Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 26, no. 4, pp. 577–588 (2022)
DOI: 10.7155/jgaa.00609

A Simple and Efficient Algorithm for Finding Minimum Spanning Tree Replacement Edges

David A. Bader[1] *Paul Burkhardt*[2]

[1]Department of Data Science
New Jersey Institute of Technology
Newark, NJ 07102
[2]Research Directorate
National Security Agency
Fort Meade, MD 20755

Submitted:	December 2021	Reviewed:	November 2022
Revised:	November 2022	Accepted:	December 2022
Final:	December 2022	Published:	December 2022
Article type:	Regular paper	Communicated by:	Petra Mutzel

Abstract. Given an undirected, weighted graph, the minimum spanning tree (MST) is a tree that connects all of the vertices of the graph with minimum sum of edge weights. In real world applications, network designers often seek to quickly find a replacement edge for each edge in the MST. For example, when a traffic accident closes a road in a transportation network, or a line goes down in a communication network, the replacement edge may reconnect the MST at lowest cost. In the paper, we consider the case of finding the lowest cost replacement edge for each edge of the MST. A previous algorithm by Tarjan takes $\mathrm{O}(m\alpha(m,n))$ time and space, where $\alpha(m,n)$ is the inverse Ackermann's function. Given the MST and sorted non-tree edges, our algorithm is the first practical algorithm that runs in $\mathrm{O}(m+n)$ time and $\mathrm{O}(m+n)$ space to find all replacement edges. Additionally, since the most vital edge is the tree edge whose removal causes the highest cost, our algorithm finds it in linear time.

1 Introduction

Let $G = (V, E)$ be an undirected, weighted graph on $n = |V|$ vertices and $m = |E|$ edges, with weight function $w(e)$ for each edge $e \in E$. A minimum spanning tree $T = \mathrm{MST}(G)$ is a subset of $n - 1$ edges with the minimal sum of weights that connects the n vertices.

Research supported by NSF Grant 2109988.

E-mail addresses: bader@njit.edu (David A. Bader) pburkha@nsa.gov (Paul Burkhardt)

In real world applications the edges of the MST often represent roadways, transmission lines, and communication channels. When an edge deteriorates, for example, a traffic accident shuts a road or a link goes down, we wish to quickly find its *replacement edge* to maintain the MST. The replacement edge is the lightest weight edge that reconnects the MST. For example, Cattaneo et al. [2] maintain a minimum spanning tree for the graph of the Internet Autonomous Systems using dynamic graphs. Edges may be inserted or deleted, and a deletion of an MST edge triggers an expensive operation to find a replacement edge of lightest weight that reconnects the MST in $\mathrm{O}(m \log n)$ time from the non-tree edges, or $\mathrm{O}(m + n \log n)$ time when a cache is used to store partial results from previous delete operations.

In this paper, we consider the problem of efficiently finding the minimum cost replacement for all edges in the MST. Recomputing the MST for each of the original tree edges is clearly too costly. The problem is deceptively difficult. Each replacement edge must be a non-MST edge in a fundamental cycle with the obsolete MST edge. But there are $\mathrm{O}(m)$ unique cycles and each cycle can have $\mathrm{O}(n)$ MST edges so choosing the lightest non-tree edges as replacements requires careful planning to prevent repeatedly referencing the same MST edges. This and related problems for updating the MST have been studied extensively (e.g., [19, 3, 22, 5, 14, 10, 11]). The best algorithm is from 1979 due to Tarjan [22] and runs in $\mathrm{O}(m\alpha(m, n))$ time and space, where $\alpha(n, m)$ is the inverse Ackermann's function. But given edges sorted by weight, we show the problem can be solved in linear time and space using a surprisingly simple approach.

