Toward Deterministic Linear MST Algorithms: Structural Reductions, Soft Heaps, and Failed Paths — CMPS316 Project Report

Prof. Amer Abdo Mouawad[a] **and Mahmoud Yaman Seraj Alddin**[a]

[a]Department of Computer Science, American University of Beirut

The minimum spanning tree (MST) problem is one of the most well-studied and foundational problems in combinatorial optimization. Classical deterministic algorithms—such as those of Kruskal, Prim, and Borůvka—achieve running times between $O(m \log n)$ and $O(m + n \log n)$, depending on data structures. Chazelle's MST algorithm achieves $O(m\alpha(n))$ Chazelle (2000a) using soft heaps. This research project examines ideas that were not explored explicitly in known literature, in hopes of achieving a deterministic linear MST algorithm.

This report summarizes three major avenues of investigation: (i) an $O(m \log n)$ algorithm based on pruning and backwards elimination; (ii) a corruption-bounded exploration using soft sequence heaps; (iii) alternative decomposition attempts.

1. Fast Reverse-Delete

Kruskal's Reverse-Delete Kruskal (1956) algorithm runs in $O(m \log n (\log \log n)^3)$ and works by removing edges that do not disconnect the graph in decreasing order of weight. Fast Reverse-Delete, while correct, does not surpass classical bounds ($O(m \log n)$) and was one of the initial ideas (and the only idea that worked) in the research.

1.1 Pruning. Pruning repeatedly eliminates degree-1 vertices and safely adds their unique edges to the MST.

Prune()

```
1: Consider a queue $Q$ with all degree-1 vertices.
2: while $Q$ not empty do
3:     Remove $v$ from $Q$.
4:     if $\deg(v) = 0$ then continue
5:     Let $e = (v, u)$ be its only incident edge.
6:     Mark $e$ as used.
7:     Remove $v$ and $e$ from $G$.
8:     Set the parent of $v$ to $u$.
9:     Decrement $\deg(u)$; if $\deg(u) = 1$ then enqueue $u$.
```

1.2 Deleting. Edges are examined in decreasing weight. Remove an edge and then prune. A removal **is** allowed to further disconnect the graph.

Delete()

```
1: Initiate a stack $R$ containing removed edges.
2: Remove parallel edges from $G$, keeping only the minimum-weight edge between each vertex pair.
3: Bucket sort the edges in decreasing order.
4: PRUNE().
5: for each edge $e$ in sorted order do
6:     Enqueue $e$ to $R$.
7:     Remove $e$ from $G$.
8:     PRUNE().
9: Build a contraction forest $F$ using the parent assignment of nodes
10: while $R$ not empty do
11:     Remove $uv$ from $R$.
12:     $e' = F.parent(u)F.parent(v)$
13:     Add $e'$ to $G$
14: return .
```

1.3 Fast Reverse-Delete. Although conceptually elegant, this approach does not reduce asymptotic complexity; its repeated pruning steps magnify underlying structural costs.

Fast Reverse-Delete(G)

```
1: Initialize a queue $Q$ with all degree-1 vertices.
2: PRUNE().
3: Sort all edges in decreasing order.
4: Replace each weight by its rank in the sorted list.
5: while $|E(G)| \neq 0$ do
6:     DELETE().
7: return Edges marked as used.
```

2. Analysis of Fast Reverse-Delete

2.1 Correctness. The proof consists of two parts. First, it is proved that the edges that remain after the algorithm is applied form a spanning forest (tree if the graph G is initially connected). Second, it is proved that the spanning tree is of minimal weight.

Spanning Forest

Define $E_{removed}$ as the set of currently removed edges and E_{used} as the set of used edges. Consider a component C in G. Initially, all nodes with degree 1 are added to Q. Before every call to DELETE, all nodes in Q are pruned and any node that becomes a leaf due to pruning, is also added to Q and pruned. Hence after pruning, all nodes have degree $>= 2$. Define a node u being pruned by an edge $e = uv$ as collapsing u into v, mapping all removed edges of u into v ($\forall\, uw \in E_{removed} :\ uw \mapsto vw$), and using e.

Lemma 1 An edge e is either used once and removed indefinitely, removed temporarily and re-added, or removed indefinitely by the parallel edge removals.

Proof. Initially all loops and multi-edges are removed from G.
Once an edge is marked as *used*, it is added to E_{used} and removed indefinitely.
By definition, all previously removed edges by the DELETE function are mapped and re-added to G. This can create loops and multi-edges.
These multi-edges and loops are removed indefinitely at the beginning of the next iteration. □

Lemma 2 An edge $e = uv$ either prunes u or v but not both.

