Data Structures for the Minimal Spanning Tree Theory and experimental evaluation

Stefano Marchini

Abstract

We conduct an experimental analysis on the impact of different algorithms and heap-based data structures for the minimal spanning tree problem in directed and undirected graphs. We analyze the behavior of Kruskal's and Prim's algorithms in undirected graphs and Edmonds' algorithm in directed graphs (spanning r-arborescences). Our study is based on random graphs varying in size and density. We aim to identify strengths and weaknesses of various approaches in order to guide programmers towards the best settings for their applications.

1 Introduction

An undirected graph $G = (V, E)$ is a set of vertices V paired with a set of edges $E \subseteq \binom{V}{2}$ where $(u, v) \in E$ implies $(v, u) \in E$ for all the edges. A directed graph $G = (N, A)$ is a set of nodes N paired with a set of arcs $A \subseteq N^2$. We use the notations of undirected graphs for general graphs, when directionality is not important. A graph is weighted when it is bundled with a function $w : E \to \mathbb{R}$ that assigns a real value to each edge. The density of a graph is defined to be the ratio between the number of edges $|E|$ and the maximum possible; thus, a dense graph has density near to 1 and a sparse graph has density near to 0, although the definition of near remains vague. A path is any sequence of vertices $(v_0, v_1, \ldots, v_k)$ such that (v_i, v_{i+1}) is an edge. A graph is connected if there is a path from every two vertices.

A fundamental problem in graph theory is to compute the minimum spanning tree. Given an undirected graph $G = (V, E)$, a spanning tree is a subset of edges $T \subseteq E$ with exactly one path for each pair of vertices. An equivalent concept exists in directed graphs and it is called r-arborescence: given a directed graphs $G = (N, A)$ a spanning r-arborescence is subset of arcs $T \subseteq A$ with exactly one path from $r \in A$ to every other arc. An arborescence is a directed rooted tree. When the graph is weighted the minimal spanning tree is a spanning tree that minimize $\sum_{e \in T} w(e)$, and the same holds for the minimal spanning r-arborescences. Minimum spanning trees and arborescences are used both as building blocks for other algorithms (e.g., Christofides' TSP approximation [1]) and for their direct application in several real-life problems (e.g., minimize cable cost in electrical networks).

The most well known algorithms for the minimum spanning tree problem are attributed to Prim [2] and to Kruskal [3] and the most well known algorithm for the minimum spanning r-arborescence has been discovered independently by Chu & Liu [4] and Edmonds [5]. Prim's and Edmonds' algorithms require to use priority queues, an abstract data type that is typically implemented on top of the heap data structure. Kruskal's algorithm requires either a sorting algorithm or a priority queue.

There are several variants of the heap data structure. We present binary heaps, d-ary heaps and Fibonacci heaps and we report an experimental evaluation of their performance in the context of computing minimum spanning trees and minimum spanning r-arborescences. Our goal is to highlight

the importance of choosing the right data structure for this task and to guide the readers towards reasonable choices for their application. We consider random graphs with different sizes and different densities. A brief recap of the algorithms and the data structures is outlined in sections 2 to 5; for an in-dept study, we refer the readers to Cormen *et al.* [6] and Kleinberg and Tardos [7]. Section 6 discusses our implementation of random graphs and section 7 reports the performance evaluation of the algorithms. The algorithms and the data structures discussed in this work are implemented form scratch in C++ and they are publicly available at `https://github.com/smarchini/mst`.

2 Disjoint-set data structure

This beautifully simple and powerful data structure maintains a disjoint set of n points, represented by the integers from 0 to $n-1$, so that we can quickly find if two points are in the same set and union two disjoint sets. Kruskal's and Prim's algorithms use a disjoint-set data structure to establish whether or not two vertices are are connected: if they are in the same set then they are connected. Each set in the data structure is represented by a tree and the whole forest is represented implicitly by a vector *parent*. The forest is kept short at each union with the help of a vector *rank* which counts the children of each node. Each node lazily discover and remember (by shrinking the tree) its topmost parent. The result is a data structure that is extremely efficient both in space and in time. Tarjan [8] proves the amortized[1] cost is nearly constant: a sequence of m *union-set* or *find-set* operations on a disjoint-set with n nodes requires total time of $\mathcal{O}(m\alpha(n))$ in the worst case, where $\alpha(n)$ is the inverse Ackermann function $A(n,n)$. The Ackermann function $A(n,n)$ can be roughly thought as a superexponential function $2^{2^{2^{\cdots}}}$ that repeats the exponentiation for n times; actually, it diverges even faster. For instance,

$$A(0,0) = 1 \quad A(1,1) = 3 \quad A(2,2) = 7 \quad A(3,3) = 61$$

and $A(4,4)$ is an astronomical number whose value is over the number particles in the known universe ($\approx 10^{81}$). For any practical purpose this value scales so slowly that we should consider disjoint-sets to be as convenient as a constant time data structure; theoretically, it diverges superlogarithmically. Moreover, its data is stored compactly in memory so it performs very well and it is simple to implement.

3 Priority queue data structures

In many combinatorial problems it is common to ask for the minimum (or maximum) value in a mutable set. The priority queue is the abstract data-type that deals with this kind of operations and the (binary) heap is a classic implementation. In the following sections we will discuss some heap-based data structure. They support the operations *insert*, *extract minimum* and *merge (two) heaps*. We do not discuss the operation to decrease priority, as we will not need it in our applications.

[1] Amortized analysis studies the time required time to perform a sequence of operations, averaged over all the operations performed. We remark, it guarantees the average performance in the worst case and it differs from average time complexity.

3.1 The binary and the d-ary heap

The binary heap is geometrically represented by a complete binary tree. In practice the tree is implicit in an array (indices starting at 0) of size n, which we navigate through the following functions

$$\text{left-child}(i) = 2i + 1 \qquad \text{right-child}(i) = 2i + 2 \qquad \text{parent}(i) = \lfloor (i-1)/2 \rfloor$$

with the property that the key of each node is either $\leq$ (min-heap) or $\geq$ (max-heap) than the keys of its children. All the priority queues operations take $\mathcal{O}(\lg n)$ time in the worst case and they rely on *heapfy*, a percolation to enforce the heap property. In a min-heap the minimum element is stored at index 0 and to extract it we swap it with the last element, resize the vector and *heapfy-down* the tree.