The main result of this paper is a simple and fast deterministic algorithm for the MST replacement edge problem. Given the minimum spanning tree and non-tree edges sorted by weight, our algorithm finds all replacement edges in $\mathrm{O}(m + n)$ time and $\mathrm{O}(m + n)$ space. Although it is known that these bounds are theoretically possible using pre-sorted edges, we give the first practical algorithm. Sorted edges come free if the MST is computed by Kruskal's algorithm. If the edge weights have fixed maximum value or bit width, then the edges can be sorted in linear time making our algorithm an asymptotic improvement over prior algorithms. Our algorithm is simple and does not require queues or computing the Lowest Common Ancestor (LCA). It assigns a pair of numbers to each vertex denoting the order in which they are first and last visited in a depth-first search (DFS) over the MST, and for each replacement it applies linear-time path compression using the static union tree of Gabow and Tarjan [7, 8]. All running times in this paper are deterministic worst-case under the Word-RAM model.

2 Related Work

The problem of updating the MST should a tree edge be deleted or its cost increased has been studied since the 1970s. In 1975 Spira and Pan [19] showed it takes $\mathrm{O}(n^2)$ time to find the replacement for one tree edge. Then in 1978 Chin and Houck [3] gave a $\mathrm{O}(n^2)$ time algorithm to find the replacements for all MST edges. This result was improved to $\mathrm{O}(m\alpha(m, n))$ time in 1979 by Tarjan [22] and remains the best deterministic runtime for the general case of unordered, arbitrary-weight edges. The algorithm due to Tarjan uses path compression that maintains balanced tree height, which does not benefit from having edges sorted by weight. It finds the replacement edges by evaluating paths in a specially constructed directed acyclic graph in which vertices correspond to tree and nontree edges, and combining operations on vertex labels. This directed acyclic graph has $\mathrm{O}(m\alpha(m, n))$ vertices and therefore Tarjan's algorithm for the MST replacement edge problem takes $\mathrm{O}(m\alpha(m, n))$ time and space. Our method is more efficient for the case in which edges are pre-sorted by weight, taking $\mathrm{O}(m + n)$ time and space.

The MST replacement edge problem can also be solved by MST sensitivity analysis, for which Tarjan [23] gave a $O(m\alpha(m,n))$ time algorithm in 1982. The MST sensitivity analysis determines the amount each edge weight can be perturbed without invalidating the MST. The sensitivity of a tree edge e is the weight of an edge f that is the minimum weight among the nontree edges that cross the cut induced by removing e, and therefore f is the replacement for e. In 1994 Booth and Westbrook [1] show for planar graphs that the MST sensitivity analysis and replacement edge problems can be solved in $O(n)$ time and space. They use a depth-first search ordering of vertices similar to our method, but explicitly compute the LCA for each edge. Their method maintains two LCA-ordered lists for each leaf vertex of the tree, and therefore sorting by edge weights does not improve their runtime. In 1996 Kooshesh and Crawford [14] proposed an algorithm for the MST replacement edge problem taking $O(\max(C_{\text{mst}}, n\log n))$ time, where C_{mst} is the cost of computing the minimum spanning tree. But their runtime is not efficient and can take $O(n\log n)$ time. Their approach uses similar ideas to that of [1] and therefore does not improve with sorted edges in advance.

It took nearly thirty years to improve Tarjan's MST sensitivity analysis result for general graphs when in 2005 Pettie [16, 17] improved the runtime to $O(m\log\alpha(m,n))$ time. A more tantalizing result is Pettie showed that the MST sensitivity analysis problem is no harder than solving the MST. This implies that the MST replacement edge problem can be solved in deterministic, linear-time given an MST algorithm with the same time complexity. But a deterministic, linear-time MST algorithm is still an open problem. Interestingly, Pettie and Ramachandran [18] gave an optimal MST algorithm. The 2002 Pettie-Ramachadran MST algorithm has the curious property that although it is provably optimal, the runtime is unknown but is between $O(m)$ and $O(m\alpha(m,n))$ time.[1] In the special case where edges are sorted by weight in advance, then the MST can be solved in deterministic linear time using the Fredman and Willard algorithm [6] by transforming real number weights to integers using relative ordering, e.g. the i^{th} ordered edge gets weight i. Then it follows from Pettie's reduction that the MST replacement edge problem can be solved in deterministic linear time, matching the same bounds as our algorithm. The important distinction is our algorithm is far simpler.

