Pettie and Ramachandran's Optimal MST Algorithm

Eli Berg, Vihan Lakshman, Sachin Padmanabhan

June 1, 2016

1 Introduction

As one of the oldest, well-studied, and application-rich combinatorial optimization tasks of all time, the minimum spanning tree (MST) problem has served as a rich breeding ground for remarkable advancements in algorithms and data structures. There may be no better example of the computer science mantra "can we do better" constantly manifesting itself within a single problem as theorists have repeatedly improved upon the previously best-known bounds for finding MSTs through the turn of millennium, introducing a host of new techniques and insights along the way. This journey to find better algorithms for the MST problems arrived at a major milestone in 2002 when Seth Pettie and Vijaya Ramachandran of the University of Texas at Austin announced that they had discovered a provably optimal deterministic algorithm, running in time proportional to the minimum number of edge-weight comparisons necessary to compute the solution on a given graph. However, while provably optimal, the precise runtime of Pettie and Ramachandran's algorithm remains an open problem since generalizing the time needed for the minimum number of edge-weight comparisons on arbitrary graphs has proven to be difficult. Nevertheless, we do have some reasonably strong bounds on the runtime. In the worst case, we must examine every edge of a graph (if it is already a tree) so we have a lower bound of $\Omega(m)$, where m is the number of edges in the graph, on the runtime of Pettie-Ramachandran. In addition, since the algorithm is optimal, it cannot perform worse than the best-known precise deterministic bound of $O(m \cdot \alpha(m, n))$ due to Bernard Chazelle in 2000, where α is the very slow-growing inverse Ackermann function and n is the number of vertices in the graph.

While Pettie and Ramachandran's algorithm contains many distinct steps and utilizes several complex data structures, we hope to provide a strong intuition in this paper for how the various pieces mesh together. In that spirit, we will begin by providing a historical overview of previous MST algorithms, borrowing heavily in both content and style, from Jason Eisner's 1997 survey. Beginning with the earliest-known procedure due to Otakar Boruvka in 1926 and culminating in Chazelle's algorithm in 2000, we hope to show that the key components of Pettie and Ramachandran's algorithm are far from out-of-the-blue and instead represent a natural extension of ideas developed in these earlier algorithms. For example, Pettie and Ramachandran's use of soft heaps in their algorithm stems directly from the work of Chazelle, who invented the data structure specifically to solve the MST problem quickly. Additionally, the optimal MST algorithm's use of a fixed number of steps of Boruvka's algorithm traces its historical legacy back to the expected linear time randomized algorithm due to David Karger, Philip Klein, and Robert Tarjan in 1995.

After completing our overview of how the Pettie-Ramachandran algorithm fits into the broader historical context of MST algorithms, we will proceed to describe the algorithm in full detail with a focus on making each part accessible and intuitive. Since the optimal MST algorithm uses many previous MST algorithms and related data structures as subroutines, it feels natural to describe our algorithm after our historical survey. After presenting the algorithm, we will proceed to present proofs of both its correctness and runtime optimality.

The "interesting" component of our project is three-fold. The first, as we have already detailed above, is our historical overview and, in particular, placing Pettie and Ramachandran's work in the broader intellectual context of the previous algorithms, a task which we believe has not been undertaken previously. Eisner's 1997 survey on MST algorithms provides an excellent overview, but does not include the subsequent groundbreaking work of Chazelle and Pettie and Ramachandran in the 21st century. Thus, we hope that our background work in charting the advancement and commonalities of ideas in the development of MST algorithms can also serve as an extension of Eisner's tutorial and be of value to the computer science community at-large.

The second part of our "interesting" component (and by far the most time-intensive portion of this project) was implementing the component data structures and some of the subroutines used in the Optimal MST algorithm to gauge the difficulty of implementing this algorithm in practice. We discuss the design of our implementation in Section 5 of this paper with a particular emphasis on certain optimizations we made to streamline implementation of our algorithm. While we were not able to implement the full algorithm, we hope that by sharing our design methodology and optimization ideas, we can provide a sense of the practical challenges associated with this algorithm and the type of modifications that could help streamline implementations.

Finally, the third leg of our "interesting" component is researching upper-bounds on the runtime of the algorithm. Though we were not able to prove the runtime of the algorithm in general, we present a very simple proof that the algorithm will run in $O(m)$ time on all graphs with fewer than $2^{2^{16}}$ vertices building off of ideas from Claus Andersen of the University of Aarhaus in Denmark. In addition, we present a simplified version of a proof given in Pettie and Ramachandran's original paper that the algorithm will run in linear time on Erdos-Renyi random graphs with high probability.

2 Motivation: Why Study MSTs or Pettie-Ramachandran?

Before jumping into our discussion of why we should study the MST problem and the Pettie-Ramachandran algorithm, let's first formally define the minimum spanning tree problem. Given a connected, undirected graph $G = (V, E)$ with no self loops and weights assigned to each edge, a minimum spanning tree is an acylic subset of the edges $T \subseteq E$ that connects the entire graph and minimizes $\sum_{e \in T} w(e)$ where $w(e)$ represents the weight assigned to edge e in the graph. As an additional assumption, we will assert that the edge weights are all distinct which is a requirement of Boruvka's and Chazelle's algorithms. Intuitively, the MST is the tree that connects the entire graph as cheaply as possible. For the remainder of the paper, we will assume that our input graph is connected. One can also compute a minimum spanning *forest* using existing MST algorithms by identifying the connected components of the graph in linear time and then computing the MST on each connected component. Therefore, we can safely assume that the input graph is connected with the knowledge that the problem generalizes quite nicely.

In addition to being a fertile ground for theoretical advancements, the minimum spanning tree problem has also enjoyed a long legacy of practical applications, both directly and indirectly. Designing circuits, computer networks or communication infrastructures, for example, are all natural application of the minimum spanning tree problem since one often wants to maintain the connections between a set of agents as cheaply as possible. In addition to these very direct and vital applications, minimum spanning trees are also used as a key heuristic in state-of-the-art algorithms for a number of other computational problems, including Christofides' 3/2-approximation algorithm for the Traveling Salesman Problem. Given these numerous applications, finding improved algorithms for computing a minimum spanning tree is clearly of immense interest from a practical perspective in addition to the theoretical benefits.

Moreover, Pettie and Ramachandran's achievement remains an extremely valuable algorithm to study in it's own right. First and foremost, the algorithm still represents the current state-of-the-art and is asymptotically optimal. Thus, studying the Pettie-Ramachandran algorithm provides a window into the cutting-edge of computer science and the mysterious nature of the runtime continues to be a fascinating open problem of immense interest. In addition, as we will discuss in the next section, the Pettie-Ramachandran algorithm also stands as an intellectual culmination of decades of work in efficiently solving the MST problem and combines many of these historic insights together into an optimal algorithm. Studying the Pettie-Ramachandran algorithm provides an excellent lens into the interplay of several other concepts in computer science, including the Fibonacci heap and soft heap data structures, other MST algorithms, the use of precomputation to speed up the runtime (Method of Four Russians), and the notion of decision tree complexity. As a result, Pettie-Ramachandran is not only an optimal algorithm, but also a celebration of several great ideas in computer science. In the next section, we will review several of these great ideas and fill in the remaining details in our full description of the algorithm in Section 4.

3 A Journey of MST Algorithms Through Time

3.1 A Whimsical Historical Overview - Extended

In his original tutorial, Eisner opens by informally (and hilariously) describing each MST algorithm in terms of the paradigm of "How to Organize a Grass-Roots Revolutionary Movement as Quickly as Possible." Below, we reproduce Eisner's original problem statement and description of some (but not all) of the algorithms he addresses in his paper. In particular, we only reproduce the algorithms which we also address in this section. The curious reader is encouraged to consult Eisner's original paper for the remainder of the algorithms.

Finally, at the end of reproducing Eisner's descriptions, we provide our own descriptions of Chazelle's Algorithm and Pettie-Ramachandran under this revolutionary movement paradigm, both to pay tribute to Eisner as we build on his survey and provide for a fun, informal introduction of the algorithms before diving into the technical details.

Problem Statement: "Russia, 1895. Idealists are scattered throughout the country. Their revolutionary ambitions hinge on their ability to make and maintain contact with each other. But channels of communication can be risky to establish, use, and maintain. The revolutionaries need to establish enough channels between pairs of individuals that a message can be sent – perhaps indirectly – from anyone to anyone else. Naturally they prefer safe channels, and no more channels than necessary: that is, the total risk is to be minimized. In addition, they want to speed the revolution by establishing this network as quickly as possible!"

Kruskal (Eisner) Safe channels should be opened first and risky ones left for a last resort. The first two individuals in Russia to establish contact should be those who can most safely do so. The next safest pair, even if in far-off Siberia, should establish contact with each other next.

Prim (Eisner) Starting with Lenin, the central revolutionary organization in St. Petersburg must grow by recruiting new members who are close friends of some existing memberas close as can be found.

Boruvka (Eisner) Every one of us should at once reach out to his or her closest acquaintance. This will group us into cells. Now each cell should reach out to another nearby cell, in the same way. That will form supercells, and so forth.

Tarjan-Fredman (Eisner) The St. Petersburg organization should grow just as Comrade Prim suggests, but only until its Rolodex is so full that it can no longer easily choose its next recruit. At that time we can start a new revolutionary organization in Moscow, and then another in Siberia. Once we have covered Russia with regional organizations, the St. Petersburg organization can start recruiting other entire organizationsso that cells will be joined into supercells, as Comrade Boruvka has already proposed.

Karger-Klein-Tarjan (Eisner) Suppose we make this easier on ourselves by ignoring half the possible channels, at random. We will still [by recursion] be able to establish some fairly good paths of communicationso good that the channels we ignored are mainly riskier than these paths. A few of the channels we ignored, but only a few, will be safe enough that we should try to use them to improve the original paths.

And now we move on to our additions to this revolutionary paradigm.

Chazelle I have invented a brilliant new communication device that will allow us to reach out to acquaintances (as Comrade Boruvka proposed) in much faster time. However, this device is highly unstable and might corrupt some of these communication channels so we must take care to identify these corrupted links and reintroduce them into the network.