Proof. By **lemma 1**, an edge is either used once or never used $\implies$ a used edge will prune either one of its endpoints.
Consider a pendant edge $e = uv$. Without loss of generality, let u be enqueued before v, when u is pruned by e, deg(v) becomes 0. By definition of PRUNE, when v is popped from Q later on, it is not processed. □

Lemma 3 $|V(C)| - 1$ nodes in a component C are pruned.

Proof. Every vertex $u \in V(C)$ is either eventually pruned by an incident edge during some call to PRUNE(), or it remains as the unique surviving representative of C. All but one vertex of C must be pruned.

Suppose that, during a particular iteration of DELETE(), a vertex $u \in V(C)$ is not pruned. If u is the last remaining vertex of C, then no further action can be done. Otherwise, there must exist another vertex $v \in V(C)$ that is still present. In this case, when the removed edges are re-added they must reconnect the vertices by **lemma 1**. PRUNE() will be invoked again on a connected version of C.

Repeating this argument, as long as at least two vertices of C remain, at least one of them must be pruned in a subsequent iteration. Hence, the process continues until exactly one vertex of C remains unpruned. Therefore, PRUNE() removes $|V(C)| - 1$ vertices of C. □

Corollary 1 The algorithm terminates. Based on **lemma 3**, $|V(C)| - 1$ vertices are pruned, each by some unique edge (**lemma 2**) $\implies$ each component C collapses into a single node whose loops are eliminated by the parallel removals $\implies$ eventually all edges are removed indefinitely.

Corollary 2 Each component is reduced to a spanning tree. All components in G form a spanning forest.

Minimality

Lemma 4 Each node is pruned by its minimum edge.

Proof. Consider node u with degree 1, its only edge is its minimum edge.
Consider node v with degree $>= 2$. When an incident edge is removed, it must be greater than any other edge, otherwise it would have appeared later in the sorted edge list.

□

Lemma 5 Each used edge is part of the MST.

Proof. By the MST cut property, the minimum edge of a node must be part of the MST.
By **lemma 4**, every used edge is the minimum of some node. □

Lemma 6 Each iteration of DELETE eliminates at least $\lceil 2n/3 \rceil$ nodes

Proof. Consider the case where each node is part of at least 1 cycle.
The current iteration of the removals can remove all the edges connecting the cycle to some other cycle(s).
In that case the cycle l becomes isolated. Removing one edge from this cycle breaks it, resulting in all $|V(l)|$ nodes becoming a single node. The smallest size for l is 3.

□

The algorithm creates a spanning tree, the spanning tree is minimal, and the algorithm halts. This concludes the correctness proof of the algorithm.

2.2 Running Time. The algorithm can be shown to run in $O(m \log n)$ this bound is achieved as follows:

1. Initializing Q costs $O(n)$.
2. Sorting edges costs $O(m \log n)$.
3. Replacing weight by rank costs $O(m)$.
4. Each node is pruned at most once, pruning costs $O(1) \implies$ Amortized $O(1)$ for the PRUNE function.
5. Building the contraction forest F costs $O(n + m)$.
6. Removing parallel edges costs $O(n + m)$.
7. Bucket sort over edges with maximum value $W = m$ costs $O(m)$.
8. Enqueue the edge, removing the edge, and PRUNE all work in $O(1)$ or amortized $O(1) \implies$ iterating over the sorted edges is $O(m)$.
9. Adding back the edges is $O(m)$.
10. By **lemma 6**, each iteration of DELETE reduces the number of nodes by at least a third $\implies$ the DELETE loop is executed $\log_3 n = O(\log n)$ time.
11. Final running time is $O((n + m) \log n)$, but the $n \log n$ factor is usually omitted in the other MST algorithms $\therefore O(m \log n)$ □.

2. Soft Sequence Heaps and Corruption Bounds

Soft heaps accelerate operations by allowing controlled key corruption Chazelle (2000b). Let ε be the error parameter.

We consider soft sequence heaps, as they are simpler, have a better interface, and server the same purpose as Chazelle's soft heap Kaplan and Zwick (2013):

- INSERT runs in $O(r_0)$ where $r_0 = \lfloor \log(1/\varepsilon) \rfloor$.
- All other operations run in amortized $O(1)$.
- After N insertions, at most N/r_0 keys may become corrupted.
- Increasing r_0 does little than delay the inevitable. For a large enough N, there will always be corrupt elements.
- Extracted items can be flagged in $O(1)$ for corruption.
- Uncorrupt items are heapified.
- However, all implementations we encountered introduced corruption on removal. For example, we insert 100 items into the soft heap with $\varepsilon = 0.1$, we have 10 corrupt items. Pop an item, if it is corrupted, then the number of elements reduces to 99 and corruption becomes 9 and can bounce back to 10. Thus, all 100 extracted elements can be corrupted.