A related problem to MST replacement edges is that of maintaining the MST as edges are repeatedly updated, where an update means deletion, insertion, or weight change of an edge. Frederickson [5] gave an algorithm to maintain an MST with edge updates (deletion, insertion, or weight change) where each update takes $O(\sqrt{m})$ worst-case time, and sparsification makes the bound $O(\sqrt{n})$. Henzinger and King [10] gave an algorithm to maintain a minimum spanning forest with edge deletions or insertions; each update takes $O(\sqrt[3]{n}\log n)$ amortized time. Holm et al. [11] give an algorithm for maintaining a minimum spanning forest with edge deletions or insertions; each update takes $O(\log^4 n)$ amortized time. Recently, Hanauer *et al.* gave a survey of fully dynamic graph algorithms and discuss maintaining minimum spanning trees [9].

3 Algorithm

Given T and the remaining non-tree edges $E\backslash E_T$ sorted from lowest to highest weight, then Algorithm 1 finds all replacement edges for an MST in $O(m+n)$ time. Observe that each of the $m-n+1$ edges in $E\backslash E_T$ induces a fundamental cycle with the edges in T. Then for any MST edge there is a subset of cycles containing that edge, and the cycle induced by the lightest non-MST edge is the replacement for it. This follows from the *Cut Property* [4, c.f. Theorem 23.1] where the

[1]This is due to the decision tree complexity of the MST; the height of the tree is optimal but unknown.

lightest non-tree edge crossing a cut must be in the MST if some other edge in the induced cycle is removed. Our Algorithm 1 finds the lightest weight cycle for each tree edge but avoids repeatedly traversing these edges. Since replacement edges are found immediately after computing an MST, we can re-use the sorted edges from Kruskal's [15] MST algorithm.

The major steps of our approach are 1) assign the parent and the first and last visited numbers to each vertex according to depth-first search over T 2) traverse the fundamental cycle induced by each non-tree edge in order of ascending weight 3) compress paths to skip edges already assigned replacements. With $\mathrm{O}(m)$ non-tree edges and $\mathrm{O}(n)$ edges in each cycle, the naïve approach has $\Omega(mn)$ time complexity. This paper introduces an algorithm that reduces the cost to $\mathrm{O}(m+n)$ time by a novel use of a special case of the disjoint set union data structure. We use the disjoint sets for fast path compression based on the Gabow-Tarjan static union tree method [7, 8].

Algorithm 1 first roots the MST at an arbitrary vertex v_r and initializes a parent array P. Next, each vertex $v \in V$ is visited during a depth-first search traversal from the root, and the value of $P[v]$ is set to its respective parent vertex from the traversal order. For the root v_r, its parent $P[v_r]$ is set to v_r. Our approach uses another innovation that alleviates the need to find the lowest common ancestor vertex in the rooted MST for each non-tree edge. To do so, we use a pair of vertex-based values, $\mathrm{IN}[v]$ and $\mathrm{OUT}[v]$, which are assigned as follows. During the depth-first traversal of the rooted tree, a counter is incremented for each step in the traversal (up or down edges). When the traversal visits v the first time during a traversal down an edge, $\mathrm{IN}[v]$ is assigned the current counter value. When the traversal backtracks up an edge from vertex v, $\mathrm{OUT}[v]$ is then assigned the current counter value. With a minor modification to our algorithm we can also employ the conventional pre- and post-order numbers from depth-first search to detect the LCA. Our ordering is chosen for convenience because it requires only a single increment on the visit order, which simplifies tracking the traversal as an increasing sequence of visit numbers. For example, using $\mathrm{pre}(v)$, $\mathrm{post}(v)$ to denote respectively the pre- and post-order of a vertex v, then when identifying if s is an ancestor of t we can replace the conjunction $\mathrm{pre}(s) < \mathrm{pre}(t)$ and $\mathrm{post}(t) < \mathrm{post}(s)$ with the straightforward inequality sequence $\mathrm{IN}[s] < \mathrm{IN}[t] < \mathrm{OUT}[t] < \mathrm{OUT}[s]$.