Pettie-Ramachandran The ideas of Comrades Boruvka, Tarjan-Fredman, Karger-Klein-Tarjan, and Chazelle are all brilliant and we propose that we combine them all using our key insight, which we shall call–in their honor–the Method of Four Russians. Before we even examine our locations across the land, we will take a moment to pre-calculate all possibilities for an optimal connection system on sufficiently small networks. Then, we will partition ourselves into smaller chunks as previously suggested and find the optimal channels of communication based on our pre-calculated work. We then combine everything together in the fastest way possible and unite us together to commence the Revolution!

3.2 Boruvka's Algorithm (1926)

The earliest algorithms for the MST problem all proceed in a greedy fashion and Boruvka's approach is no exception. Invented by the Czech mathematician Otakar Boruvka to address the problem of constructing an electricity network in his home country, Boruvka's algorithm proceeds by recursively grouping nodes into larger and larger connected components, often referred to as "supernodes," and terminating once a single connected component remains. The edges of this connected component are then returned as the MST edges. Despite being the oldest-known MST algorithm, Boruvka's approach remains extremely relevant today because it's fundamental idea of recursively grouping nodes into clusters is used in both Pettie-Ramachandran and in the randomized Karger-Klein-Tarjan algorithm. Moreover, the algorithm is one of the few highly parallelizable MST procedures, making it further relevant in the age of distributed computing. We now present the full algorithm below:

Algorithm 1 Boruvka's Algorithm

```
procedure BORUVKA(G = (V, E))
    $S \leftarrow \emptyset$
    $T \leftarrow \emptyset$
    for each $v \in V$ do
        Create a supernode $s$ containing $v$
        $S \leftarrow S \cup v$
    end for
    while There is more than one supernode in S do    ▷ Find cheapest edge out of each node
        Create a supernode $s$ containing $v$
        $X \leftarrow \emptyset$
        for each $s \in S$ do
            Find minimum weight edge $(s, s')$ going from $t$ to another supernode
            $X \leftarrow X \cup (s, s')$
        end for
        for each $(s, s') \in X$ do
            Merge the supernodes $s$ and $s'$
            $T \leftarrow T \cup (s, s')$
        end for
    end while
    return $T$
end procedure
```

Intuitively, Boruvka's algorithm iteratively merges supernodes into larger connected components. One can optimize this contraction procedure through the use of a disjoint set data structure, which we will discuss further when detailing our own implementation in Section 5.

Runtime Analysis

During each iteration of the algorithm, the number of available vertices decreases by at least half since we merge each vertex with at least one other vertex. Therefore, the algorithm will run for $O(\log(n))$ iterations. Moreover, each iteration takes $O(m)$ time since we must scan every edge to determine the cheapest edge coming out of each supernode. Therefore, we can bound the total runtime of Boruvka's algorithm by $O(m \log(n))$. In his survey paper, Eisner sharpens this bound to $O(\min(m \log(n), n^2))$ through a more careful analysis. We will return to Boruvka's algorithm in our discussion of Karger-Klein-Tarjan and, in particular, consider the impact on the runtime if we only run Boruvka for a fixed number of iterations. We will then revisit this analysis when we turn our attention to Pettie-Ramachandran.

3.3 Prim's Algorithm (1930)

The next algorithm we will consider is Prim's algorithm, alternatively called Jarnik's algorithm or the Dijkstra-Jarnik-Prim algorithm. It was first discovered by Czech mathematician Vojtech Jarnik in 1930 and later independently by Prim in 1957. The algorithm operates in a greedy fashion, arbitrarily choosing a start vertex v and then incrementally growing the MST by choosing the cheapest edge $e = (v, v')$ out of v and growing our set of visiting vertices to include both v and v'. In the next iteration, the algorithm will choose the cheapest edge out of either v or v', add a new node to the set of visited nodes and then continue until $n - 1$ edges are chosen. The full pseudocode of the algorithm is presented below:

Algorithm 2 Prim's Algorithm

```
 1: procedure PRIM(G = (V, E))
 2:     $S \leftarrow \emptyset$
 3:     $T \leftarrow \emptyset$
 4:     Choose an arbitrary start vertex v
 5:     $S \leftarrow S \cup v$
 6:     while $|T| \leq |V| - 1$ do
 7:         Find the cheapest edge $e$ going out of a node in $S$
 8:         $T \leftarrow T \cup e$
 9:         Add $e$'s other endpoint into $S$
10:     end while
11:     return $T$
12: end procedure
```

One interesting observation is that Prim's algorithm closely resembles Dijkstra's shortest path algorithm by maintaining a set of visiting nodes and searching for the cheapest edge to a node outside of this set. Prim's algorithm also plays a major role in the Pettie-Ramachandran algorithm even though it is not explicitly invoked as a subroutine. Prim's approach also plays a foundational role in Tarjan-Fredman MST algorithm.

Runtime Analysis:

In order to efficiently implement Prim's algorithm, we need a priority queue data structure that maintains a decrease key function such that the cost of adding an edge not connecting to the current set of visited nodes is $+\infty$, a cost which decreases to the edge's actual cost once the edge is adjacent to the visited set. The most efficient priority queue data with decrease key functionality is the Fibonacci heap, invented by Tarjan and Fredman in the very same paper in which they present their improved MST algorithm (which unsurprisingly utilizes a Fibonacci heap). Using a Fibonacci heap, Prim's algorithm runs in time $O(n \log(n) + m)$

3.4 Kruskal's Algorithm (1956)

Another greedy algorithm with a long and celebrated history is Kruskal's algorithm, first proposed by American mathematician Joseph Kruskal in 1956. Although Kruskal's algorithm does not appear directly in the Pettie-Ramachandran algorithm, we include it in this historical survey because it represents the last of the "greedy era" MST algorithms and is perhaps the simplest MST algorithm to digest and implement. As a result, Kruskal's algorithm is also a very natural benchmark with which to test the later, more advanced MST algorithms in practice. Within the context of MST algorithms, Kruskal's procedure reinforces the idea that algorithms for finding MSTs can be extremely intuitive, but, as we shall see, beating the the time bounds of these greedy algorithms requires quite a bit more insight.

Algorithm 3 Kruskal's Algorithm

```
1: procedure KRUSKAL(G = (V, E))
2:     $T \leftarrow \emptyset$
3:     while $|T| \leq |V| - 1$ do
4:         Select the cheapest edge in the graph $e \notin T$ such that $T \cup e$ does not create a cycle
5:         $T \leftarrow T \cup e$
6:     end while
7:     return $T$
8: end procedure
```

Runtime Analysis
The most efficient method of implementing Kruskal's algorithm in practice is to first sort the edges by weight and use a union find data structure to check if adding an edge will create a cycle in the existing tree. The sorting step will take $O(m \log(m))$ time and the runtime cost of the union find data structure will be $O(m \cdot \alpha(m, n))$ where α is the inverse Ackermann function. The runtime is therefore dominated by the sorting time so Kruskal's runs in time $O(m \log(m))$

3.5 Tarjan-Fredman Algorithm (1984)

Up to this point, we have considered very natural greedy algorithms for the MST problem that almost seem too obvious to be correct. Then, in 1984, Tarjan and Fredman presenting a game-changing algorithm with the introduction of a powerful new data structure: the Fibonacci heap. Their discovery represented a major step forward in efficiently finding MSTs by introducing a new data structure, a theme we will see again when we turn our attention to Chazelle's algorithm in 2000.

To build an intuition for the Tarjan-Fredman algorithm, let's revisit Prim's algorithm and recall that, using the Fibonacci heap invented by Tarjan and Fredman, Prim's algorithm can run in time $O(n \log(n) + m)$. This bound is certainly an improvement over the time cost of running Prim without a Fibonacci heap, but Tarjan and Fredman observed we could do even better by cleverly beating down the $\log(n)$ factor. The key question they considered is: When does the log factor hurt us? As one might expect, precisely when the size of the heap is large, so Tarjan and Fredman developed the key insight to restrict the number of elements that can go into a Fibonacci heap while running Prim's algorithm. When the heap reaches it's maximum size, we then choose another vertex and start over, merging the trees at the end of the algorithm.
Runtime Analysis
One can show through a careful analysis that the runtime of the Tarjan-Fredman algorithm, using Fibonacci heaps to execute the Prim subroutine, is $O(m \log^{\star}(n))$ which Gabow, et al. later further reduced, using an additional clever counting scheme of "heavy" and "light" edges to $O(m \log(\beta(m, n)))$ where $\beta(m, n) = \min\{i | \log^{(i)}(n) \leq \frac{m}{n}\}$

Perhaps the most significant runtime property of the Tarjan-Fredman algorithm is that it finally allows us to take a small sip from the holy grail of a linear time MST algorithm! In the special case where the input graph is sufficiently dense, which Pettie and Ramachandran define for the purposes of their algorithm as $\log^{(3)}(n) \leq \frac{m}{n}$, the Tarjan-Fredman algorithm will run in $O(m)$ time. This fact will be crucial for the design of the Pettie-Ramachandran algorithm, since at one stage

Algorithm 4 Tarjan-Fredman Algorithm

```
1: procedure Tarjan-Fredman(G = (V, E))
2:     Select a threshold value $k$        ▷ The details of how to compute $k$ are omitted for simplicity
3:     while Not all nodes belong to a tree do
4:         Pick an arbitrary unmarked vertex $v \in V$ and start Prim's algorithm from $v$, growing a
       tree $T$.
5:         Keep track of the lightest edge from $T$ to each vertex not in $T$ but adjacent to $T$
6:         Call this set of vertices adjacent to $T$ the neighborhood of $T$, denoted $N(T)$
7:         if $N(T) >$ k or $T$ has the algorithm just added an edge to a vertex previously marked
       then
8:             Mark all vertices in the currently tree $T$
9:         end if
10:    end while
11:    return $T$
12: end procedure
```

the algorithm contracts nodes to create a dense graph and then runs a linear-time "dense case" algorithm, which we know exists thanks to work of Tarjan and Fredman. Consequently, we can view the development of the Tarjan-Fredman algorithm (and their invention of the Fibonacci heap) as a major pillar of inspiration for the Optimal MST algorithm since Pettie and Ramachandran essentially use Tarjan-Fredman as a subroutine to compute an MST on dense graphs in linear time. Moreover, Tarjan and Fredman introduced the idea of partitioning graphs into smaller sizes and computing MST's on each of these subgraphs, which shaves a log factor from Prim's algorithm. Pettie and Ramachandran build off this idea to partition the input graph and then compute the MST on each subgraph in the optimal amount of time using a Four Russians-style technique of precomputing decision trees (which we will discuss in more depth in the following section). The main takeaway from the Tarjan-Fredman algorithm, though, is that Pettie and Ramachandran's insight of partitioning has a legacy in earlier algorithms.