- All soft heaps have this limitation except a certain implementation in a master's thesis McCann (2023), which claims no additional corruption on removal.

The project tries attempted to exploit the property of McCann's soft heap to design a corruption-bounded BFS-like process.

3. Corruption-Bounded Exploration Attempt

Corruption-Bounded Exploration

```
1: if $G$ has no edges then
2:     return
3: Initialize a soft heap $\mathcal{H}$ with error $\varepsilon$.
4: Perform a Borůvka phase (reduces nodes to $\lfloor n/2 \rfloor$).
5: for each vertex $v$ do
6:     if $v$ is visited then
7:         continue
8:     Insert all neighbors of $v$ into $\mathcal{H}$.
9:     Mark $v$ as visited.
10:    while $\mathcal{H}$ is not empty do
11:        Extract item $x$ from $\mathcal{H}$.
12:        if $x$ is corrupted then
13:            continue
14:        Visit the vertex associated with $x$.
15:        Insert all neighbors of that vertex into $\mathcal{H}$.
```

3.1 The Algorithm. The intended recurrence is:

$$(n, m) \mapsto \left(\left\lfloor \frac{n}{2} \right\rfloor, \, \varepsilon(m - \lfloor n/2 \rfloor) + \left\lfloor \frac{n}{4} \right\rfloor - c \right).$$

This suggested geometric decay of n while corruption remained bounded. The algorithm is correct iff McCann's soft heap functions correctly. However, it turns out the data structure was broken and in fact such a data structure can never exist.

3.2 Why the Reasoning Fails. The argument implicitly relied on the incorrect assumption that corruption could remain below $(1 - \varepsilon)n$ under repeated heap operations. If this were true, one could sort in $o(n)$ time, contradicting established comparison-based lower bounds. To see this sorting method, consider the following algorithm given an array L of n arbitrary values:

QuickerSort(L)

```
1: Initialize McCann soft heap $S$.
2: Initialize array $UC$ of size $\log n$.
3: $i \leftarrow 0$.
4: while $|L| \neq 0$ do
5:     $S.insert(L)$.
6:     $UC[i] \leftarrow S.extract\_uncorrupt()$.
7:     $L \leftarrow S.extract\_uncorrupt()$.
8:     $i \leftarrow i + 1$.
9: Merge sorted lists in $UC$ using tournament tree in $O(n \log \log n)$
10: return Sorted array.
```

Consequently, the approach collapses: no data structure can provide a sorted proportion of the n items. There has to be a hidden logarithmic cost in McCann's soft heap or a broken assumption.

4. Alternative Directions

4.1 Bucketing. Attempts to group edges into $O(m/n)$-sized buckets fail because worst-case bucket-induced components can degenerate into trivial or overly large structures. However, we can know the number of edges that we must pick from each bucket (number of components formed after removing the bucket edges - 1).

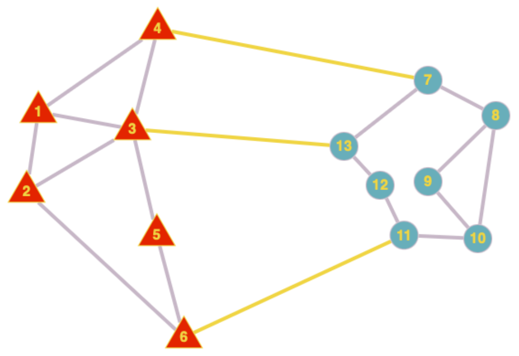

Fig. 1. Worst case example for bucketing, red triangles are individual nodes of the graph, blue ones are part of the component

4.2 Induced Subgraphs Contraction. Even when each local induced subgraph has k interior nodes and B boundary nodes (initially constant), recursive reductions cause k and B to grow toward $\Theta(n)$ unless a uniform decomposition exists—which is not generally possible.