The $m-n+1$ remaining edges in $E \backslash E_T$ are scanned in ascending order by weight, inspecting the tree edges in each corresponding fundamental cycle. In this order, the first time a tree edge e is included in a fundamental cycle, its replacement R_e is set to the non-tree edge from that cycle. As we will describe, the disjoint sets provide subpath compression as replacement edges are assigned to MST edges. In Algorithm 1, the disjoint sets are updated through the **makeset**, **find**, and **link** functions.

For each non-tree edge (s,t), if vertex t is a descendant of s, (if and only if $\mathrm{IN}[s] < \mathrm{IN}[t] < \mathrm{OUT}[t] < \mathrm{OUT}[s]$), we make a single PathLabel call for the edges from t up to s. Since $\mathrm{IN}[t] < \mathrm{OUT}[t]$, we simplify this check in Algorithm 1, line 2, to $\mathrm{IN}[s] < \mathrm{IN}[t] < \mathrm{OUT}[s]$.

Otherwise, two calls are made to PathLabel, corresponding to inspecting the *left* and *right* paths of the cycle from s and t, respectively, that would meet at the LCA of s and t in the tree. We assume, without loss of generality, that s is visited in the depth-first search traversal before t. Let's call $z = \mathrm{LCA}[s,t]$. It is useful to use z in describing the approach, yet we never actually need to find the LCA z. We know $\mathrm{IN}[z] < \mathrm{IN}[s] < \mathrm{OUT}[s] < \mathrm{IN}[t] < \mathrm{OUT}[t] < \mathrm{OUT}[z]$ by definition of the depth-first traversal. As illustrated in Figure 1, consider a vertex w that lies on the path from s to the root v_r. Vertex w must either lie on the path from s to the LCA z (where $\mathrm{OUT}[w] < \mathrm{IN}[t]$), or from z to the root v_r (where $\mathrm{OUT}[w] > \mathrm{IN}[t]$). We use this fact to detect when PathLabel reaches the LCA without computing it.

As mentioned earlier, the disjoint sets provide subpath compression as replacement edges are assigned to MST edges. Initially, each vertex is placed in its own set. While traversing edges in

Algorithm 1 Linear Time MST Replacement Edges

```
Require: Graph $G$, MST edges labeled, and sorted list of non-MST edges
Require: Zero-initialized arrays $P$, IN, OUT of size $n$ indexed by all vertices $v \in V$.
1: procedure PathLabel($s$, $t$, $e$)
2:   if IN[$s$] < IN[$t$] < OUT[$s$] then                     ▷ $s$ is ancestor of $t$
3:     return
4:   if IN[$t$] < IN[$s$] < OUT[$t$] then                     ▷ $t$ is ancestor of $s$
5:     PLAN ← ANCESTOR, $k_1 \leftarrow$ IN[$t$], $k_2 \leftarrow$ IN[$s$]
6:   else
7:     if IN[$s$] < IN[$t$] then
8:       PLAN ← LEFT, $k_1 \leftarrow$ OUT[$s$], $k_2 \leftarrow$ IN[$t$]          ▷ $s$ is left of $t$
9:     else
10:      PLAN ← RIGHT, $k_1 \leftarrow$ OUT[$t$], $k_2 \leftarrow$ IN[$s$]        ▷ $s$ is right of $t$
11:  $v \leftarrow s$
12:  while $k_1 < k_2$ do                                  ▷ Detecting when below LCA($s$, $t$)
13:    if find($v$) $= v$ then                   ▷ If true, set replacement edge for $(v, P[v])$
14:      $R_{(v,P[v])} \leftarrow e$                                  ▷ Set the replacement edge
15:      link($v$)                               ▷ Union the disjoint sets of $v$ and $P[v]$
16:    $v \leftarrow$ find($v$)
17:    switch PLAN do
18:      case ANCESTOR
19:        $k_2 \leftarrow$ IN[$v$]
20:      case LEFT
21:        $k_1 \leftarrow$ OUT[$v$]
22:      case RIGHT
23:        $k_2 \leftarrow$ IN[$v$]

24: Root the MST $T$ at arbitrary vertex $v_r$ and store parents in $P$.
25: $P[v_r] \leftarrow v_r$                                   ▷ root's parent points to root
26: Run DFS on $T$, setting IN[$v$] and OUT[$v$] to the counter value when $v$ is first and last visited,
    respectively.
27: for all vertices $v \in V$ do
28:   makeset($v$)                                            ▷ Initialize the disjoint sets
29: for all edges $e \in E_T$ do
30:   $R_e = \emptyset$                                     ▷ Initialize the replacement edges
31: for $k \leftarrow 1 \mathinner{..} m-n+1$ do                ▷ Scan the $m-n+1$ sorted non-MST edges
32:   $(v_i, v_j) \leftarrow e_k$
33:   PathLabel($v_i, v_j, (v_i, v_j)$)
34:   PathLabel($v_j, v_i, (v_i, v_j)$)
```