The most interesting operation is heap construction, which iteratively repeats *hepfy-down* from the bottom to the top of the tree. A binary heap with n nodes has height $\lfloor \lg n \rfloor$, it has at most $\lceil n/2^{h+1} \rceil$ nodes of height h and *heapfy* from a node at height h takes $\mathcal{O}(h)$ time. Hence, the running time of the construction algorithm is bounded from above by[2]

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}(n) \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} < \mathcal{O}(n) \sum_{k=0}^{\infty} \frac{h}{2^h} = \mathcal{O}(n) \cdot \frac{1/2}{(1-1/2)^2} = \mathcal{O}(n) \cdot 2 = \mathcal{O}(n).$$

The d-ary heap is a generalization of the binary heap where internal nodes have d children

$$\text{parent}(i) = \lfloor (i-1)/d \rfloor \qquad k\text{th-child}(i) = i * d + k + 1 \quad \text{for } 0 \leq k < d.$$

Computers are very good at performing short linear scans and d-ary heaps are convenient because we can trade off the the height of the (implicit) tree with (short) linear scans. In our experiments we analyze good values for d. Regardless of the application, some good candidates for the best value of d may be of the size of a cache line (64 bytes) or of the size of a cache memory level (few kilobytes).

3.2 Binomial trees and binomial heaps

We introduce binomial trees and binomial heaps for the sake of better understand how the Fibonacci heap works, which is an overall better data structure than the binomial heap. Binomial trees are so recursively defined: the binomial tree of order zero is a single node, and the one of order k has a root of degree k whose children are binomial trees of orders $k-1, k-2, \ldots, 0$ enumerated in decreasing order. A binomial tree of order n has $\binom{n}{d}$ nodes at depth d, hence the "binomial" in its name. The binomial heap is implemented as a forest of binomial trees such that each of them obeys the heap property and such that there can only be either one or zero binomial trees for each order. Thus, the root of each binomial tree contains the smallest key in the tree and a binomial heap with n nodes consists of at most $\lceil \log n + 1 \rceil$ binomial trees. The number and orders of the trees are uniquely determined by the number of nodes n: each binomial tree corresponds to one digit in the binary representation of n; for example, a binomial heap with $13_{\text{dec}} = 1101_{\text{bin}}$ nodes consists of three binomial trees of orders 3, 2, and 0. A binomial tree of order k has 2^k nodes and height k and it can be constructed by linking together two binomial trees of order $k-1$: attach one of them as the leftmost child of the root of the other. To merge two binomial trees we iteratively link same order trees. Inserting a new element consists in creating a new order zero binomial heap and to merge it inside the other. To extract the minimum we remove a root, we insert the children in the forest and we merge to consolidate the binomial property.

[2]We remark, the geometric series converges as $n \to \infty$ for $|r| < 1$. More precisely $\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}$ for $|r| < 1$ and $a \in \mathbb{R}$.

3.3 The Fibonacci heap

Due to merge all the operations of a binomial heap take $\mathcal{O}(\lg n)$ time. The Fibonacci heap [9] is a lazy implementation of the ideas behind the binomial heap. All operations except extract minimum run in $\mathcal{O}(1)$ amortized time. From a theoretical standpoint it is very good; however, we cannot represent the trees implicitly as we do in the d-ary heap. Fibonacci heaps are harder to implement (we have to deal with pointers and dynamic memory management) and, as we will show, they are much slower in practice. The repeated linking required in binomial heaps is done lazily, as late as possible. To insert a node add a single node tree to the forest. To merge two Fibonacci heaps, just connect their forests. The children of each node are maintained in a circular doubly-linked list where removal and concatenation are constant time operations and where the order of the siblings is arbitrary. We consolidate the binomial properties (i.e., high rank means big tree) only when we extract the minimum. The name "Fibonacci" comes from what happens when we perform the operation of decrease key: a full Fibonacci heap looks like a binomial heap, but if we decrease keys in a way that we cut out as many nodes as possible, the Fibonacci heap of rank k might have only F_k (i.e., the k-th Fibonacci number) nodes. In fact, we will never use decrease key so our implementation is kept simple.

4 Minimum Spanning Tree

In the minimum spanning tree problem we want to minimize the sum of all the edges in the spanning tree. We remark that this is different than minimizing the height of the tree (shortest path problem) and that we want to connect all the vertices and not just a subset (steiner tree problem). A spanning tree has exactly $|V|-1$ edges. If we really want to compute the minimum spanning tree we should test if it even exists (i.e., the graph is connected), otherwise we may be interested in computing the minimum spanning forest: the set of minimum spanning tree for each connected component in the graph. We note that, given an algorithm for the minimum spanning tree, we can trivially extend it to compute the minimum spanning forest with a preprocessing phase where one vertex is connected to every other vertices using (dummy) edges weighted $1 + \max_{e \in E} w(e)$. Nevertheless, some minimum spanning tree algorithm may naturally compute minimum spanning forests without this precomputational step. Another trivial observation is that negative weights do not make the problem harder, because we can compute $w'(e) = w(e) - \min\{w(e) \mid e \in E\}$ and use it as a nonnegative weight function without affecting the result (i.e., the edges in the tree are the same). As a consequence, the maximum spanning tree is find by minimum spanning tree algorithms run on opposite-weights.

A cut (A, B) is a partition of the vertices into two subsets $A \subset V$ and $B \subset V$. The cut-set of the cut (A, B) is $\{(u, v) \in E \mid u \in A,\ v \in B\}$. If $e \in T$ then $T \setminus \{e\}$ is a forest made of two connected components separated by the cut $\{e\}$. We highlight an important property.

Proposition 1. *If the forest F belongs to the minimum spanning tree T and $e \notin F$ then $F \cup \{e\}$ belongs to T if and only if there is a cut C disjoint from F such that $e \in C$ is the minimum cost edge in C.*

Proof. Consider two edges $e, f \in C$ in a cut disjoint from F. The set $T \setminus \{e\} \cup \{f\}$ is a spanning tree, but since T is the minimum then $c(e) \leq c(f)$. If $e \notin T$ then there is a path passing for f that connects the two endpoints of e. Hence, $T \setminus \{f\} \cup \{e\}$ is also a minimum spanning tree. □

4.1 Kruskal's algorithm

Kruskal's algorithm [3] is a greedy algorithm[3] that works by exploiting proposition 1 starting from $T = \emptyset$ and by repeatedly adding to T the leanest edge that will not form a cycle. As we stated before, to detect cycles we use a disjoint-set data structure.

