpoint indices of edges are not explicitly updated after each contraction. Thus, one has to deal with the possibility that the cycle vertices are contracted vertices representing previous cycles. In this case, v might be contained in a cycle vertex $v' \in C$ rather than being part of the cycle itself. This is, e.g., the case in Figure 2 where the edge g replaces the edge d. Stangl tackles this challenge as follows. Since Tarjan maintains an incoming edge for each vertex during the main algorithm, the reconstruction phase processes the cycles from last to first and performs the necessary replacements. When a cycle is processed, the edge (u, v) that was picked as incoming for this cycle can be found as the incoming edge to the vertex representing the cycle. To find the cycle edge it should replace, a persistent DSU is used to query the cycle vertex v' containing v at the time just before the cycle was contracted. To make the DSU persistent, Stangl drops path compression [34] from the data structure which means each *find* call takes $O(\log n)$. However, the main algorithm as well as the reconstruction perform only $O(n)$ *find* calls thus leaving the total running time unchanged.

Another issue during implementation is that, after contracting a cycle, it is represented by one of its cycle vertices. The representative is chosen by the DSU among the cycle vertices according to the union-by-size strategy [34]. The picked edge incoming to the contracted vertex thus overrides the cycle edge of this representative. The representative and the edge that was (mistakenly) replaced are saved during the main algorithm and restored in reconstruction just before the actual edge is determined that should be replaced.

[12] The standard must be purchased but a working draft is available at `http://www.open-std.org`