3.6 Karger-Klein-Tarjan Algorithm (1995)

As mentioned, the study of MST algorithms can be thought of as an eternal quest to find a deterministic algorithm that runs in linear time in all cases. In 1995, Karger, Klein, and Tarjan took another major step forward towards this goal by presenting a Las Vegas randomized algorithm for the MST problem that runs in $O(m)$ time in expectation. The algorithm can also be considered a direct intellectual ancestor of the Pettie-Ramachandran algorithm because of it's introduction of "Boruvka steps" - calling the inner-loop of Boruvka's algorithm exactly two times on a given graph. Pettie and Ramachandran invoke precisely this same subroutine of running two Boruvka steps in their optimal MST algorithm.

The other major building block of Karger-Klein-Tarjan was the discovery of an $O(m)$ time algorithm for the related *MST verification problem* which is a decision problem that takes in a tree as input and asks whether or not the given tree as a minimum spanning tree and, if not, which edges do not belong. The description and analysis of this verification algorithm is highly nontrivial and we defer the details to Eisner's survey paper. Nevertheless, we can treat the existence of a linear time algorithm for MST verification as a black box and present the pseudocode of Karger-Klein-Tarjan below:

Algorithm 5 Karger-Klein-Tarjan Algorithm

```
1: procedure KARGER-KLEIN-TARJAN(G = (V, E))
2:     Create a contracted graph $G'$ by running two iterations of Boruvka's algorithm on $G$
3:     Create a subgraph $H$ by selecting edges from $G'$ with probability 1/2 of being included in $H$
4:     Recursively apply the algorithm to obtain a minimum spanning forest $F$ on $H$    ▷ Since we choose edges of $H$ randomly, the subgraph is not guaranteed to be connected, so we find a minimum spanning forest as opposed to a tree
5:     Connect $F$'s components into a tree by adding infinitely heavy edges
6:     Run an MST verification algorithm on $F$ to identify edges too heavy to be in the MST of $G$
7:     Remove these bad edges from $G$ and recursively apply the algorithm to the pruned graph
8:     Return the MSF on this pruned graph
9: end procedure
```

As one might expect, the proof of correctness and runtime analysis of this algorithm is quite complex and we will again defer to Eisner for the details. As mentioned, the most pertinent feature of this algorithm for our purposes in the introduction of the Boruvka step and we hope that presenting Karger-Klein-Tarjan can provide some motivation of why Pettie and Ramachandran would choose to include Boruvka steps in their algorithm since the contradiction proces for a fixed number of steps runs in linear time and often simplifies the input graph dramatically. The remaining details of this algorithm are incredibly fascinating as they draw on techniques from greedy algorithms, divide-and-conquer, dynamic programming, and this state-of-the-art MST verification process. We encourage the interested reader to consult Eisner and the author's original paper for further exploration.

3.7 Chazelle's Algorithm (2000)

Chazelle's algorithm represents another watershed moment in the history of solving the minimum spanning tree problem. The algorithm is arguably the most complex of all of the MST algorithms we discuss in this paper, a far departure from the natural greedy algorithms half a century earlier. Like the Tarjan-Fredman algorithm which utilized Fibonacci heaps, Chazelle's algorithm was developed following the emergence of a brand new data structure, the soft heap, which also plays an integral role in Pettie and Ramachandran's algorithm.

Before we describe the properties of a soft heap, let's first provide some motivation for why this data structure is useful for the minimum spanning tree problem by considering what properties we would like the data structure to have. Let's begin by considering an analogy to another well-known data structure, the count[-min] sketch, which allows one to estimate the number of each particular element in data stream by reducing the amount of space required in exchange for relaxing the demand of absolute correctness in our estimate. Instead, the count[-min] sketch might corrupt some of the data, up to a fixed error tolerance parameter. Similarly, for the minimum spanning tree problem, we would like a data structure with a similar tradeoff. While the count[-min] sketch trades accuracy for space efficiency, we would like a priority queue data structure that can perform operations faster, at the risk of losing some accuracy. Fibonacci heaps take $O(\log(n))$ amortized time to extract the minimum element and this is too slow for our purposes of improving our runtime bounds for finding an MST. We would like a priority queue that can support $O(1)$ extract min, but we recognize that there is no free lunch so we are willing to sacrifice some guarantees on accuracy. This is exactly what the soft heap provides us, and, amazingly, Chazelle also showed that we can use this data structure for the MST problem and still devise a deterministic algorithm that is always correct despite the fact that the soft heap may corrupt some data.

We will now briefly describe the properties of the soft heap data structure, which will again come into play in the Pettie-Ramachandran algorithm. The soft heap supports all of the bread-and-butter operations of a priority queue, namely insertion, extraction of the minimum element, and melds. As mentioned, soft heaps trade absolutely correctness for speed and thus each soft heap is parameterized by an error tolerance ϵ which is typically set between 0 and 1/2. The primary goal of soft heaps, as we discussed in the previous paragraph, is to support fast extract-min operations and we can perform this operation in $O(1)$ time on a soft heap. Moreover, we can also meld two heaps in constant time. Now, an insertion will run in time $O(\log(\frac{1}{\epsilon}))$. We observe that as our error tolerance increases the time to insert an element decreases, but at the cost of allowing more elements to be corrupted. In particular, as proven in Chazelle's original paper on soft heaps, if n elements are inserted into a soft heap, then at most $\epsilon \cdot n$ elements can be corrupted (and one can show that this bound is tight).

For the sake of concision, we will avoid delving into the mechanics of each soft heap operation but encourage the interested reader to consult Chazelle's paper for the details. One point we will briefly address is how elements are corrupted. Using a technique that Chazelle refers to as "carpooling," the soft heap will group elements together within a single node and each node will maintain a *ckey* value representing an upper bound on the keys contained inside. A key is corrupted if it belongs to a node with a higher ckey value than itself. When an extract-min operation is called, the node with the lowest ckey then returns an arbitrary element from it's internal list, which may not be the true minimum element, thereby introducing error. However, this procedure allows us to maintain a single pointer to the minimum ckey node and thereby perform extractions in $O(1)$ time, as desired.

Now that we have introduced soft heaps, let's turn our attention back to Chazelle's algorithm. Since the algorithm is highly nontrivial, we will not present the pseudocode or analyze the runtime here. Instead, we will discuss the feature of the algorithm most relevant to understanding the origin of the ideas in the Pettie-Ramachandran Algorithm. This idea is the notion of *contractible subgraphs.* One of Chazelle's many insights in this algorithm was the observation that if we can partition our input graph into a set of vertex-disjoint subgraphs of sufficiently small size. Then we can efficiently compute the MST on these subgraphs and then contract each of these partions into a single node to form a graph minor G' and then recursively compute the MST overall. The soft heap is used to build this hierarchy of sub-problems efficiently. In the Pettie-Ramachandran algorithm, we will see precisely this same idea resurface again with the additional insight that a little precomputation can drastically speed up the process of computing MSTs on these partitions. Chazelle also showed, by exploiting several graph theoretic properties of MSTs, that the edges corrupted by the soft heap will not affect the correctness of the algorithm. We will return to this idea when discussing the use of soft heaps in Pettie Ramachandran when we formally prove correctness of the Optimal MST algorithm.

4 The Optimal MST Algorithm

We are now ready to move on to Pettie and Ramachandran's optimal MST algorithm. To begin, let's briefly summarize the components of the algorithm that we have seen in at least some manifestation in earlier MST algorithms:

- Contractible subgraphs (Chazelle)
- Running Boruvka's algorithm for a fixed number of steps (Karger-Klein-Tarjan, Chazelle)
- An $O(m)$ time MST algorithm on dense graphs (Tarjan-Fredman)
- Soft heaps (Chazelle)

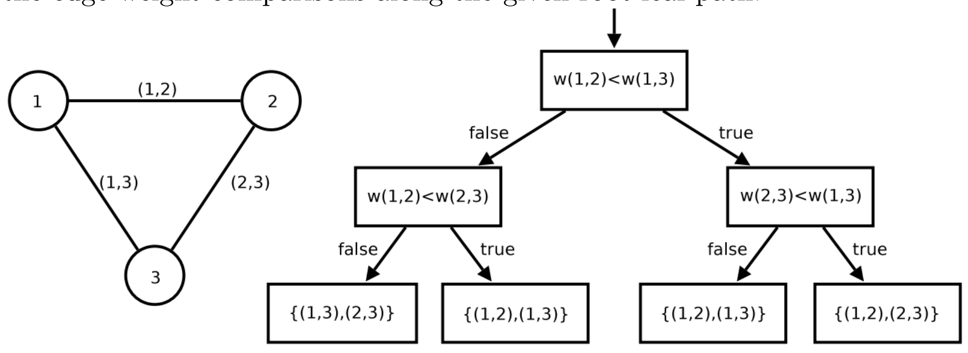
We will now turn our attention to introducing the key concept new to Pettie and Ramachandran's algorithm, namely *MST Decision Trees.* Afterwards, will present the pseudocode of the algorithm to illustrate it's relative simplicity to state. We will then step through each subroutine of the pseudocode, providing an intuitive explanation of each operation and formally proving key lemmas. Finally, we will combine these lemmas to present a full proof of correctness and follow that up with a proof of the optimality of the runtime.

4.1 MST Decision Trees

Recall from our discussion of Chazelle's algorithm in the previous section that one can find an MST by splitting the graph into smaller subgraphs and finding the MST within each of these components. A natural question that arises from this approach is how might we compute the MST on these sufficiently small subgraphs as fast as possible. Pettie and Ramachandran provide an answer that leads to a provably optimal runtime: precomputed MST decision trees.