In a classic implementation we start by sorting all the edges by their weight, in ascending order. A well-known lower bound proves sorting n elements must take $\Omega(n \lg n)$ time if the smaller element is get by value comparison. Since $E \subset V^2$ and $\lg x^2 = 2 \lg x$, sorting the edges takes $\mathcal{O}(|E| \lg |V|)$ time and so it is the overall running time of Kruskal's algorithm. If the weights are integers we can aim to do better using radix sort or some other non-comparison based sorting algorithm and achieve a worst-case running time of $\mathcal{O}(|E|\alpha(|V|))$ (see the inverse Ackermann function in section 2). If all the weights are equal, or if the graph is unweighted, we can disregard edge sorting entirely or simply use a graph traversal algorithm. In general scenarios, however, programmers should expect sorting to be the bottleneck of their implementation and they must be careful to not to use a poorly optimized sort procedure. In our experiments we rely on libstdc++'s sort implementation, which currently uses an optimized introsort: a quicksort/heapsort hybrid algorithm which provides both fast average-case performance and asymptotically optimal worst-case running time. The linear time algorithm we use in our study is counting sort, which is particularly convenient for when the weights are in a small range of integers and it is also simple to implement [10, Chapter 5.2].

A major drawback of sorting all the edges is that a dense graph has a quadratic number of edges $|E| \approx |V|^2$ while the spanning tree uses only a linear $|V|-1$ amount them. If the fattest edge is part of the minimum spanning tree then sorting everything from the beginning might be convenient; otherwise, we are likely to sort many more edges than what we actually use. Another approach suggests to implement Kruskal's algorithm using a priority queue to discover the minimum weight edges one at the time. The worst-case running time of this approach is still $\mathcal{O}(|E| \lg |V|)$ with any of the heap-based data structure discussed above, but now the bottleneck is within the loop whose add edges to the spanning tree; therefore, the effective running time of the algorithm is more tightly related to the traits of the minimum spanning tree instead than to the density of the graph.

4.2 Prim's algorithm

Prim's algorithm [2] is a greedy algorithm that exploits proposition 1 starting from the forest $T = \emptyset$ with a vertex and no edges for each tree, and then by repeatedly adding to T the leanest edge that connects distinct trees. At the beginning each vertex but the source has ∞-cost reachability and is maintained in a priority queue. The weights (and the predecessor node) of each node is update as we discover shorter paths to reach them. A less natural but slightly more efficient implementation, used in our code, would start with only the source node in the priority queue and then discover all the other (connected) vertices incrementally; this way, the priority queue is typically kept smaller. The same trick could be done in the priority queue implementation of Kruskal's algorithm but, in that case, we would also need a disjoint-set of size $|E|$ on the edges, making it inconvenient in respect of its natural description. The worst case time complexity of Prim's algorithm is strictly dependent on the implementation of the priority queue: $\mathcal{O}(|E|^2)$ with a list, $\mathcal{O}(|E| \lg |V|)$ with binary heap, $\mathcal{O}(|E| \log_d |V|)$ with a d-ary heap and $O(|E| + |V| \ln |V|)$ with a Fibonacci heap.

[3] Greedy algorithms aim to find a globally optimal solution by repeatedly choosing a locally-optimal move.

5 Minimum Spanning r-arborescence

The minimum spanning r-arborescence problem is the natural generalization of the minimum spanning tree problem in the context of directed graphs. We want to find a (minimum weight) set of edges that makes us able to reach all the nodes, starting from node r. Again, we do not want to minimize the distance between r and every other node (shortest path problem), we want to minimize the sum of all the edges in the arborescence. Similar observations to the minimum spanning tree problem holds: negative weights are not a problem, the algorithms to find the minimum can also find the maximum, and connectivity can be enforced. The problem to find a minimum directed spanning forest is best known as the minimum spanning branching problem. Every spanning arborescence is a spanning tree if we ignore directionality, not vice versa. Proposition 1 does not apply because in undirected graphs every edge is part of some spanning tree, but in directed graphs some edges (e.g., those entering r) cannot be used. The directed version of cut and cycle rules are no longer valid. Prim's and Kruskal's algorithms fail to find the minimum spanning r-arborescence because it seems that edge-directionality introduces significant complications to the problem.

5.1 Edmonds' algorithm

Rooted trees are univocally defined by a parent relationships for all the nodes but the root. Given a directed graph $G = (N, E)$ an r-arborescence $T \subseteq E$ has no cycles and for each node $v \in N$ except r there is exactly one edge in T entering v. We observe, selecting the leanest entering edge for each node except the root is a promising greedy algorithm because it succeeds in minimizing the sum, but it may fail to generate an arborescence (it might create cycles). Also, subtracting a constant amount $c \in \mathbb{R}$ for each edge entering a node does not change the set of edges in the outcome: every arborescence have weight decreased by c. Edmonds' algorithm [5] follows from these two observations: it first selects the minimum cost entering edges for all the nodes but the root, then it accordingly decrease the costs of all the entering edges; when it detects cycles it contracts them into a single node and it repeats the procedure until it succeed at generating an arborescence. It makes greedy choices, but it is not a purely greedy algorithm as it works in multiple phases. It uses a disjoint-set data structure to detect cycles and $N - 1$ priority queues to select the minimum entering edge for each node except the root. There is no need to implement decrease key in the priority queue because in the contraption phase all the edges in the queue decrease by the same amount; hence, the queue order does not change. However, when it contracts a cycle into a single node it needs to merge multiple queues into one. Fibonacci heaps implement merge lazily (with no consolidation), in constant time, while binary and d-ary heap merge in linear time (perform a reconstruction). There are at most $|V| - 2$ contractions, so the worst case running time using binary heaps is $\mathcal{O}(|E||V|)$. Fibonacci heaps improves it to $\mathcal{O}(|E| + |V|\lg|V|)$ [9].

6 Random graphs and their representation

Let $G = (V, E)$ be a graph and let $(i, j) \in E$ be an edge of weight $w \in \mathbb{R}$. We encode the set of nodes with the first $|V|$ natural numbers, starting from zero. There are few ways to represent graphs: an *adjacency matrix* representation consists in an $|V| \times |V|$ matrix m with $m_{i,j} = w$, an *edge list* representation is compact list of (i, j, w) entries, and an *out-stars* representation use an array of $|V|$ lists with (j, w) in list

i. Each of them have strengths and weaknesses and, clearly, the most convenient for our purpose is the adjacency matrix. The analysis of the algorithms already take this representation in to consideration.

data structure	space	access	iterate
adj matrix	$\mathcal{O}(\lvert V\rvert^2)$	$\mathcal{O}(1)$	$\mathcal{O}(\lvert V\rvert)$
edge list	$\mathcal{O}(\lvert E\rvert)$	$\mathcal{O}(\lvert E\rvert)$	$\mathcal{O}(\lvert E\rvert)$
out-stars	$\mathcal{O}(\lvert E\rvert+\lvert V\rvert)$	$\mathcal{O}(d_G^+(v))$	$\mathcal{O}(d_G^+(v))$

We defined these structures with focus on the out-edges, but they can also be defined in terms of in-edges by swapping i with j in their definitions. This inverted representation is not frequently used, but it is convenient for selecting the leanest entering node in Edmonds' algorithm (see section 5).