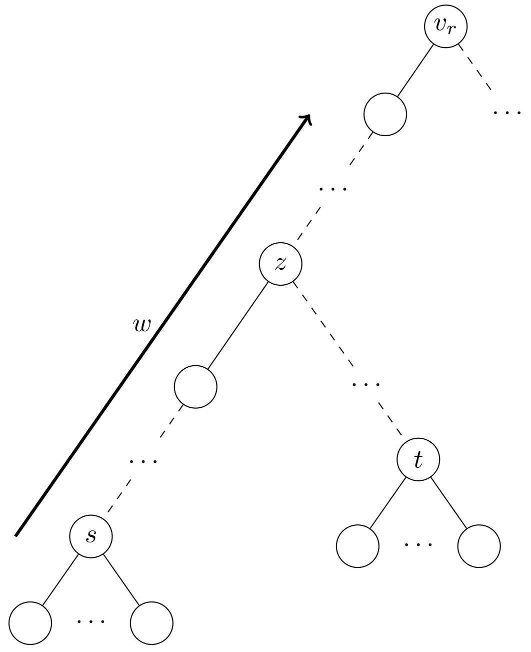

Figure 1: The PathLabel algorithm detects when vertex w on the path from s to the root v_r is an ancestor of the vertex $z = \text{LCA}[s, t]$, without determining z.

a cycle that have not yet been assigned a replacement, the disjoint sets compress the subpath by uniting the sets corresponding to each vertex and its parent in the tree, thereby ensuring that MST edges are traversed at most once.

The conventional union heuristic with path compression for the disjoint set union problem would not lead us to a linear-time algorithm. Gabow and Tarjan [7, 8] designed a linear-time algorithm for the special case where the structure of the unions, called the *Union Tree*, is known in advance. The Gabow-Tarjan approach executes a sequence of m union and find operations on n elements in $\text{O}(m + n)$ time and $\text{O}(n)$ space. The functions are **makeset**(v) that initializes v into a singleton set with label v, **find**(v) that returns the label of the set containing v, and **link**(v) that unites the sets v and $P[v]$, where $P[v]$ is the parent of v in the union tree, and gives it the label of the set containing $P[v]$.

For the path compression used in our MST replacement edge algorithm, the structure of unions is known in advance; that is, the union tree is equivalent to the MST. Hence, we use the Gabow-Tarjan approach for the disjoint sets and path compression.

There are cases when the algorithm may terminate prior to scanning the entire list of edges. This observation leads to a faster implementation that still runs in linear time. A *bridge* edge of a connected graph is defined as an edge whose removal disconnects the graph. Clearly, bridge edges will always be included in the MST and will not have a replacement edge in the solution. Tarjan [21] shows that counting the number of bridges in the graph G takes $\text{O}(m + n)$ time. Thus, Algorithm 1 may terminate the scanning of remaining edges once $n - 1 - k$ replacement edges are identified, where k is the number of bridges in G.

