Performance Comparison of Minimum Spanning Tree Algorithms

Graph Algorithms Course Project
Faculty of Information Technology, Brno University of Technology

Jiří Hrabovský (xhrabo17@vutbr.cz), Jan Vašák (xvasak01@vutbr.cz)

December 13, 2024

Abstract

In this project we explore and compare the performance of various algorithms for finding minimum spanning trees, specifically Prim's, Kruskal's, Boruvka's and Karger-Klein-Tarjan algorithms. We implement all of these algorithms in `c++` and evaluate their performance on graphs of different sizes and densities. We show that Prim's algorithm with Fibonaci heap is the fastest and that despite Karger-Klein-Tarjan having the best complexity in theory, it is heavily outperformed by the other algorithms on reasonably sized inputs.

1 Preliminaries

Here, we introduce some terminology and give an overview of the assessed algorithms. An *undirected weighted graph* is a tuple $G = (V, E, w)$, where V is a finite set of *vertices*, E is a set of *edges* of the form $\{u, v\}$, where $u, v \in V$ s.t. $u \neq v$, and $w\colon E \to \mathbb{N}$ is the *weight function*. We say that $G' = (V', E', w')$ is a *subgraph* of G if $V' \subseteq V$ and $\forall e \in E'\colon e \in E \land w(e) = w'(e)$. A *path* of length n in G is a sequence of vertices $v_1 v_2 \dots v_n$, where $\forall 1 \leq i \leq n-1\colon \{v_i, v_{i+1}\} \in E$. A *tree* is a graph $T = (V_T, E_T, w_T)$, where for each $u, v \in V_T$ there exists a path $v_1 \dots v_n$ in T, where $v_1 = u$ and $v_n = v$, and where $|E_T| = |V_T| - 1$. We will also use the term *forest* to describe a graph consisting of pairwise unconnected trees.

The *minimum spanning tree* (or MST for short) of a graph $G = (V, E, w)$ is a tree $T_{min} = (V, E_{min}, w_{min})$, s.t. T_{min} is a subgraph of G and there is no tree $T' = (V, E', w')$, where T' is a subgraph of G and $\sum_{e \in E'} w'(e) < \sum_{e \in E_{min}} w_{min}(e)$. The MST problem is the task of finding an MST of a given graph. We will also use the term *minimal spanning forest* for an equivalent problem in the case of unconnected graphs.

Given a spanning tree $T = (V, E_T, w_T)$ of a graph $G = (V, E, w)$, a T-heavy edge is an edge $e \in E$, such that $w(e) > \bar{w}_T(e)$, where $\bar{w}_T(\{u, v\})$ is defined as the weight of the heaviest edge on the path from u to v in T. Note that no T-heavy edge can be in an MST of G, because any cut this edge crosses will also be crossed by some edge from the cycle created by inserting the edge into T, so it will never be a light edge.

We will consider a total order on the weight of edges of any given graph, i.e., all edges must have a unique weight. This can be assumed without loss of generality, as we can simply assign each edge a unique number between 1 and $|E|$, and use this to break any tie on weight.

1.1 Overview of Assessed Algorithms

Here, we will briefly summarize the algorithms (and their variants) whose performance we evaluated in the experiments. Although most of the algorithms solve the more general problem of finding a minimum spanning forest, for simplicity, we will only consider the case of connected graphs.

Kruskal's Algorithm The algorithm was proposed by Joseph Kruskal in 1965 [10]. The first step is to sort the edges of the input graph by weight. The algorithm then processes the sorted edges, adding each edge to the MST in order, provided that it does not connect two vertices that are already part of the same connected component in the MST. To determine whether two vertices are already connected, the algorithm uses a *disjoint-set* data structure. Initially, each vertex belongs to its own set, and adding an edge to the MST involves performing a union operation on the sets containing the two vertices connected by the edge. The worst-case time complexity is $O(|E| \log |V|)$.

Jarník's (Prim's) Algorithm First invented by Vojtěch Jarník in 1930 [7], it is more commonly known as Prim's algorithm, after Robert Prim, who rediscovered it in 1957 [11]. The algorithm builds a tree starting from an arbitrary vertex, and always adds the lightest edge connecting the tree to an unconnected vertex. To find the lightest edge to an unconnected vertex, the algorithm keeps a priority queue of the lightest edges for each connectable vertex. We will consider two different implementations of the priority queue:

- Binary heap (worst-case time complexity $O(|E| \log |V|)$)
- Fibonacci heap (worst-case time complexity $O(|E| + |V| \log |V|)$)

Borůvka's Algorithm The algorithm was developed by Otakar Borůvka in 1926 [3]. We will first describe the so-called *Borůvka step*. For each vertex, color the lightest edge incident to it. Then, merge all vertices that form a component in the colored graph into a *macro-vertex* in a new graph. In the new (smaller) graph, remove all self-loops, isolated vertices, and all but the lightest edge in each set of edges between two vertices. The final algorithm then simply runs Borůvka steps until the graph has no vertices left. The MST is constructed with the set of edges that were colored at any point in the algorithm. The worst-case time complexity of the Borůvka algorithm is $O(|E| \log |V|)$.