To generate random graphs (or digraphs) we require that all the edges have the same probability to be select and such probabilities to be independent. Let d be the density of the graph, m be the maximum number of edges and $k = md$ the number of edges we want to randomly extract. A wasted iteration occurs when the extracted edge is already in the set. If $d > ½$ it is faster to select the $m-k$ edges in the complement graph; hence, with no loss of generality we assume $k \leq m/2$. The geometric distribution $\Pr(X = k) = (1-p)^{k-1}p$ gives the probability that the first success requires k independent trials, each with success probability p. By contrast, $\Pr(Y = k) = \Pr(X = k+1) = (1-p)^k p$ is the number of failures until the first success. Let W_k be the number of wasted iterations when building a set E of k edges and let $R_k = W_k - W_{k-1}$ be the number of wasted iterations it takes to expand the set of edges from size $k-1$ to size k. R_k is geometrically distributed with $p = 1 - \frac{k-1}{m}$, therefore

$$\Pr(R_k = n) = (1-p)^n p$$
$$E[R_k] = E[W_k] - E[W_{k-1}] = \frac{1}{p} - 1 = \frac{m}{m-k+1} - 1$$

and since we know that $E[W_0] = 0$, we can sum it up

$$E[W_k] = \sum_{n=1}^{k} E[R_k] = \sum_{n=1}^{k} \left(\frac{m}{m-k+1} - 1 \right).$$

The expected number of wasted iterations increases as $\mathcal{O}(m \ln m/k)$ as k approaches $m/2$ because[4]

$$H(n) = \sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < 1 + \int_1^n \frac{1}{x}\,dx = 1 + \Big[\ln x\Big]_1^{n+1} = 1 + \ln(n+1)$$
$$E[W_k] = m\left(\sum_{n=m-k+1}^{m} \frac{1}{n} \right) = m\,(H_m - H_{m-k}) - k < m\,(\ln(m+1) - \ln(m-k+1)) - k = m \ln \frac{m+1}{m-k+1} - k.$$

7 Experimental results

Our benchmarks are performed on an Intel® Core™ i7–7770 CPU @3.60 GHz (Kaby Lake), with 64 GiB of DDR4 RAM, Linux 5.15.16 and the GNU C compiler 11.2.0 with optimization enabled. The

[4]The n-th harmonic number $H(n)$ sums the reciprocals of the first n strictly positive natural numbers.

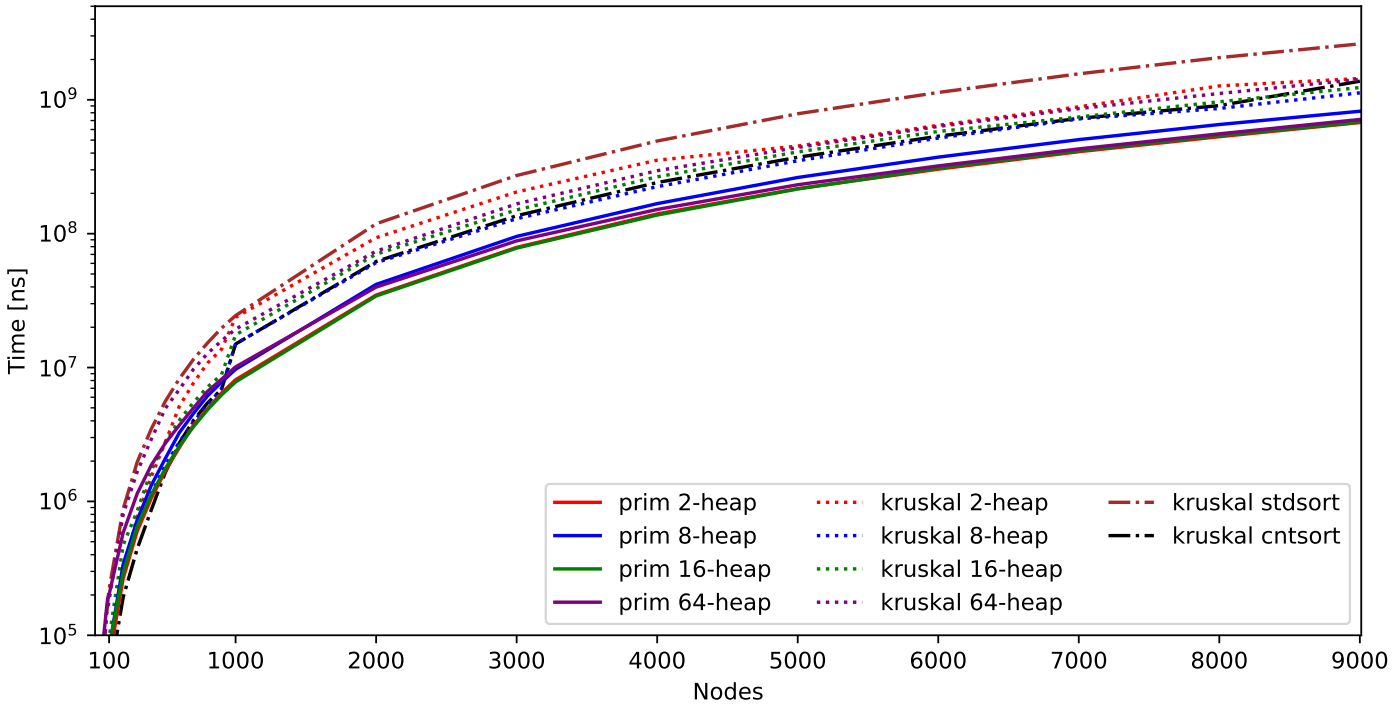

Figure 1: Performance of Prim's and Kruskal's algorithms on medium density graphs at varying size.

graphs are constructed with fixed number of nodes, fixed density and random edges, as discussed in see section 6. We enforce the existence of spanning trees and spanning r-arborescences by adding overweight edges (see section 4). We use line style to distinguish the main algorithm— Prim's algorithm (solid lines), Kruskal's algorithm using heaps (dotted lines), Kruskal's algorithm using sort (dash-dot lines), Edmonds' algorithm (dash lines)—and colors to distinguish their variants—the arity of the underlying heap data structure and the sorting algorithm used.