3.1 Example

In this section we give a simple walk-through of the algorithm on the graph in Figure 2. This example exercises all three plans in the algorithm.

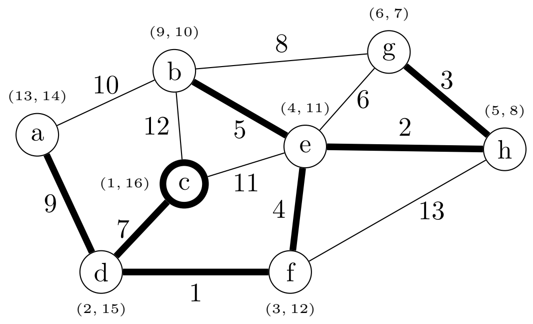

Figure 2: An example graph on 8 vertices $(a, \ldots, h)$ and 13 weighted edges. The MST root vertex c and MST edges are highlighted by thicker lines.

The MST edges have weights 1, 2, 3, 4, 5, 7, and 9, and say the root of the MST tree is vertex c. The vertices in Figure 2 have been labelled with (IN,OUT) numbers assigned by DFS over the MST edges with branching order by ascending edge weight. In the following walk-through of the algorithm, the reader should note that all vertices retain their original parents. Also we remark that all walks are in order from descendent to ancestor or from *last* to *first* in DFS order.

Then in sorted, non-MST edge order we begin with the (g, e) edge at line 33.

1. Vertex e is the ancestor of g, then at line 4 we get the ancestor (ANCESTOR) plan with $k_1 = \text{IN}[e]$, $k_2 = \text{IN}[g]$ and thus $k_1 < k_2$.

2. The cycle traversal begins with g (line 11). Since g has not yet been visited then line 14 assigns the current non-MST edge (g, e) to (g, h), where h is the parent of g.

3. The disjoint sets are linked (line 15) so g's disjoint set gets h's label. This compresses the subpath (g, h).

4. The next vertex is h since it is the parent of g (line 16) and then k_2 is updated to $\text{IN}[h]$.

5. Continuing the traversal with h (line 12), again line 14 assigns (g, e) to (h, e) where e is the parent of h.

6. The disjoint sets are linked (line 15) so h's disjoint set gets e's label. This compresses the subpath (g, h, e).

7. Now the next vertex is e so k_2 gets $\text{IN}[e]$ making it equal to k_1, thus ending the while loop.

8. The oppositely-oriented edge (e, g) input at line 34 is not processed because e is the ancestor of g and we have already followed the path from descendent to ancestor.

The next non-MST edge is (b, g) and input at line 33, since it happens that g was reached before b in the DFS.

9. We get the RIGHT branch plan with $k_1 = \text{OUT}[g]$, $k_2 = \text{IN}[b]$ and so again $k_1 < k_2$.
10. The traversal begins with b (line 11) and since b has not yet been visited then (b, e) gets the non-MST edge (b, g), where e is the parent of b.
11. The disjoint set is linked (line 15) so b's disjoint set gets e's label and the subpath (b, e) is compressed.
12. The next vertex is e (the parent of b) and thus k_2 is updated to $\text{IN}[e]$ (lines 22-23) making $k_2 < k_1$ and thus ending the while loop.
13. The oppositely-oriented edge (g, b) is input at line 34.
14. We get the LEFT branch plan (line 8) with $k_1 = \text{OUT}[g]$, $k_2 = \text{IN}[b]$ so $k_1 < k_2$ and start the traversal with g.
15. Now observe that the disjoint sets had previously compressed the subpath (g, h, e). Thus **find**$(g) \neq g$. This jumps the walk to the LCA, which is vertex e, and updates k_1 to $\text{OUT}[e]$ to end the while loop.

Observe for edge (b, g) that if b were reached before g in the DFS, it would have finished earlier but all subpaths would have been compressed as before. We leave it as an exercise for the reader to finish the algorithm on the remaining non-MST edges.