Karger-Klein-Tarjan Algorithm The Karger-Klein-Tarjan (KKT) algorithm is a randomized algorithm with a linear expected runtime, designed by Karger, Klein, and Tarjan in 1995 [8]. The algorithm works as follows:

1. Apply two successive Borůvka steps to the graph.
2. In the resulting graph G, choose a subgraph H by removing edges from G, each with probability $\frac{1}{2}$. Apply the KKT algorithm recursively to H, producing a minimum spanning forest F of H. Find all the F-heavy edges in G and remove them from G.
3. Apply the KKT algorithm recursively to the resulting graph, computing a minimum spanning forest F'. Return the edges colored in step 1, along with the edges of F'.

The finding of F-heavy edges in step 2 can be done in linear time using a (rather complicated) MST verification algorithm, originally proposed by Dixon, Rauch, and Tarjan [4], with subsequent simplifications by King [9] and Hagerup [6]. The algorithm verifies MSTs by solving the *tree-path-maxima* problem. Given a tree $T = (V_T, E_T, w_T)$ and a list of pairs $\{u, v\}$, where $u, v \in V_T$, the tree-path-maxima problem is the task of finding $\bar{w}_T(\{u, v\})$, i.e., the heaviest edge on the path between u and v in T. The pairs of vertices given in the input list are called *queries*. It is easy to see that applying the algorithm (in step 2 of the KKT algorithm described above) to calculate $\bar{w}_F(e)$ for each edge e in G, we can decide if e is F-heavy by simply checking if $w_G(e) > \bar{w}_F(e)$ (w_G being the weight function of G).

As it is the most complicated part of the KKT algorithm, we will give a high-level description of how the tree-path-maxima algorithm operates. The algorithm makes use of the fact that it runs on a fully branching tree (a tree whose non-leaf vertices all have at least two descendants, and whose leaf vertices are all in the same depth). We can convert any tree T into a full branching tree by running Borůvka's algorithm on it (which is linear for a tree) and creating a new tree T', where the vertices on each level are the vertices of the graph resulting from each Borůvka step. Specifically, the leafs of T' are the vertices of T, each leaf's parent is the macro-vertex into which the leaf was merged after one Borůvka step and so on. The edge connecting a vertex to its ancestor is the edge incident to the vertex that was colored in the corresponding Borůvka step. It holds that for all queries $\{u, v\}$, $\bar{w}_T(\{u, v\}) = \bar{w}_{T'}(\{u, v\})$ [9, Theorem 1].

This means that all the queries we are interested in are between leaf vertices. We will use $LCA(u, v)$ to denote the *lowest common ancestor* of the nodes u, v (see [2] for an algorithm that computes all LCAs in linear time). We use this to split each query $\{u, v\}$ into two *ancestor-descendant* queries $\{u, LCA(u, v)\}$ and $\{LCA(u, v), v\}$ (we will also refer to the ancestor vertex as the *upper* vertex, and the descendant vertex as the *lower* vertex). The queries of each leaf are the queries it is the lower vertex of.

In the main algorithm, starting from the leafs, each vertex propagates its queries to its direct ancestor, provided that the ancestor is not the upper vertex of the query. I.e., the queries are distributed along the bottom end vertices of each potential heaviest edge. Then, the algorithm works top-down, each vertex propagating the current best answers to queries down to its descendants. More specifically, in any given vertex, the current best answer to a query is either the answer propagated from an ancestor or the edge linking the vertex to its direct ancestor.

To keep the linear-time requirement, the sets of queries and answers in each vertex are implemented as bit words, the propagation of answers to descendants as a bit-wise operation, and determining which answers should be replaced by the currently processed edge utilizes a binary search. For a more detailed explanation of the tree-path-maxima algorithm, we refer the interested reader to [13] and [6].

2 Implementation

As our aim was to evaluate the performance of the algorithms, we chose the `C++` language for our implementation of them. We used the `Boost` library's graph component's undirected adjecency list graph, as well as its implementations of the Fibonacci heap. We also used μt [5] from the `Boost-ext` libraries for unit tests and `argparse` for parsing command-line arguments. To build the project we used `CMake`, steps to build the project are in the top level `README.md` in the code repository.