7.1 The minimum spanning tree

We display the pros and the cons of finding the minimum spanning tree with various approaches. We analyze distinctly fixed-density graphs at varying size (figures 1 to 3) and fixed-size graphs at varying density (figure 4). In order to enable the use of the linear time counting sort, the weight of each node is extracted randomly in the range of integers from 1 to 1000. The Fibonacci heap variant of the algorithms (excluded in the figures) performs about two order of magnitudes slower than the others. As expected, sorting is most convenient when the graph is sparse; otherwise, the edges in the graphs are a lot more than the edges in the spanning tree and sorting them all is usually a waste of time. If the graph is sparse and the weights can be sort in linear time, Kruskal's algorithm is substantially faster than the alternatives. The 64-heap never performs better than the other heaps; therefore, it is better to use a small value for the arity. Our results suggest that the best arity is indeed hardware-dependent fine tuning. Prim's algorithm is overall faster than Kruskal's algorithm with priority queues and it is less sensitive to variation of arity.

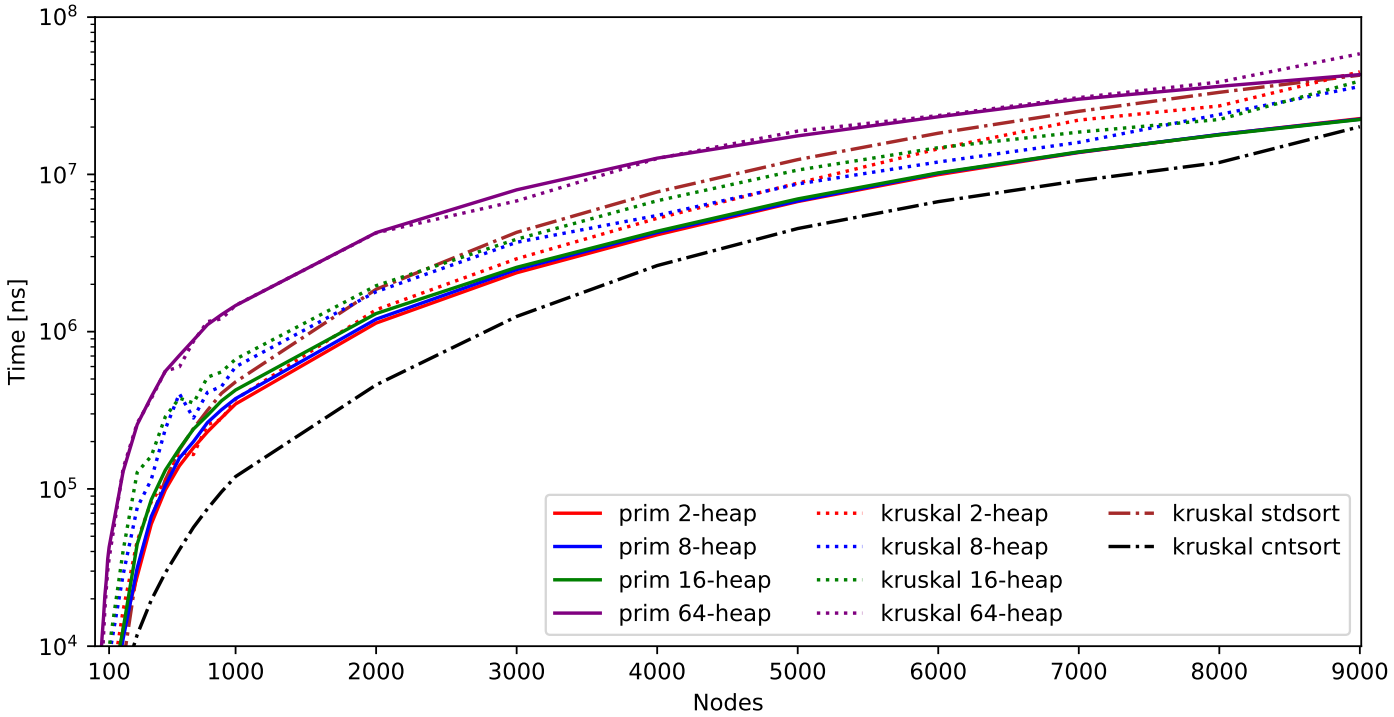

Figure 2: Performance of Prim's and Kruskal's algorithms on sparse graphs at varying size.