3.2 Proof of correctness

Claim 1 *The lowest weight non-MST edge that induces a cycle containing an MST edge e is the replacement for e. This follows from the* Cut Property *[4, c.f. Theorem 23.1].*

Claim 2 *Algorithm 1 traverses the cycle induced by a non-MST edge from descendent to ancestor, and stops at the LCA (in the case that the LCA is different from s and t).*

Proof: Observe that the parent is set for each vertex in DFS order so that the traversal carried out by lines 12–23 follows a single path from descendent to ancestor. The path is an upwards traversal of the compressed subpaths in the disjoint sets. For each (s, t) edge, s may be the ancestor of t or vice versa, or the LCA is neither s nor t. The lines 2–10 always set the starting vertex in the traversal of the cycle so that it proceeds from descendent to ancestor as follows.

If s is the ancestor of t then no traversal is made because line 2 returns. If t is the ancestor of s, then the traversal begins with s at line 11 and each traversal up using the disjoint sets leads to t. Otherwise, there is an LCA and from lines 33–34 each branch is traversed from s and t up to the LCA. The subpath compression using disjoint sets occurs at line 15. The linking unites all sets corresponding to vertices in the tree traversal from s and t up to the LCA. □

Claim 3 *Algorithm 1 traverses only those edges in the unique cycle induced by a given non-MST edge.*

Proof: We prove this using a loop invariant for a single cycle. Let (s, t) be a non-MST edge and denote the cycle it induces by $s, v_i, v_{i+1}, \ldots, t, s$.

The loop invariant is: v at the start of the while loop at lines 12–23 must be a vertex in the cycle induced by (s, t).

The base step holds trivially since the starting vertex is s.

The inductive step maintains the loop invariant as follows. At each iteration the disjoint sets of each vertex and its parent are united and by Claim 2 this vertex must be a predecessor in the path from descendent to ancestor. Thus every iteration produces the sequence $v_i, v_{i+1}, \ldots, v_p$ where v_p is either t or an LCA of s and t. By Claim 2, the traversal cannot go above the LCA of s and t.

Termination of the loop is determined by new values for either k_1 or k_2 between lines 17–23. If the case was that t was the ancestor of s, then k_2 decreases in value as the path traversal using disjoint sets approaches t. Otherwise the LCA is neither s nor t and if s is visited before t in DFS order, then it is in the *left* branch and k_1 increases in value as the upwards path traversal using disjoint sets approaches t, otherwise we have the *right* branch and similarly the loop ends as the path traversal using disjoint sets moves towards the other endpoint. □

Theorem 1 *Given the Minimum Spanning Tree for an undirected, weighted graph $G = (V, E)$, and non-tree edges sorted by weight, then Algorithm 1 correctly finds all minimum cost replacement edges in the Minimum Spanning Tree of G.*

Proof: First observe that all non-MST edges are processed in ascending order by weight between lines 31–34. Then the (s, t) edge that induces the first cycle to contain an MST edge must be the replacement edge for that MST edge following Claim 1 and the order of processing. This is carried out by line 14, hence each MST edge gets the first non-MST edge that induces a cycle containing it.

It follows from Claim 3 and the loop over all non-MST edges at lines 31–34 that all MST edges in a cycle will get a replacement edge.

At the end of a cycle, the traversed edges in the subpath are compressed with each parent set by linking the disjoint sets so that any edge from this cycle cannot be traversed again because it has been assigned a replacement edge. □

3.3 Complexity analysis

Claim 4 *Algorithm 1 updates disjoint sets in $O(m + n)$ time and $O(n)$ space.*

Proof: The Gabow-Tarjan disjoint sets use $\mathrm{O}(n)$ **makeset** operations (lines 27-28), one for each vertex $v \in V$; and $\mathrm{O}(n)$ **link** operations (line 15) since there are at most $n - 1$ replacement edges. For each non-tree edge, there are at most two **find** operations at the start and end of each of the two PathLabel calls, corresponding with the initial **find**(s) (line 13) and the final **find**(v) (line 16) that returns a label of either t or an ancestor of t. Hence these contribute to at most $4(m - n + 1) = \mathrm{O}(m)$ **find** operations. Every other **find** precedes a **link** operation, so there are $\mathrm{O}(n)$ of these **find** operations. Therefore, Algorithm 1 uses $\mathrm{O}(m)$ **find** operations.