Viewing algorithms as binary decision trees where the procedure has to make a series of yes/no decisions is a major concept within the field of computational complexity theory, particularly as a technique for establishing lower bounds. For example, decision tree models provide a very simple framework for proving the lower bound of $\Omega(n \log(n))$ on comparison-based sorting algorithms. A key insight of Pettie and Ramachandran is that the minimum spanning tree problem can be encoded as a binary decision tree. Specifically, an MST decision tree for a given graph G is a rooted binary tree where each internal node represents a comparison of two particular edge weights. Each root-leaf path thus represents a series of edge-weight comparisons and given these edge-weight comparisons, which tell us the relative orderings of the edges, we can determine the MST along that given path by brute-force search.

The following figure, from Claus Andersen's 2008 exposition of the optimal MST algorithm, illustrates the decision tree for a triangle where the leaf nodes contain the edges of the MST given the result of the edge-weight comparisons along the given root-leaf path.



At this point, we have shown how to map a graph into it's corresponding decision tree, but we must also address this question of how long computing such a decision tree, especially the brute-force computation of leaf node entries, will take. We will answer the question through the following theorem, which will then allow us to determine the optimal partition size for the Pettie-Ramachandran Algorithm.

Theorem 4.1. *Computing an MST decision tree on a graph with at most r vertices will take time $O(2^{2^{(r^2+o(r))}})$*

We defer the proof of this result to Andersen's paper. The result is highly accessible through reasoning about upper bounds on the depth of the decision tree. Using this theorem, we now prove an important corollary, which will determine the size of our partitions in the optimal MST algorithm.

Corollary 4.1.1. *For partitions with maximum size $r = \lceil \log^{(3)}(n) \rceil$ vertices, the decision trees for the optimal MST algorithm can be precomputed in $o(n)$ time.*

Proof. We will begin by first computing

$$\lim_{r\to\infty} \frac{(\log^{(3)}(r))^2}{\log^{(2)}(r)}$$

Letting $x = \log^{(2)}(r)$ the previous expression becomes

$$\lim_{r\to\infty} \frac{(\log(x))^2}{x} = 0$$

since the denominator increases exponentially faster than the numerator. Thus, we have shown that $(\log^{(3)}(r))^2 = o(\log^{(2)}(r))$ and that $\log^{(3)}(r)$ is $o(\log^{(2)}(r))$.

Now, by Theorem 4.1, we note that the time to compute a decision tree on a graph with at most r vertices is $O(2^{2^{(r^2+o(r))}})$. Plugging in $r = \lceil \log^{(3)}(n) \rceil$ we have a runtime of

$$2^{2^{((\log^{(3)}(n))^2+o(\log^{(3)}(n))}}$$

which, using our results that $(\log^{(3)}(r))^2 = o(\log^{(2)}(r))$ and that $\log^{(3)}(r)$ is $o(\log^{(2)}(r))$ simplifies to

$$2^{2^{o(\log^{(2)}(n))+o(\log^{(2)}(n))}} = 2^{2^{o(\log^{(2)}(n))}}$$

which simplifies to

$$2^{o(\log(n)} = o(n)$$

as desired.

□

In summary, we have shown that for a partition size of at most $r = \lceil \log^{(3)}(n) \rceil$ we can compute the MST decision tree on a graph of that size in linear time. Thus, we can precompute these decision trees at the start of our algorithm and speed up the process of computing the MSTs of our partitions. After presenting the algorithm and discussing the partition procedure, we will return to our decision tree analysis to show that computing the MST on a subgraph of size at most r given this precomputed decision tree will be proportional to the optimal number of edge-weight comparisons necessary to compute such an MST.

Algorithm 6 Pettie-Ramachandran Optimal MST

```
procedure OptimalMST(G = (V, E))
    if then$E = \emptyset$
        return $\emptyset$
    end if
    $r \leftarrow \lceil \log^{(3)} |V| \rceil$
    $M, C \leftarrow$ Partition$(G, r, \epsilon)$
    $F \leftarrow$ DecisionTree$(C)$
    $k \leftarrow C$
    $\{F_1, \dots, F_k\} \leftarrow F$
    $G_a \leftarrow G \setminus (F_1 \cup \dots \cup F_k) - M$
    $F_0 \leftarrow$ DenseCase$(G_a, |E|)$
    $G_b \leftarrow F_0 \cup F_1 \cup \dots F_k \cup M$
    $T', G_c \leftarrow$ Boruvka2$(G_b)$
    $T \leftarrow$ OptimalMST$(G_c)$
    return $T \cup T'$=
end procedure
```

4.2 The Full Algorithm

Algorithm 6 shows the pseudocode for the Optimal MST algorithm. Note that $G \setminus F$ denotes G, with the edges in F contracted.

4.3 The Partition Procedure

One of the key steps of this algorithm is the Parition procedure, which creates subgraphs of size at most r using a soft heap very much in the spirit of Chazelle's algorithm. We present the pseudocode for the partition procedure below and, afterwards, we will state a key lemma stating that the presence of corrupted edges due to the soft heap will not affect the correctness of our algorithm, allowing us, in effect, to have our cake and eat it too in benefiting from the fast extraction times of the soft heap without suffering a loss of absolute correctness. We will not present a proof of the lemma here, but the full details are available in Claus Andersen's survey.
Now we turn our attention to the key lemma. As an additional definition, a subgraph is *DJP contractible* it is the subgraph induced by running Prim's algorithm for a certain number of steps

Lemma 4.2. *Let M be a subset of corrupted edges in a graph G. If C is a subgraph of G that is DJP contractible with respect to G and these corrupted edges, then $MSF(G)$ is a subset of $MSF(C) \cup MSF(G \setminus C - M_C) \cup M_C$ where $G \setminus C$ denotes the contraction of the edges C into G*

Remarkably, this theorem shows that we can simply reintroduce our corrupted edges into the graph without losing any edges of our MSF and thereby proceed with our algorithm without losing any guarantees on correctness.

Algorithm 7 Partition Subroutine

```
procedure PARTITION($G, r, \epsilon$)
    Mark all vertices in the graph as initially "alive"
    $M_c \leftarrow \emptyset$                              ▷ $M_c$ represents the set of corrupted edges
    $i \leftarrow 0$
    while there is a live vertex do
        i++
        Let $V_i = \{v\}$ for some arbitary live vertex $v$
        Create a soft heap with error tolerance $\epsilon$ consisting of $v$'s edges
        while all vertices in $V_i$ are live and $|V_i| \leq r$ do
            Repeat:
            Extract the minimum weight edge $(x, y)$ from the soft heap      ▷ Assume $x \in V_i$
            if $y \in V_i$ then
                Break out of this loop
            end if
            $V_i \leftarrow V_i \cup \{y\}$
            If $y$ is live then insert each of $y$'s edges into the soft heap
        end while
        Set all vertices in $V_i$ to be dead
        Let $M_{V_i}$ be the set of corrupted edges with an endpoint in $V_i$
        $M_c \leftarrow M_c \cup M_{V_i}$
        $G \leftarrow G - M_{V_i}$
        Dismantle the soft heap
    end while
    Return the set of partitions $C_1, \ldots C_k$
end procedure
```

4.4 Using Decision Trees

Thanks to the algorithm's legwork up-front in precomputing optimal MST decision trees of size at most r, this step will run as efficiently as possible, using the optimal number of edge-weight comparisons. Since the decision tree is optimal and hence of minimum height, we do not have to do more work than the bare minimum necessary. In this step of the algorithm, we begin at the root node of the decision tree and traverse down the tree, going either left or right according to the results of the binary wedge-weight comparisons. Once we arrive at a leaf, we can then determine the MST for this partition and thus return it using the minimum number of edge weight comparisons required. In keeping with the notation of the authors' original paper, let $T^\star(m,n)$ denote the minimum number of edge-weight comparisons to compute an MST on a graph with m edges and n nodes.

4.5 Dense Case

Let's now turn our attention to the Dense Case portion of the algorithm. In order to preserve our time bounds, we require this step to run in linear time, which naturally leads us to use the Tarjan-Fredman MST algorithm, which runs in liner time on sufficiently dense graphs. We will now prove that the graph formed from our algorithm by contracting our partitioned subgraphs is sufficiently dense such that Tarjan-Fredman will run in linear time on each call to Dense Case.

The key component of the Tarjan-Fredman approach, as we discussed in Section 3, is to limit the number of elements than can be enqueued into a Fibonacci heap. In Tarjan and Fredman's original paper, they compute the bound k on the number of elements in a Fibonacci heap as $2^{\frac{2m}{t}}$ where m is the original number of edges in the graph and t the number of vertices at the time of this call to the Dense Case algorithm. Moreover, Tarjan and Fredman show that a single pass of their algorithm will run in time $O(t\log(k)+m')$ where t is the number of vertices in the graph and m' the number of edges at the time of the pass. These key facts from Tarjan and Fredman will allow us to prove our critical theorem concerning the runtime of the Dense Case algorithm.

Theorem 4.3. *Let G be a connected graph where $m \geq n$. Let G' be the graph formed by contracting G during execution of our algorithm. Then, the running time of the Dense Case algorithm for G' is $O(m)$*

Proof. Let n' denote the number of vertices in G' and m' the number of edges. Since we create G' by contracting the partitions of G and we have $\log^{(3)}$ partitions, we note that $n' \leq n/\log^{(3)}$ and $m' \leq m$ since some edges might have been contracted. Now, we note that initially, the Tarjan-Fredman algorithm will set it's heap bound k to be $k = 2^{\frac{2m}{t}}$ which implies, using Tarjan and Fredman's original formula, that each pass of their algorithm will take time $O(t\log(2^{\frac{2m}{t}})+m') = O(m+m') = O(m)$

Now, we must bound the number of passes the Tarjan-Fredman algorithm can make. Using their original bound on the algorithm's passes, we have that the maximum number of passes is

$$\begin{aligned}
O(1)+\beta(m,n') &= O(1)+\min\{i|\log^{(i)}(n') \leq m/n'\} \\
&\leq O(1)+\min\{i|\log^{(i)}\left(\frac{n}{\log^{(3)}(n)}\right) \leq \frac{m}{n}\log^{(3)}(n)\} \\
&\leq O(1)+\min\{i|\log^{(i)}(n) \leq \log^{(3)}(n)\} \\
&= O(1)+3 = O(1)
\end{aligned}$$

Therefore, we will run the Tarjan-Fredman algorithm for a constant number of passes, meaning that the Dense Case will run in linear time for each recursive call, completing the proof.