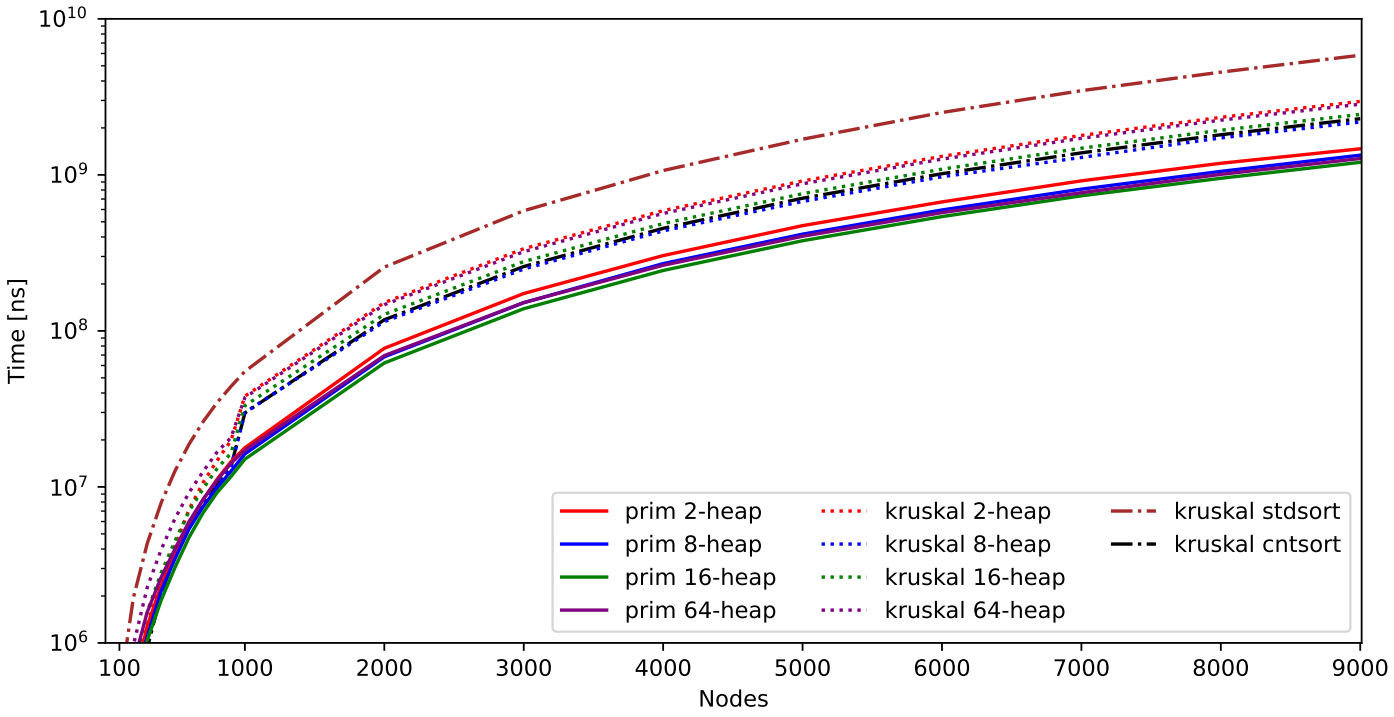

Figure 3: Performance of Prim's and Kruskal's algorithms on complete graphs at varying size.

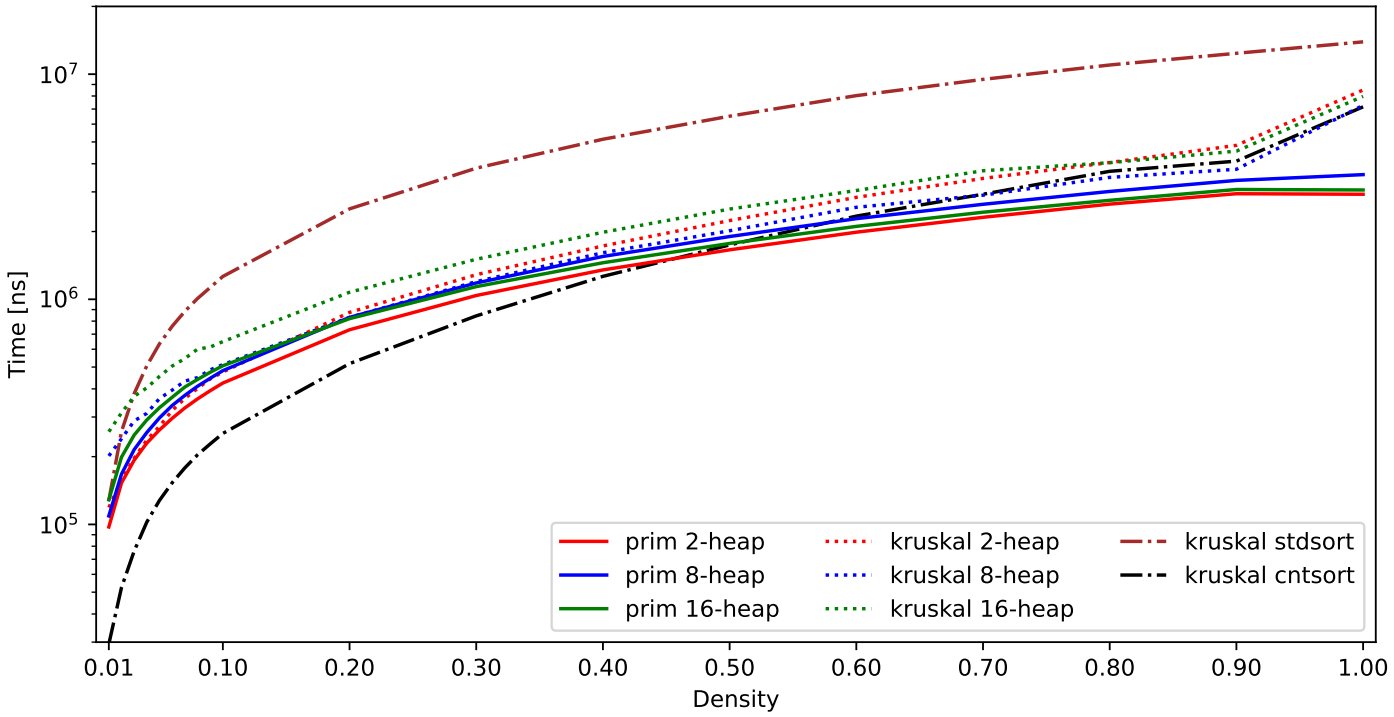

Figure 4: Performance of Prim's and Kruskal's algorithms on 500 nodes graphs at varying density.

7.2 The minimum spanning r-arborescence

In figures 5 and 6 we display the performance of Edmonds' algorithm in graphs at varying size and density. Graphs are represented with the inverted adjacency list discussed in section 6. Unlike Kruskal's and Prim's algorithm, Edmonds' running time is extremely dependent on the shape of the graph: as the algorithm proceeds, it may select a set of arcs that form a cycles; when a cycle is detected, the priority queues merge and grow in size. Random graphs are extremely susceptible to this behavior: in some of them Edmonds' algorithm does not merge frequently while in others it merges a lot. In order to display meaningful results, each point in figures 5 and 6 is the median of 50 repetitions of the algorithm and figure 7 show the variation[5] of 1000 repetitions at fixed size and fixed density. At each repetition we rerandomize the graph without altering size and density. From a theoretical standpoint Edmonds' algorithm benefits from the Fibonacci heap but, in practice, as we have seen in section 7.1, the memory layout of the structure and its sophisticated implementation is unfriendly to modern computers. In our study we do not use large enough instances for the Fibonacci heap variant to prevail to all the others, yet it appears to be only slightly slower than the others and faster than the 64-heap while in Prim's and Kruskal's algorithms the difference was much more considerable.