The following mst algorithms were implemented for the experiments:

- Prim's algorithm with binary heap (`prim_bin_heap`),
- Prim's algorithm with Fibonacci heap (`prim_fib_heap`),
- Kruskal's algorithm (`kruskal`),
- Boruvka's algorithm (`boruvka`),
- Karger-Klein-Tarjan algorithm (`random_KKT`).

In addition, Prim's algorithm which uses 4-ary heap and Kruskal's algorithm both from the `Boost` library were used in the experiments to verify our implementations.

Finding F-heavy edges in the KKT algorithm requires a linear-time algorithm for mst verification. We used the Hagerup implementation [6], copied the code given in the paper and modified it so that it works in c++ and with our graph representation. The verification algorithm used can only answer queries on max edge between list vertex and it's ancestor, so to transform the original query to this format we need to find the lowest common ancestor of each query pair, we achieved this with Bender's algorithm [2], using the implementation from [1] and modified it to work on our graphs.

3 Experimental Setup

All experiments were carried out on a machine with the specifications shown in Table 1.

Graph Generation We generated graphs of different sizes (vertex counts) and densities. As we only considered connected graphs, we started by generating a random spanning tree of a given size. This was done by selecting a random vertex u connected to the tree and a random vertex v not connected to the tree and adding an edge $\{u, v\}$ to the tree. After the spanning tree was constructed, edges were randomly added until the desired density was achieved (with each non-existing edge having the same probability to be selected at each step). We created graphs of sizes $100, 200, \ldots, 1000$, each size with six different variants based on density. The chosen densities were '0' (i.e., the generated graph is exactly the spanning tree), 0.2, 0.4, 0.6, 0.8, and 1 (i.e., a complete graph).

For measuring time, we used `std::chrono::steady_clock` and the runtime was computed as an average of 10 runs.

Table 1: System Specifications

OS	Ubuntu 22.04.3 LTS
Number of CPUs	4
CPU model	Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
L1d	128 KiB (4 instances)
L1i	128 KiB (4 instances)
L2	1 MiB (4 instances)
L3	6 MiB (1 instance)
RAM	24 GiB

4 Experimental Results

The general order of performance from slowest to fastest was `random_KKT`, `boruvka`, `prim_bin_heap`, `kruskal`, `prim_fib_heap`. This can be seen in almost all the plots shown in Figure 2. For sparse graphs, the `random_KKT` and `boruvka` algorithms are significantly slower compared to the others. However, their relative performance improves as the graph density increases. In contrast, `prim_bin_heap` becomes less efficient with denser graphs and performs on par with `boruvka` for complete graphs. Both of these observations can also be seen in Figure 1, which depicts the scaling behavior of each algorithm as the density of the graph increases for graphs with 1,200 vertices.

The best performing algorithm was `prim_fib_heap` by a considerable margin. It is the fastest for all densities except for graphs that contain only spanning trees, where it is not far behind (especially when considering the difference in absolute numbers). The `kruskal` algorithm also performs very consistently, although it is a bit slower than `prim_fib_heap`.

5 Conclusion

The experimental results show the following behavior for each algorithm:

1. **KKT:** Although it does appear to behave linearly relative to the number of edges in a graph, the linear coefficient seems to be too large for the algorithm to be useful in practice. We speculate that this is (at least partly) due to the fact that at multiple points, the algorithm relies on methods that need to be heavily optimized to perform in linear time.

2. **Borůvka:** The algorithm is not very effective, but it is surprising that it performs better than the KKT algorithm and is even on par with Prim's algorithm with binary heap in the case of complete graphs.

3. **Prim with binary heap:** Performs well for sparse graphs, but scales poorly with increasing density.