□

4.6 Boruvka Steps

As described previously, a *Boruvka step* is one iteration of Boruvka's MST algorithm. We identify the minimum weight edge incident to each vertex of G, which forms a spanning forest F. We then contract the edges in F, forming the new graph $G \setminus F$, which continues with the rest of the algorithm. Each edge in F is guaranteed to be in the MST, by the cut property. Furthermore, each Boruvka step runs in $O(m)$ time.

4.7 Proof of Correctness

We start with a few definitions:

- If M is a subset of edges, then the graph formed by raising the weights of the edges in M by an arbitrary amount is denoted $G \Uparrow M$.
- If C is a subgraph and M is a set of edges, then let M_C denote the set of edges in M with exactly one endpoint in C.
- A subgraph C is DJP-contractible with respect to G if C corresponds to the subgraph induced by the tree formed at some iteration of the DJP algorithm on G.

To prove correctness, we first state a key technical lemma:

Lemma 4.4. *If M is a subset of edges of G and C is a subgraph of G that is DJP-contractible with respect to $G \Uparrow M$, then*

$$MSF(G) \subseteq MSF(C) \cup MSF(G \setminus C - M_C) \cup M_C$$

Now, we can use this lemma to prove that the Optimal MST algorithm is correct:

Theorem 4.5. *The Optimal MST algorithm returns a minimum spanning tree of G.*

Proof. The execution of the Optimal MST algorithm exactly splits G into C, which is DJP contractible with respect to $G \Uparrow M$ and the corrupted edges M. The algorithm removes the corrupted edges M_C and finds the MSF of C using decision trees, and then contracts these and finds the MSF of the resulting graph, which is exactly $G \setminus C - M_C$. So, the input to the Boruvka steps in every call of the Optimal MST algorithm is to the subgraph $MSF(C) \cup MSF(G \setminus C - M_C) \cup M_C$, which is guaranteed to contain the MST of G, by the key lemma above. We only add edges to the MST when they are selected and contracted by Boruvka's algorithm. Since the Boruvka steps are passed a superset of MST edges, by the correctness of Boruvka's algorithm, the Optimal MST algorithm will only select MST edges. We strictly reduce the input size at each recursive call and we reach a base case for sufficiently small sizes, the algorithm is guaranteed to terminate. Now, inductively assuming that the recursive call returns the correct MST on its smaller input, the resulting tree will span all edges, and we have only included edges guaranteed to be in the MST. Thus, we will return a valid MST. □

4.8 Proof of Runtime Optimality

Let's begin with a quick summary of the runtimes of each of the major operations on a single recursive call of the algorithm, as we previously proved in the preceding sections.

- Precomputing the optimal decision trees: $o(n)$
- Partition Procedure: $O(m\log(\frac{1}{\epsilon})) = O(m)$ for a fixed error tolerance, ϵ
- Computing MSTs on the partition subgraphs using decision trees: $O(T^{\star}(m,n))$
- Dense Case: $O(m)$
- Boruvka Steps: $O(m)$

Now, we will proceed to combine all of the elements together by defining a recurrence relation, which we will then solve to prove our optimal time bounds. Our proof will follow the argument laid out by Andersen with a bit more exposition within each step.

Our strategy for defining a recurrence relation will be to determine the amount that G_c is smaller than G since G_c is the graph fed into the subsequent recursive call of the algorithm. To begin our journey towards defining a recurrence relation, let's return to a result we stated earlier in our discussion of the Partition procedure. Recall that we can upper bound the number of corrupted edges with endpoints in distinct partitions as $|M| \leq 2\epsilon m$. If we choose the constant $\frac{1}{8}$ as in Pettie and Ramachandran's original paper, then this bound simplifies to $|M| \leq m/4$.

Now, we also note that if no edges are removed during the partition procedure, then $|F_0 \cup F_1 \cup \cdots \cup F_k| = n-1$ and since we never add edges during the partition step of our algorithm, we observe that $|F_0 \cup F_1 \cdots \cup F_k| < n$ which implies that $|F_0 \cup \cdots \cup F_k \cup M| = m_b \leq n + m/4$.

Now we have a bound on the number of edges of G_b in terms of n and m but we really want a bound on G_c since that is the graph we feed into the next recursive call. Therefore, we'll now work towards relating G_b and G_c. Our first observation towards this goal is that G_b is the graph input into the Boruvka steps. Since we do not remove any vertices from the original graph in constructing G_b, we note that $n_b = n$. As we argued earlier, each Boruvka step will decrease the number of vertices in the input graph by at least 1/2 and decrease the number of edges by at least that same amount. Therefore we can achieve a bound on m_c, the number of edges on G_c. In particular, we have that

$$m_c \leq m_b - \frac{n}{2} - \frac{n}{4} = m_b - \frac{3n}{4}$$

and plugging in our previous inequality this simplifies to

$$m_c \leq n + m/4 - \frac{3n}{4} = \frac{n}{4} + \frac{m}{4} \leq \frac{2m}{4} = \frac{m}{2}$$

Moreover, since two steps of Boruvka will decrease the number of vertices in the input graph by at least half, we also have that $n_c \leq \frac{n}{4}$ and we now see that there is a geometric reduction in the number of vertices and edges heading into the next recursive call.

Now we are ready to define our recurrence relation. Note that the one computational step that we have largely ignored in our analysis thus far is the decision tree complexity of computing an MST on each of our partitioned subgraphs C_i. As shown in Andersen's paper, we can reintroduce the optimal decision tree complexity term as well as the cost of the $O(m)$ time operations Partition, Dense Case, and Boruvka2 to define our recurrence relation as

$$T(m,n) \leq \sum_i c_1 T^\star(C_i) + T(\frac{m}{2}, \frac{n}{4}) + c_2 m$$

where c_1 and c_2 are two positive constant values.

At this point, we have successfully defined our recurrence relation and the challenge now shifts to solving it. In order to do so, we must examine further properties of our decision trees.

Let's now introduce a new definition. Let $C_1, C_2, \ldots, C_k$ be the subgraphs produced by the partition procedure. We will now define a new set H as

$$H = \bigcup_{i=1}^{k} C_i$$

and proceed to prove one of our three key lemmas leading up to solving the recurrence.

Lemma 4.6. $MSF(H) = \bigcup_{i=1}^{k} MSF(C_i)$

Proof. From the cycle property, it follows that the heaviest edge in each simple cycle of H is not in the minimum spanning forest of H while all other edges are in the MSF. Moreover, if every simple cycle of H were contained in a single partition C_i then again by the cycle property it follows that the heaviest edge of each cycle will not be included in the MSF of C_i while the other edges will. Thus, if we can show that every simple cycle of H is contained in a single partition, we will prove the claim. This latter result follows directly from our partition procedure. Since a partition C_i can only share at most one vertex with previously grown components, it is impossible for a simple cycle in H to span multiple components, completing the proof.

□

Before we proceed to the next lemma, let's first establish two important definitions about decision trees that will be necessary in stating the next results.

Definition 4.6.1. *An inter-component comparison is a decision tree comparison that compares the weights of two edges in different components C_i and C_j where $i \neq j$*

Definition 4.6.2. *An intra-component comparison is a decision tree comparison that compares the weights of two edges in the same component*

These definitions now lead us to the following lemma

Lemma 4.7. *There exists an optimal decision tree T for H that makes no inter-component comparisons.*

Proof. By our previous lemma, we know that $MSF(H) = \bigcup_{i=1}^{k} MSF(C_i)$. Therefore, if T is an optimal decision tree for H then it must be able to determine the MSF of each C_i. Moreover, we note from our Partition procedure that the MSF of each C_i is entirely self-contained and therefore can be computed using only intra-component comparisons. As a result, a decision tree for H consisting of only intra-component comparisons must exist and it must and, in addition, it must be optimal since one cannot correctly compute the MSFs of each C_i without the intra-component comparisons and adding any inter-component comparisons would be superfluous and only increase the tree height.

□

Now, we still need a way of relating our decision tree H to the minimum comparison function $T^\star$. The following lemma will achieve precisely this, but we must first introduce one more definition.

Definition 4.7.1. *A canonical decision tree for a subgraph H is the decision tree where all intra-C_i comparisons appear above all intra-C_{i+1} comparisons in the tree.*

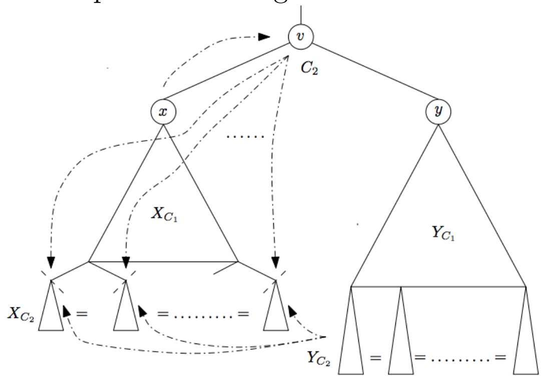
Lemma 4.8. $T^\star(H) = \sum_i T^\star(C_i)$

Proof. We will first show by construction that $T^\star(H) \leq \sum_i T^\star(C_i)$. After achieving this initial bound, we will show that there exists no optimal decision tree H with height strictly less than $\sum_i T^\star(C_i)$.