[5]The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

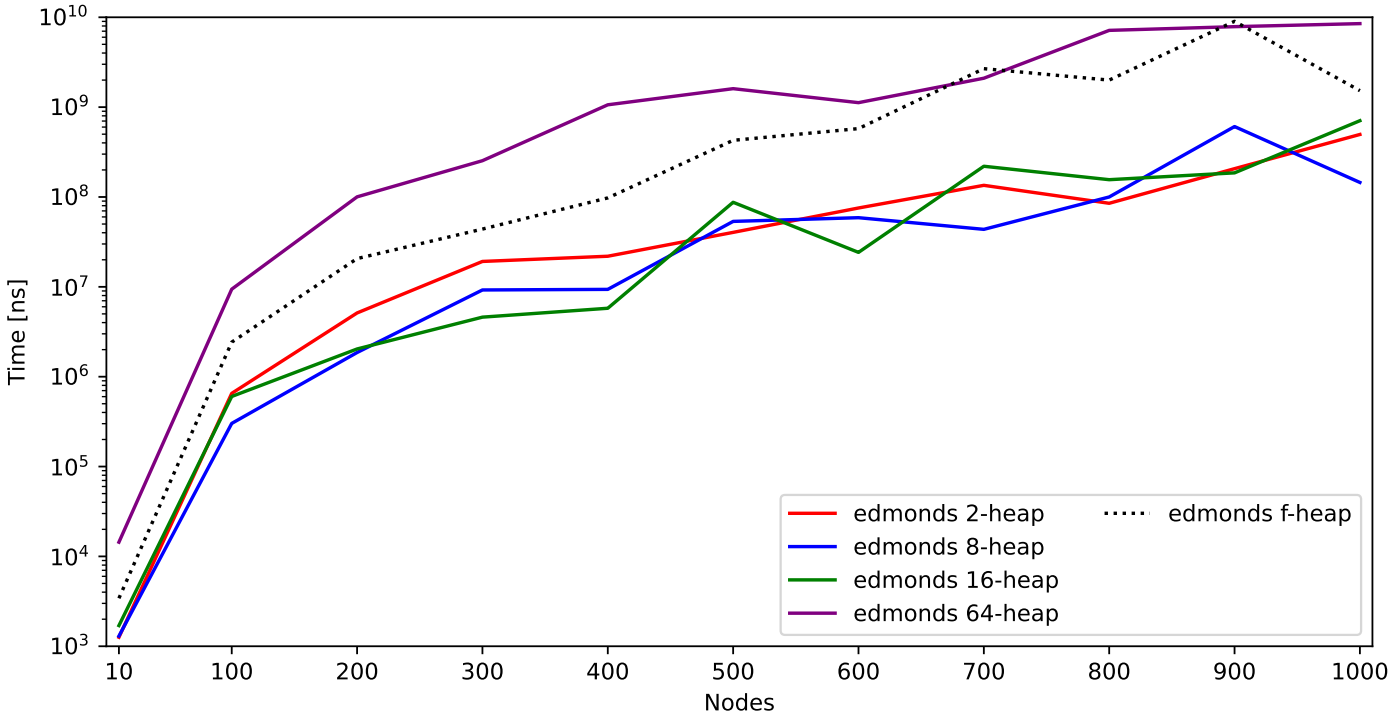

Figure 5: Performance of Edmonds' algorithm on complete graphs at varying size.

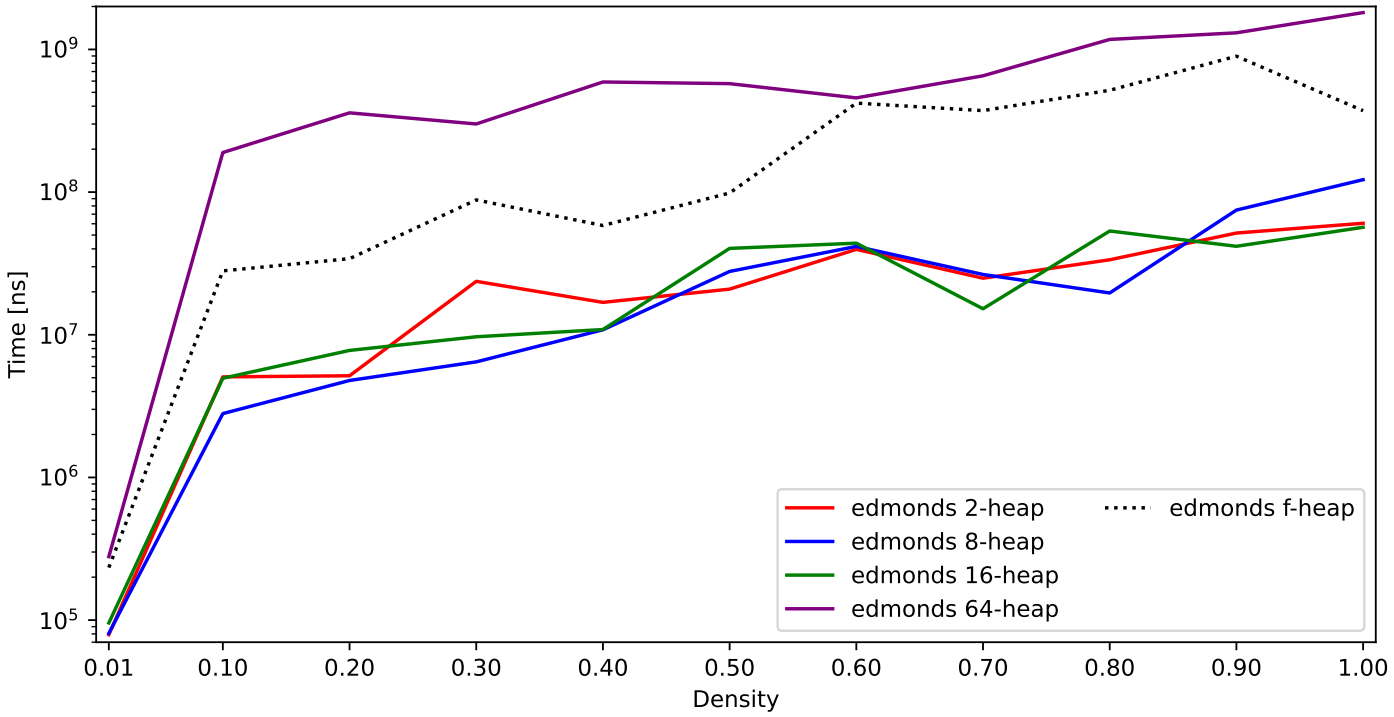

Figure 6: Performance of Edmonds' algorithm on 500 nodes graphs at varying density.