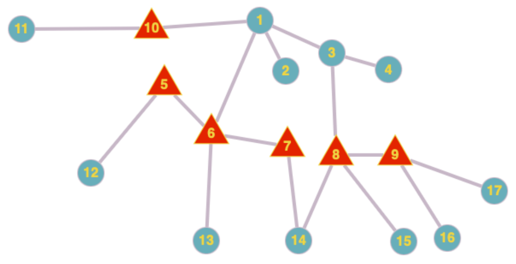

Fig. 2. Initial graph before contracting

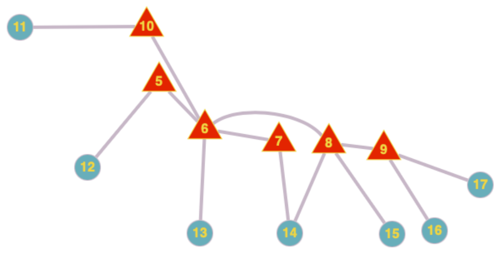

Fig. 3. Graph after contracting, we removed the nodes but added $B - 1$ edges in place

4.3 BFS Triangles and 3-Regular Reductions. Transforming graphs to 3-regular form does not reduce MST complexity: any general graph can be encoded as a 3-regular graph, and solving MST on the latter yields solutions on the former. Trying to apply the induce and reduce trick while keeping the degree constant at 3 doesn't work either; all reduced nodes must be re-added to bring the degrees back to 3-regular.

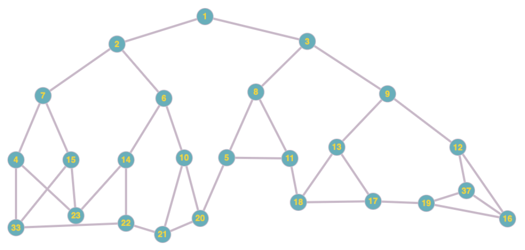

Fig. 4. Example 3-Regular graph after applying BFS and splitting an edge to get a node of degree 2 as the root

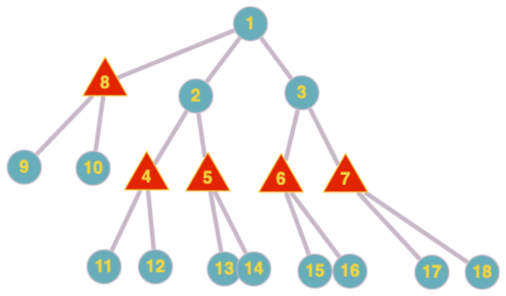

Fig. 5. Example 3-Regular graph before applying reduction trick

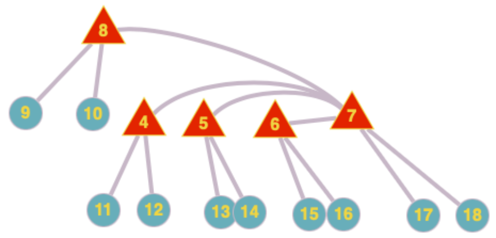

Fig. 6. Example 3-Regular graph after applying reduction trick case 1

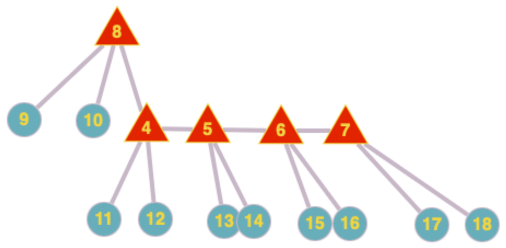

Fig. 7. Example 3-Regular graph after applying reduction trick case 2

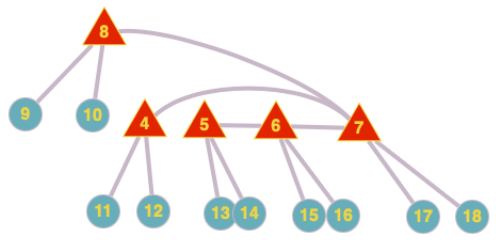

Fig. 8. Example 3-Regular graph after applying reduction trick case 3

Conclusion

This investigation documents several natural but ultimately unsuccessful strategies toward deterministic linear MST algorithms. Despite their failure, these attempts sharpen our understanding of structural pruning, soft-heap corruption, and limitations of local reductions. Future work may require insights from decision-tree complexity or fundamentally new deterministic analogues of randomized MST structures.

Note on implementation

The Fast Reverse-Delete algorithm was implemented in the Github repository linked in the bottom-left corner of the page.

References

Chazelle, B. (2000a). A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47(6):1028–1047.

Chazelle, B. (2000b). The soft heap: An approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027.

Kaplan, H. and Zwick, U. (2013). A simpler implementation and analysis of chazelle's soft heaps. *SIAM Journal on Computing*, 42(4):1660–1673.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50.

McCann, S. (2023). Soft heaps visualized. Master's thesis, Appalachian State University, Boone, North Carolina.