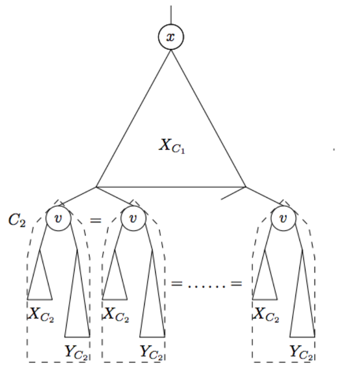
We can achieve our first bound by constructing a canonical decision tree for H by placing the decision tree for C_1 at the top of the tree, the decision trees for C_2 directly underneath the tree for C_1 and so forth. It is then easy to verify that this tree is indeed canonical and is a correct decision tree for H since a root-leaf path in this tree will represent the union of MSFs $C_1, \ldots C_k$ which is precisely the definition of $MSF(H)$. This construction does not rule out the possibility that there exists a decision tree for H of smaller height, but it does establish our desired initial bound of $T^\star(H) \leq \sum_i T^\star(C_i)$

To complete the proof, we must now show that there in fact cannot exist a decision tree for H of smaller height than the canonical decision tree. This proof will also be constructive. In particular, we can modify any given decision tree for H into the canonical decision tree by starting from the bottom of the tree and bubbling intra-C_i comparisons above intra-C_j comparisons where $i < j$. □

The details of the construction are a bit technical and we omit them from this paper for the sake of concision. Instead, we include a diagram from Andersen below illustrating this modification process of a decision tree into canonical form. The interested reader can consult Andersen's paper to see the proof in full rigor.



(a) The subtree rooted at v before the transformation.



(b) The canonical subtree after the transformation. The new identical C_2-subtrees are shown with dashed lines.

In addition, we now introduce three additional relations on MST decision trees that will be useful for our analysis. We omit the proofs here, but one can read Andersen for the details.

1. $m \leq T^\star(m,n)$
2. $T^\star(m,n) \leq \frac{1}{2}T^\star(2m,2n)$
3. $T^\star(m,n) \leq T^\star(m,n'$ where $n \leq n'$

Armed with the previous pieces, we are now ready to solve our recurrence relation and deduce the runtime of the algorithm

Theorem 4.9. *The Optimal MST Algorithm runs in time $O(T^\star(m,n))$*

Proof. Recall the recurrence relation we derived at the beginning of this section:

$$T(m,n) \leq \sum_i c_1 T^\star(C_i) + T(\frac{m}{2},\frac{n}{4}) + c_2 m$$

By the previous lemma, this is equivalent to

$$T(m,n) \leq c_1 T^\star(H) + T(\frac{m}{2},\frac{n}{4}) + c_2 m$$

and since the number of vertices in H is equal to n and the number of edges in H is at most m we can also write that

$$T(m,n) \leq c_1 T^\star(H) + T(\frac{m}{2},\frac{n}{4}) + c_2 m \leq c_1 T^\star(m,n) + T(\frac{m}{2},\frac{n}{4}) + c_2 m$$

Now, we will use a standard trick for solving recurrences by introducing a function $f(m,n)$ such that $T(m,n) \leq f(m,n)T^\star(m,n)$ we will then solve the recurrence relation in terms of this function f and go back and solve for f afterwards.

Introducing our new inequality into our previous relation we have

$$T(m,n) \leq c_1 T^\star(m,n) + f(m,n)T^\star(\frac{m}{2},\frac{n}{4}) + c_2 m$$

Simplifying this inequality with our 3 relations from earlier we have

$$\begin{aligned}
&c_1 T^\star(m,n) + f(m,n)T^\star(\frac{m}{2},\frac{n}{4}) + c_2 m \\
&\leq c_1 T^\star(m,n) + f(m,n)T^\star(\frac{m}{2},\frac{n}{4}) + 2c_2 T^\star(m,n) \\
&\leq c_1 T^\star(m,n) + \frac{1}{2}f(m,n)T^\star(m,\frac{n}{2}) + 2c_2 T^\star(m,n) \\
&\leq c_1 T^\star(m,n) + \frac{1}{2}f(m,n)T^\star(m,n) + 2c_2 T^\star(m,n) \\
&= T^\star(m,n)(c_1 + \frac{1}{2}f(m,n) + 2c_2)
\end{aligned}$$

By our choice of f, we know that $f(m,n) \geq c_1 + \frac{1}{2}f(m,n) + 2c_2$ and rearranging this inequality we have that $f(m,n) \geq 2c_1 + 4c_2$ which implies that we can set $f(m,n)$ to be a constant $c \geq 2c_1 + 4c_2$ so our previous inequality becomes

$$T(m,n) \leq T^\star(m,n)(c_1 + \frac{1}{2}c + 2c_2) = O(T^\star(m,n))$$

proving the claim.

□

We have now bounded the runtime of this algorithm by the minimum number of edge-weight comparisons necessary, but we do not know precisely how long these comparisons will take!

5 Implementation Details

In addition to our theoretical exploration of the Optimal MST algorithm, we had the lofty goal of implementing it from scratch, in its entirety, using nothing more than the standard C++ libraries. To accomplish this goal, we began by imlementing all the component data structures necessary for each component algorithm. The component data structures are the `FibonacciHeap`, `SoftHeap`, `DisjointSet`, `DisjointSetForest`, and `UndirectedGraph`. As we will discuss in more depth in the next section, there is no need to implement optimal decision trees in practice to maintain the runtime bounds of the algorithm. As our implementation can function only under the transdichotomous machine model, the graphs upon which the decision trees would theoretically operate are of size ≤ 3 and therefore the MST of such a graph can be trivially calculated in constant time. For a graph containing two nodes, we return the only edge because the graph is trivially its own MST. For a graph of 3 nodes, if it is complete, we iterate over the 3 edges, remove the edge with greatest weight, and return the remainning two edges. If the graph contains only two edges to begin, we trivially return them. We now give a brief overview of our implementation of each component data structure, each of which is implemented as a generic library data structure with a template interface.

5.1 `DisjointSet, DisjointSetForest, and UndirectedGraph`

We implemented disjoint sets using union find and path compression in order to support edge contraction in $\alpha(n)$ amortized time. Again assuming the transdichotomous machine model, this implementation of edge contraction introduces only an amortized constant factor to the runtime of component algorithms which utilize edge contraction. Our `UndirectedGraph` and `DisjointSetForest` implementations both represent a node's set of edges as a hash map mapping their end node to their weight. The list of nodes itself is represented as a hash map mapping each node to its map of edges. This representation allows amortized constant time access on any single edge as well as constant time to return a node's entire edge set. Iterators are also provided for unordered iteration on the list of nodes. Since our graph types are undirected, every edge is represented by a bidirectional mapping. The implementations of undirected graphs and disjoint set forests differ in that a `DisjointSetForest` represents a node as a `DisjointSet` containing that node while an `UndirectedGraph` represents a node as an integer key. The disjoint set forest also enforces the invariant that only a single edge may connect two `DisjointSet` nodes and that all edges connect only the parents of the disjoint sets. This is necessary to suppor the `contractEdge` operation, which removes an edge between two nodes, then unions the nodes so that all edges connected to either node are now connected to the resulting supernode. This operation is necessary for Boruvka and after the partition step of the main algorithm.

5.2 `FibonacciHeap`

The Fibonacci heap is necessary for the implementation of the Dense Case algorithm, which requires the decrease key operation. Since this data structure was covered extensively by the course, we will refrain from discussing it here in great depth. `FibonacciHeap` is a template implementation of the Fibonacci heap data structure, which supports all operations in amortized $O(1)$ time, except for `extractMin`, which operates in amortized $O(\log n)$ time as proven in class. The interface could perhaps use a bit more work as it leaves more functionality to the client program than is desireable. In particular, the client is responsible for freeing any returned entries and a new heap-allocated `FibonacciHeap` is returned by each `meld` operation, which the client must later free, along with the two instances which were passed into the `meld` function and destructively modified.

5.3 `SoftHeap`

The soft heap is necessary for the critical Partition procedure of the Optimal MST algorithm. Much of our time spent coding, we spent on attempting to implement `SoftHeap` correctly, and to modify it in such a way that it would could be used to efficiently implement the Partition procedure. Similar to `FibonacciHeap`, `SoftHeap` was implemented as a template version of the soft heap data structure as designed by Kaplan and Zwick.

This design differs from the original design of the soft heap as presented by Chezelle in that it uses binary trees to store elements rather than binomial trees. The overall structure is quite similar in form to that of a binomial heap, in which there is no more than a single tree of a given rank present in the root list at any given time. Each node in the tree is associated with a list of elements, and a *ckey* value which represents an upper bound on the keys of all elements stored in the node's list. Any element in the list with original key lower than *ckey* is considered 'corrupted', and any one of the elements in the list may be returned by `extract_min` if the node has the lowest *ckey* in the heap. Because of this, there is some degree of error associated with whether or not `extract_min` will return the actual minimum element. As discussed earlier, this error manifests as the number of 'corrupted' entries present in the heap at any given time, which is bounded to ϵn, given parameter ϵ on construction of the heap. Within the heap itself, ϵ is represented as the value r, which is obtained from the equation $r = \lceil \log_2 \frac{1}{\epsilon} \rceil + 5$ and evaluates to 8 for the use of `SoftHeap` by the Optimal MST algorithm, corresponding to a parameter $\epsilon = \frac{1}{8}$.

The design difference of representing each queue as a binary tree rather than a binomial tree makes the implementation of the `sift` operation, which is key in providing the low amortized runtime required of the soft heap operations, significantly simpler. In this simpler version of the soft heap, `sift` is implemented by simply ensuring that the left child of the node being sifted is the child with the lower *ckey* of the two children, then concatenating all entries associated with the left child onto the list of the node being sifted, thereby potentially corrupting the entries in the parent node's list.

Since the Optimal MST algorithm requires knowledge of a particular subset of all edges which have been 'corrupted', we made a structural modification to `SoftHeap` which allows it to return the set of all corrupted entries in constant time. To do this we store a linked list of corrupted entries as a member of `SoftHeap` and return it to the client program on request via the `getCorrupted` method. While this is certainly far from ideal coding practice, particularly since we made the `SoftHeap` itself responsible for deallocating memory associated with this list, a linked list supports constant time concatenation during a meld operation. Additionally, the cost of iterating over the returned list once, either to determine the membership of some set critical set of edges (some M_{V_i}) in the list, or to store them into a set or map for later constant time lookup, can be charged to the initial addition of the edge to the soft heap. This is possible because our procedure for creating the list of corrupted edges ensures that no edge is added more than once.