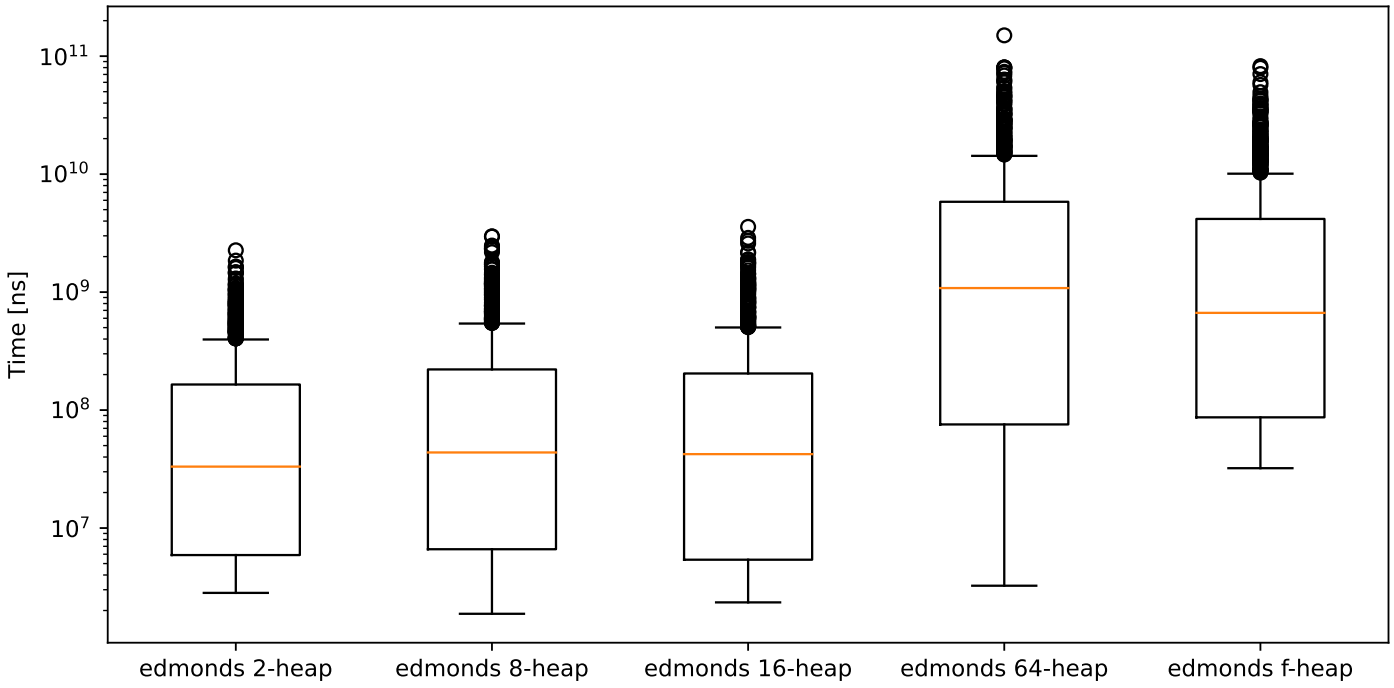

Figure 7: Performance of Edmonds' algorithm on 500 nodes complete graphs.

8 Conclusions

We have studied the behavior of the main algorithms to compute the minimum spanning tree for directed and undirected graphs. The main lesson learned is that trying different algorithms is more meaningful than tweaking their underlying data structures and that efficient in theory does not always coincide with efficient in practice.

For a general purpose algorithm to solve the minimum spanning tree problem we suggest Prim's algorithm; nevertheless, Kruskal's algorithm has its use cases. While Prim's performance is consistent at varying density, Kruskal's shines on sparse graphs. Kruskal's algorithm is also very convenient (both in theory and in practice) when it is possible to use some specific-purpose sublinearithmic time sorting algorithms like the linear time counting sort for integer weights. Moreover, its memory consumption is about half the memory consumption of Prim's because every edge is implicitly bidirectional. We recall, this study focus on sequential approaches and does not consider Borůvka's algorithm [11] which might be more convenient in parallel applications [12].

The underlying heap data structure to implement priority queues should either be the binary heap or the d-ary heap. The Fibonacci heap is most suited for theoretical studies. The binary heap is practical because it is a standard tool available as an off-the-shelf software from the standard library of many programming languages. The d-ary heap can be slightly faster than the binary heap, but our results suggest that the best value of d is hardware dependent fine tuning: 8 and 16 works fairly well in our benchmark workstation, but newer x86_64 processors ship memory caches of size about ten times larger than what available in our system; thus, your mileage may (slightly) vary. In Edmonds' algorithm the difference in performance between Fibonacci heaps and the others is not as abysmal as in Prim's and Kruskal's algorithm. The better asymptotic behavior suggests they will be convenient

at some point, in larger graphs, nonetheless the algorithm is already quite slow and it should be questionable whether or not it would be practical to use it on much larger graphs.

In fact, a strict theoretical lower bound for the minimum spanning tree and the minimum spanning arborescence problems is still unknown; however, the minimum spanning arborescence problem seems to be intrinsically harder. Edmonds' performance does not only depends on size and density, but also on the hostility of the graph in making the algorithm form cycles and merge heaps. We believe the discovery of a faster minimum spanning arborescence algorithm would focus on not violating the arborescence structure rather than on enforcing the weight minimality as an invariant, the opposite of Edmonds' algorithm does.

References

[1] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.

[2] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[3] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[4] Y.-J. Chu, "On the shortest arborescence of a directed graph," *Scientia Sinica*, vol. 14, pp. 1396–1400, 1965.

[5] J. Edmonds *et al.*, "Optimum branchings," *Journal of Research of the national Bureau of Standards B*, vol. 71, no. 4, pp. 233–240, 1967.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009, ISBN: 978-0-262-03384-8. [Online]. Available: `http://mitpress.mit.edu/books/introduction-algorithms`.

[7] J. M. Kleinberg and É. Tardos, *Algorithm design*. Addison-Wesley, 2006, ISBN: 978-0-321-37291-8.

[8] R. E. Tarjan and J. Van Leeuwen, "Worst-case analysis of set union algorithms," *Journal of the ACM (JACM)*, vol. 31, no. 2, pp. 245–281, 1984.

[9] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM (JACM)*, vol. 34, no. 3, pp. 596–615, 1987.

[10] D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973, ISBN: 0-201-03803-X.

[11] J. Nešetřil, E. Milková, and H. Nešetřilová, "Otakar boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history," *Discrete mathematics*, vol. 233, no. 1-3, pp. 3–36, 2001.

[12] S. Chung and A. Condon, "Parallel implementation of bouvka's minimum spanning tree algorithm," in *Proceedings of International Conference on Parallel Processing*, IEEE, 1996, pp. 302–308.