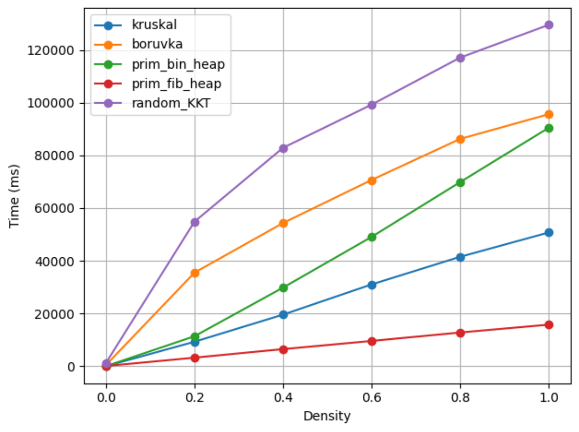
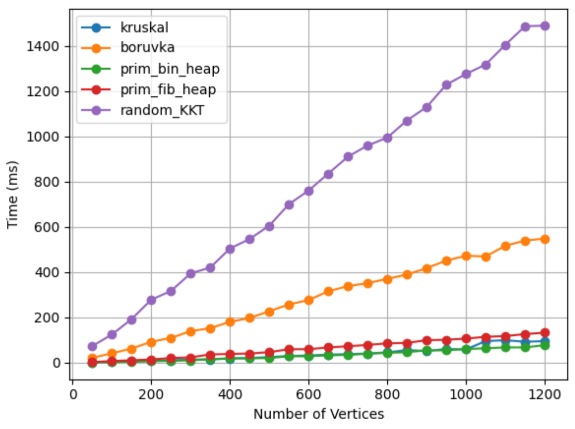
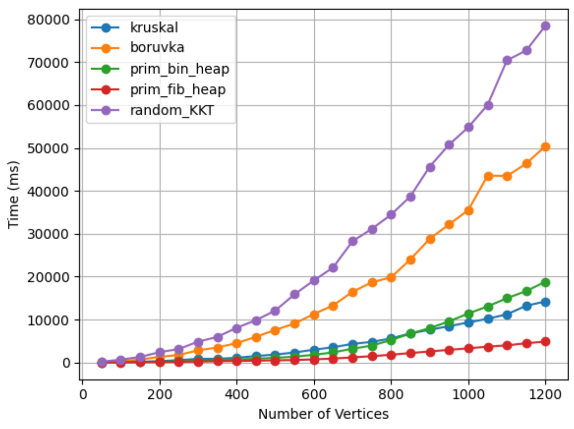


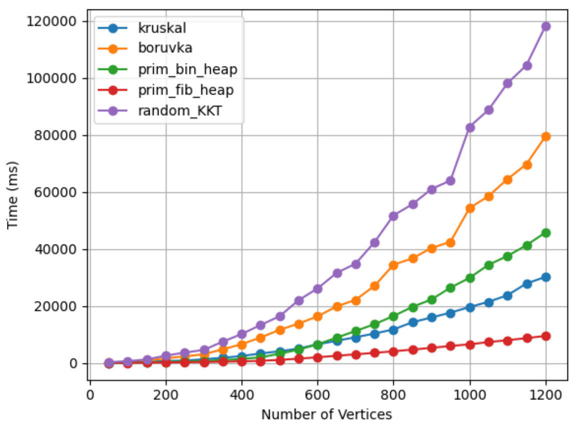
Figure 1: Plot of algorithm runtime against graph density (of a graph with 1200 vertices)



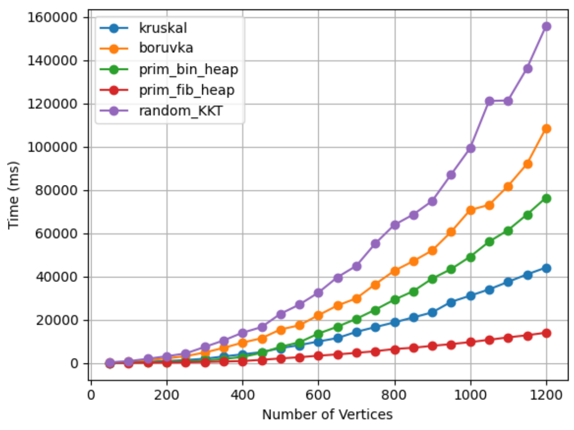
(a) Density ‘0’ (spanning tree only)