While we have not yet determined whether our corrupted list maintenance implementation (or indeed our implementation of `SoftHeap` itself) is entirely correct, we will endeavor to provide some intuition as to why our procedure for maintaining the list of corrupted items should work as expected. Since each element is initially inserted into a node with only a single member in its element list, we know that at this point the element cannot have been corrupted, because its key still corresponds to its node's *ckey*. We then note that an element only becomes corrupted when it becomes a member of a list of size > 1, and its key is no longer equal to its node's *ckey*. The only time this happens is during the `sift` operation, during which the list member elements of a node's left child are added to the list of the parent node, and the *ckey* is updated to that of the left child. The first time additional elements are sifted up into a node which had previously only ever contained the element in question, that element becomes corrupted, and it is at this point we add it to the list. We then fix the concatenate operation such that the list of the left child, which has been sifted up, is always prepended to the parent list. In this way we ensure that if there is an uncorrupted element present in the node's list, it will always be the first element. In order to track whether the first element is an uncorrupted element, and to keep track of it until it becomes corrupted, each node also stores a pointer *ckeyEntry* to it. When another node's elements are sifted up, this pointer is replaced by the value of that other node's *ckeyEntry* pointer, just as the parent node's *ckey* is updated to the value of the other node's *ckey*. If an element is ever removed from the list by an `extract_min` operation, we return the element at the front of the list and clear *ckeyEntry*, because we now know this list contains only corrupted elements, if any.

5.4 Implementation Progress

Unfortunately, as of the completion of this paper, we have not completed the implementation of the Optimal MST algorithm which we set out to create. The requisite data structures have been implemented completely to the extent described and we have implemented some of the subroutines of the algorithm, such as Prim's algorithm (DJP). We are still working on ways to ensure the correctness of `SoftHeap` as, by its nature, it has some potential for uncertainty in both its output and which edges it corrupts. We are also still working to ensure that our novel method for maintaining the list of corrupted nodes is in fact correct as we have not developed a formal proof of correctness for it. Hopefully we will be able to implement the remainder of the algorithm in short order, for the sake of our own interest in it if not for the completeness of our original goals for this assignment. Our code as it stands can be found on our public GitHub repository: https://github.com/ejdb00/optimal-mst.

6 Finding an Upper Bound on the Runtime

6.1 Restricting the number of vertices, Pettie-Ramachandran runs in $O(m)$ time

As we noted in our implementation strategy, we did not need to explicitly precompute decision trees since $\log\log\log(n) \leq 3$ for $n \leq 2^{256}$ which holds for all inputs on which we could test based on the transdichotomous machine model. Since our machines support only a 64-bit machine word, the transdichotomous machine model says that the largest structure we can operate on, in practice, is of size at most 2^{64}. As a result, with only at most 3 nodes in a single partition, we can do a scan of the edges and compute a minimum spanning tree in linear time, which is within our overall time bounds.

Motivated by this optimization we made to our algorithm in practice, we thought further about potential upper bounds on the runtime of the Pettie-Ramachandran algorithm on graphs of restricted sizes and now present the following theorem:

Theorem 6.1. *If $n \leq 2^{2^{16}}$, then the Pettie-Ramachandran algorithm will run in $O(m)$ time.*

Proof. As we observed in our runtime analysis of Pettie-Ramachandran in Section 4, each step of the algorithm will run in $O(m)$ time with the exception of the step of brute-force searching for the MST of a partitioned subgraph using decision trees. Let r denote the maximum size of a partition in our algorithm and we recall that we compute r as $r = \lceil \log\log\log(n) \rceil$. Now, by our initial assumption, we note that $n \leq 2^{2^{16}}$. Therefore, $r \leq 4$. If $r = 1$ or 2 then we can compute the MST on a partition of maximum size r in constant time since we either choose any existing edges or return immediately. If $r = 3$ or 4, then by Theorem 6.2, the resulting partitioned subgraphs must be planar. Therefore, by Lemma 6.3, we can compute the MST on these subgraphs in $O(n)$ time using Boruvka's algorithm. Since the decision tree computation runs in optimal time, it cannot be any slower than running Boruvka's algorithm on these subgraphs. Therefore, each step of the algorithm runs in linear time for a total runtime of $O(m + n) = O(m)$

□

The main idea behind the above proof is combining two key facts: that all graphs with four and fewer vertices are planar and that Boruvka's algorithm runs in linear time on planar graphs. These two properties in conjunction prove that we can compute MSTs on our partitions in linear time, which implies that the overall runtime of Pettie-Ramachandran is linear on these particular inputs. Below, we fill in the remaining details of the proof by presenting Theorem 6.2 and Lemma 6.3.

Theorem 6.2. *(Wagner 1937) A finite graph is planar if and only if it contains neither the complete graph on 5 vertices or the complete bipartite graph on 6 vertices, with three nodes on each side of the partition, as a minor*

Proof. $\Rightarrow$ The forward direction of this theorem follows directly from the fact that the minors of a planar graph must also be planar.

$\Leftarrow$ This direction of the biconditional follows as a consequence of Kuratowski's theorem, which states that a graph is planar if and only if it does not contain a subgraph that is a subdivision of the complete graph on 5 vertices or the complete bipartite graph on 6 vertices with three nodes on each side of the partition. We omit the proof of Kuratowski's theorem here for concision, but it is a result included in most standard graph theory textbooks.

□

We note that, as a consequence of Wagner's theorem, a graph with four or fewer vertices must be planar since it cannot contain a graph of five or six vertices as a minor. Since constructing a graph minor involves removing nodes or contracting and deleting edges in the original graph, it is impossible to have a graph minor with more nodes than the original graph.

Lemma 6.3. *Boruvka's Algorithm runs in $\Theta(n)$ time on planar graphs*

Proof. We first note that in a planar graph $m \leq 3n - 6$. We don't present a proof of this result in this paper, but it follows immediately from Euler's formula for planar graphs. Therefore, in planar graphs $O(n) = O(m)$. Therefore, each iteration of Boruvka's algorithm will run in $O(n)$ time since in the worst case we scan all of the edges. Now, we observe that in every iteration of Boruvka's algorithm, the number of supernodes available in the next recursive call decreases by at least 1/2. This gives rise to the recurrence relation $T(n) \leq T(\frac{n}{2}) + O(n)$ which by the Master Theorem gives us a bound of $\Theta(n)$ on the runtime of the algorithm, completing the proof.

□

Combining the three previous results together, we have shown that the Pettie-Ramachandran algorithm will run in linear time on all graphs with at most $2^{2^{16}}$ vertices. Though, we were unable to prove the runtime of the algorithm in general, we believe that this bound is a reasonably non-trivial one since any graph in practice will have fewer than $2^{2^{16}}$ vertices.

7 Performance on Random Graphs

We will now show that on the vast majority of graphs, *regardless of edge weights*, the optimal algorithm runs in linear ($O(m)$) time. This improves on the bounds of and Karger, Klein, and Tarjan's randomized MST algorithm that runs in $O(m)$ time with high probability, but can be shown to take $\Omega(n \log n)$ time with adversarially selected edge weights. Along the way, we will see beautiful applications of ideas from stochastic processes as well as earlier key results, and will fill in some details in the proofs of the results in the original paper.

We start with a definition: the Erdös-Renyi random graph model, $G_{n,m}$, samples each of the $\binom{\binom{n}{2}}{2}$ graphs on n vertices and m edges with equal probability; The other Erdös-Renyi random graph model, $G_{n,p}$, inclues each of the $\binom{n}{2}$ possible edges independently with probability p. Equivalently, each graph on n vertices and m edges is sampled with probability $p^m(1-p)^{\binom{n}{2}-m}$.

The theorem we want to prove is as follows:

Theorem 7.1. *The algorithm finds the MST in $O(m)$ time with probability*

1. $1 - \exp(-\Omega(m/\alpha^2))$ *for a graph drawn randomly from* $G_{n,m}$
2. $1 - \exp(-\Omega(pn^2/\alpha^2))$ *for a graph drawn randomly from* $G_{n,p}$

where $\alpha = \alpha(m,n)$ is the inverse Ackermann function.

We will prove the theorem in two steps. At a high level, the first is to exhibit that if the graph G satisfies a certain condition, then the algorithm will run in $O(m)$ time. The second step is to show that this condition is satisfied on random graphs with high probability. The condition will be based on the following definitions.

Define the *excess* of a subgraph H as $|E(H)| - |F(H)|$, where $F(H)$ is any spanning forest of H. Running the Partition procedure results in components of size at most $k \leq \log^{(3)} n$. Define $f(G)$ to be the maximum excess of the subgraph induced by the intracomponent edges over any possible execution of the Partition procedure. Intuitively, $f(G)$ is the number of "bad" edges inside the components. If $f(G)$ is small, then the resulting recursive call will be smaller, and thus the runtime will be faster. More concretely, we have the following lemma:

Lemma 7.2. *If $f(G) \leq m/\alpha$, then the algorithm runs in $O(m)$ time.*

Proof. After we have run the Partition and DenseCase procedures, which take $O(m)$ time, we run Boruvka steps, which only contract edges that are guaranteed to be MST edges. Thus, Boruvka steps will never contract the $f(G) \leq m/\alpha$ "bad" edges. Furthermore, note that by ???, not only the number of vertices, but also the number of edges decrease by a constant factor at every pass. Denote this constant factor by which edges decrease c. Consider the state of the algorithm after $\log \alpha$ steps. There are at most m total intracomponent edges initially. After, the number of intracomponent edges is at most

$$\frac{m}{c^{\log_c \alpha}} = \frac{m}{\alpha}$$

Thus, the total number of intracomponent edges is at most $2m/\alpha$. Now, we have run $O(\log \alpha)$ Boruvka steps and each Boruvka step reduces the number of nodes by a factor of 2, so the number of nodes remaining is $O(n/\alpha)$. Since the graph is connected, this is $O(m/\alpha)$. These Boruvka steps run in time $O(m)$. To see this, note that the first set of Boruvka steps run in $O(m)$. After the number of edges is reduced by a factor of c, the next set run in time $O(m/c)$, and so on. Thus, the total time spent on Boruvka steps is

$$O\left(m + \frac{m}{c} + \frac{m}{c^2} + \cdots + \frac{m}{c^{\log_c \alpha - 1}}\right) = O(m)$$

Now, the total number of edges is at most $2m/\alpha + O(m/\alpha) = O(m/\alpha)$. But, at this point, DJP will solve the MST problem on this graph in time $O(\alpha \cdot \frac{m}{\alpha}) = O(m)$. Thus, the total runtime is $O(m)$. □

We have shown that if the condition above holds, the algorithm runs in $O(m)$ time. Now, we just need to show that this condition holds with high probability. For this, we will need some machinery from stochastic processes.