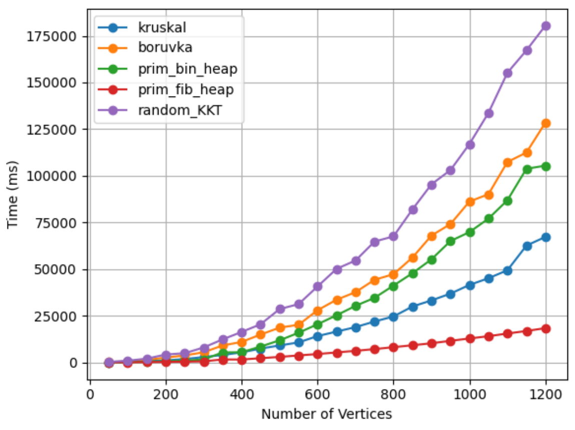
(b) Density 0.2



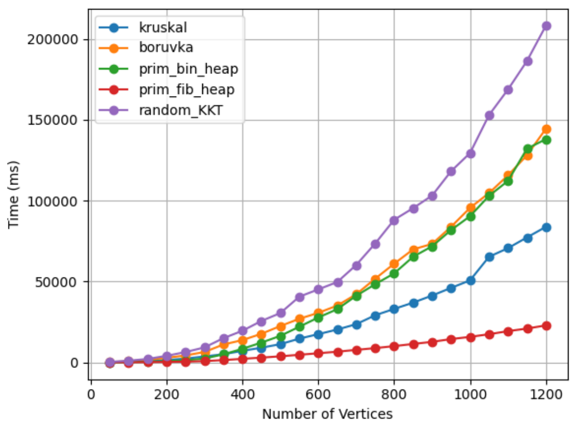
(c) Density 0.4



(d) Density 0.6



(e) Density 0.8



(f) Density 1.0

Figure 2: Plots of algorithm runtime against graph size in vertices for different densities

4. **Kruskal:** Has a good baseline and scales well with increasing densities, making it very consistent across all measured graphs, albeit noticeably slower than Prim's algorithm with Fibonacci heap.

5. **Prim with Fibonacci heap:** The theoretical low complexity provided by the Fibonacci heap proves itself very effective in practice, making this algorithm the fastest on all but the simplest graphs.

Overall Prim's algorithm with the Fibonacci heap performs the best, and, based on our results, there does not appear to be a reason to use the other algorithms when finding MSTs in practice. We also note that the inefficiency of the KKT algorithm could be partly caused by the fact that it is complex to implement, however, our result seems to be consistent with previous research [12].

References

[1] ALGORITHMS cp. *Algorithms for Competitive Programming* [online]. 2024. Available at: https://github.com/cp-algorithms/cp-algorithms. [cit. 2024-12-12].

[2] BENDER, M. A.; FARACH COLTON, M.; PEMMASANI, G.; SKIENA, S. and SUMAZIN, P. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*. USA: Academic Press, Inc., november 2005, vol. 57, no. 2, p. 75–94. ISSN 0196-6774.

[3] BORŮVKA, O. O jistém problému minimálním [On a Certain Problem of Minimization]. *Práce moravské přírodovědecké společnosti*, 1926, vol. 3, no. 3, p. 37–58. Zbl JFM 57.1343.06.

[4] DIXON, B.; RAUCH, M. and TARJAN, R. E. Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time. *SIAM J. Comput.*, 1992, vol. 21, no. 6, p. 1184–1192. Available at: https://doi.org/10.1137/0221070.

[5] EXT boost. *C++20 μ(micro)/Unit Testing Framework* [online]. 2024. Available at: https://github.com/boost-ext/ut. [cit. 2024-12-12].

[6] HAGERUP, T. An Even Simpler Linear-Time Algorithm for Verifying Minimum Spanning Trees. In: PAUL, C. and HABIB, M., ed. *Graph-Theoretic Concepts in Computer Science*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 178–189. ISBN 978-3-642-11409-0.

[7] JARNÍK, V. O jistém problému minimálním. (Z dopisu panu O. Borůvkovi) [On a Certain Problem of Minimization (From a Letter to O. Borůvka)]. *Práce moravské přírodovědecké společnosti*, 1930, vol. 6, no. 4, p. 57–63. Zbl JFM 57.1343.07.

[8] KARGER, D. R.; KLEIN, P. N. and TARJAN, R. E. A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees. *J. ACM*, 1995, vol. 42, no. 2, p. 321–328. Available at: https://doi.org/10.1145/201019.201022.

[9] KING, V. A Simpler Minimum Spanning Tree Verification Algorithm. *Algorithmica*, 1997, vol. 18, no. 2, p. 263–270. Available at: https://doi.org/10.1007/BF02526037.

[10] KRUSKAL, J. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 1956, vol. 7, p. 48–50. Available at: http://dx.doi.org/10.1090/S0002-9939-1956-0078686-7.

[11] PRIM, R. C. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 1957, vol. 36, no. 6, p. 1389–1401.

[12] THIESEN, F. *A C++ implementation for an Expected Linear-Time Minimum Spanning Tree Algorithm* [online]. 2020. Available at: https://github.com/FranciscoThiesen/karger-klein-tarjan. [cit. 2024-12-13].

[13] ZWICK, U. *Verification of Minimum Spanning Trees* [online]. Tel Aviv 69978, Israel: [b.n.], november 2009. Available at: https://www.cs.tau.ac.il/~zwick/grad-algo-0910/mst-verify.pdf. [cit. 2024-12-09]. Lecture notes for "Advanced Graph Algorithms".