A *martingale* is a sequence of random variables $Y_0, \ldots, Y_m$ such that $E[Y_i \mid Y_{i-1}] = Y_{i-1}$ for $0 < i \leq m$. Intuitively, a martingale is a sequence of random variables over time where, in expectation, at each time step, it remains the same as in the previous time step.

Erdös and Renyi showed that $G_{m,n}$ can be generated by a simple stochastic process: at each time step, we pick a random edge that we haven't already picked, and add it to the graph. More formally, we start with the empty graph on n nodes G_0 and at each time step i, we pick a random edge X_i such that $X_i \neq X_j$ for $j < i$ and let G_i be the graph containing all the edges from X_1 through x_i; that is $G_i = G_{i-1} \cup \{X_i\}$.

So, we can generate the random graphs we care about via a simple stochastic process but the amazing part is that we can use this process to define the *edge-addition martingale*, which allows us to use this same generative process for $G_{m,n}$ to reason about arbitrary graph theoretic functions on $G_{m,n}$!

Lemma 7.3. *The edge-addition martingale, $g_E(G_i) = E[g(G_m) \mid G_i]$ for $0 \leq i \leq m$ is a martingale, where g is any graph theoretic function. As before G_0 is the empty graph on n nodes, and G_i is generated as $G_i = G_{i-1} \cup \{X_i\}$ by choosing a random new edge X_i.*

Before we prove the lemma, we note that the beauty of this construction is that as time progresses, the martingale reveals progressively more about $g(G_m)$. We begin with no information about G_m, and the value of the martingale is just $E[g(G_m)]$. As we get more information in the form of X_i's, we learn more until at the end, we have complete information about G_m and thus $g(G_m)$.

Proof. We will show directly that given G_{i-1}, $E[g_E(G_i)] = g_E(G_{i-1})$. Taking the convention of the original paper, let X_i^j denote $\{X_i, \ldots, X_j\}$. Then,

$$\begin{aligned}
E[g_E(G_i)] &= E[E[g(G_m) \mid G_i]] \\
&= E\left[\sum_{x_{i+1}^m} g(G_{i-1} \cup x_i^m) \Pr[X_{i+1}^m = x_{i+1}^m \mid G_{i-1}, X_i = x_i]\right] \\
&= \sum_{x_i} \left(\sum_{x_{i+1}^m} g(G_{i-1} \cup x_i^m) \Pr[X_{i+1}^m = x_{i+1}^m \mid G_{i-1}, X_i = x_i]\right) \Pr[X_i = x_i | G_{i-1}] \\
&= \sum_{x_i^m} g(G_{i-1} \cup x_i^m) \Pr[X_{i+1}^m = x_{i+1}^m \mid G_{i-1}, X_i = x_i] \Pr[X_i = x_i | G_{i-1}] \\
&= \sum_{x_i^m} g(G_{i-1} \cup x_i^m) \Pr[X_i^m = x_i^m \mid G_{i-1}] \\
&= E[g(G_m) \mid G_{i=1}] \\
&= g_E(G_{i-1})
\end{aligned}$$

□

So, we now know that for any graph theoretic function on $G_{m,n}$, our estimate of the function tends to stay the same as we get each new piece of information about the graph. We will use $f(G)$ as our graph theoretic function and our goal will be to show that with high probability, our estimate in the beginning is not too far off using a powerful tail bound called Azuma's inequality. But to apply Azuma's inequality, we need to show that the value of the martingale doesn't fluctuate too much.

The following is a technical result. We encourage the reader to refer to the original paper for details.

Lemma 7.4. *For the edge-addition martingale $g_E(G_i)$ defined above, if $|g(G) - g(G')| \leq 1$ for any pair of graphs of the form $G = H \cup \{e\}$ and $G' = H \cup \{e'\}$ for some graph H, then $|g_E(G_i) - g_E(G_{i-1})| \leq 1$ for $0 < i \leq m$.*

We now show that the lemma above applies to $f(G)$, which shows that the edge-addition martingale for f, $f_E(G_i) = E[f(G_m) \mid G_i]$ does not fluctuate too much between consecutive time steps.

Lemma 7.5. *$|f(G) - f(G')| \leq 1$ for any pair of graphs of the form $G = H \cup \{e\}$ and $G' = H \cup \{e'\}$ for some graph H.*

Proof. By contradiction, suppose that $f(G) - f(G') > 1$. Take the components returned by Partition that result in the maximum excess in G. For these components in G', since all the edges are the same except for e and e', every intracomponent edge in G remains an intracomponent edge in G' except for e, which is no longer in the graph. This reduces the excess from G to G' by at most 1, a contradiction. A similar argument holds to contradict that $f(G') - f(G) > 1$, so we must have that $|f(G) - f(G')| \leq 1$. □

This, in addition to the previous lemma, show that $|f_E(G_i) - f_E(G_{i-1})| \leq 1$. With one more technical lemma, we can use Azuma's inequality on f to prove the result.

Lemma 7.6. $f_E(G_0) = o(m/\alpha)$.

We can finally use Azuma's inequality, a tail bound in the same vein as Markov's inequality, Chebyshev's inequality, or the Chernoff bound.

Theorem 7.7. *Let $Y_0, \dots, Y_m$ be a martingale such that $|Y_i - Y_{i-1}| \le 1$ for $0 < i \le m$, Then, for any $\lambda > 0$, $\Pr[|Y_m - Y_0| > \lambda\sqrt{m}] < \exp(-\lambda^2/2)$.*

Using this result, we can prove the following result:

Lemma 7.8. *If G is a random graph drawn from $G_{n,m}$, then $\Pr[f(G) \le m/\alpha] \ge 1 - \exp(-\Omega(m/\alpha^2))$.*

Proof. By Azuma's inequality, $\Pr[|f_E(G_m) - f_E(G_0)| > \lambda\sqrt{m}] < \exp(-\lambda^2/2)$. Now, we set $\lambda = \frac{\sqrt{m}}{\alpha} - \frac{f_E(G_0)}{\sqrt{m}}$ and note that by Lemma 7.6, $\lambda = \Omega(\sqrt{m}a)$ Thus,

$$\Pr[f(G) - f_E(G_0) > m/\alpha - f_E(G_0)] = \Pr[f(G) > m/\alpha] < \exp(-\Omega(m/\alpha^2))$$

from which the result follows immediately. □

We have now shown the two key parts to prove Theorem 7.1. We present the final proof below.

Proof. For $G_{n,m}$, $\Pr[f(G) \le m/\alpha] \ge 1 - \exp(-\Omega(m/\alpha^2))$ by Lemma 7.8. Furthermore, if $f(G) \le m/\alpha$, the algorithm runs in $O(m)$ time, by Lemma 7.2. Thus, with probability $1 - \exp(-\Omega(m/\alpha^2))$, the algorithm runs in $O(m)$ time.

For $G_{n,p}$, we refer the reader to a similar proof in the original paper.

□

8 References

1. "An optimal minimum spanning tree algorithm." PhD diss., Aarhus Universitet, Datalogisk Institut, 2008.

2. Bondy, John Adrian, and Uppaluri Siva Ramachandra Murty. Graph theory with applications. Vol. 290. London: Macmillan, 1976.

3. Charikar, Moses, Kevin Chen, and Martin Farach-Colton. "Finding frequent items in data streams." Automata, languages and programming. Springer Berlin Heidelberg, 2002. 693-703.

4. Chazelle, Bernard. "A minimum spanning tree algorithm with inverse-Ackermann type complexity." Journal of the ACM (JACM) 47.6 (2000): 1028-1047.

5. Chazelle, Bernard. "The soft heap: an approximate priority queue with optimal error rate." Journal of the ACM (JACM) 47.6 (2000): 1012-1027.

6. Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.

7. Cormode, Graham, and S. Muthukrishnan. "An improved data stream summary: the count-min sketch and its applications." Journal of Algorithms 55.1 (2005): 58-75.

8. Eisner, Jason. "State-of-the-art algorithms for minimum spanning trees-a tutorial discussion." (1997).

9. Fredman, Michael L., and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms." Journal of the ACM (JACM) 34.3 (1987): 596-615.

10. Jarnk, Vojtech. "O jistm problmu minimlnm." Prca Moravsk Prrodovedeck Spolecnosti 6 (1930): 57-63.

11. Kaplan, Haim, and Uri Zwick. "A simpler implementation and analysis of Chazelle's Soft Heaps." Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, 2009.

12. Kleinberg, Jon, and Eva Tardos. Algorithm design. Pearson Education India, 2006.

13. Kruskal, Joseph B. "On the shortest spanning subtree of a graph and the traveling salesman problem." Proceedings of the American Mathematical society 7.1 (1956): 48-50.

14. Lei, Jinna. "Three minimum spanning tree algorithms." University of California, Berkeley (2010).

15. Neetil, Jaroslav, Eva Milkov, and Helena Neetilov. "Otakar Borvka on minimum spanning tree problem translation of both the 1926 papers, comments, history." Discrete Mathematics 233.1 (2001): 3-36.

16. Pettie, Seth. "Finding Minimum Spanning Trees in O (m (m, n)) Time." (1999).

17. Pettie, Seth, and Vijaya Ramachandran. "An optimal minimum spanning tree algorithm." Journal of the ACM (JACM) 49.1 (2002): 16-34.

18. van Lint, Jacobus Hendricus, and Richard Michael Wilson. A course in combinatorics. Cambridge university press, 2001.