The union tree is equivalent to the MST tree. Hence, Algorithm 1 uses the special case of disjoint set union when the union tree is known in advance. Using the Gabow-Tarjan disjoint set union, thus, takes $\mathrm{O}(m + n)$ time and $\mathrm{O}(n)$ space. □

Theorem 2 *Given the Minimum Spanning Tree for an undirected, weighted graph $G = (V, E)$, and non-tree edges sorted by weight, then Algorithm 1 finds all minimum cost replacement edges of the Minimum Spanning Tree of G in $O(m + n)$ time and $O(m + n)$ space.*

Proof: Let T be the Minimum Spanning Tree of G. Initializing all values in the parent array P takes $\mathrm{O}(n)$ time. Since there are $n - 1$ edges in T then running DFS on T (line 26) to initialize the IN and OUT arrays takes $\mathrm{O}(n)$ time. Initializing the replacement edges of the MST edges takes $\mathrm{O}(n)$ time.

There are $m - n + 1 = \mathrm{O}(m)$ non-MST edges read in ascending order by weight, taking $\mathrm{O}(m)$ time. For each non-MST edge, it was established by Claim 3 that the algorithm can only reference edges in the fundamental cycle induced by that non-MST edge. These edges are traversed only once as follows.

The algorithm walks each fundamental cycle in the same direction from descendant to ancestor, as imposed by the DFS ordering set in the IN and OUT arrays. On visiting a vertex v, if v's set label equals v then the edge $(v, P[v])$ has not been visited before, otherwise it violates the path compression at lines 13-15. When an edge $(v, P[v])$ gets a replacement (line 14), the disjoint set corresponding with parent $P[v]$ is united with v's set (line 15) using the Gabow-Tarjan disjoint set **link** operation. The label of the new set is the root of the induced subtree of the MST. Therefore when a vertex v is first visited, **link**(v) results in the set label being the label of the set containing $P[v]$. On completing the walk along the cycle, the set label will be the label of the set containing the LCA. Then subsequent **find**(v) operations return the most recent root of the subtree containing v. Thus this sequence of disjoint set unions perform path compression on tree edges with assigned replacements. The compressed path decreases the traversal length of subsequent walks beginning at vertices lower in the DFS ordering by skipping over tree edges already with assigned replacement edges.

It follows from Claim 3 and this specific ordering of the disjoint set labels that the algorithm cannot follow a path that does not close the cycle. Then because of path compression only $\mathrm{O}(m)$ edges are traversed, taking $\mathrm{O}(m)$ time. Claim 4 establishes $\mathrm{O}(m + n)$ time and $\mathrm{O}(n)$ space for all disjoint set operations. Hence it takes $\mathrm{O}(m + n)$ time to find all replacement edges in T. The data structures are simple arrays and Gabow-Tarjan disjoint set union data structures, taking $\mathrm{O}(n)$ space, and all non-MST edges take $\mathrm{O}(m)$ space. Therefore it takes $\mathrm{O}(m + n)$ time and $\mathrm{O}(m + n)$ space as claimed. ☐

4 Most Vital Edge

The most vital edge of a connected, weighted graph G is the edge whose removal causes the largest increase in the weight of the minimum spanning tree [12]. When the graph contains bridges (which can be found in linear time [21]), the most vital edge is undefined. The input for methods to find the most vital edge include both the graph and the edges sorted by weight. Hsu et al. [12] designed algorithms to find the most vital edge in $\mathrm{O}(m \log m)$ and $\mathrm{O}(n^2)$ time. Iwano and Katoh [13] improve this with $\mathrm{O}(m + n \log n)$ and $\mathrm{O}(m\alpha(m, n))$ time algorithms. Suraweera et al. [20] prove that the most vital edge is in the minimum spanning tree. Hence, once Algorithm 1 finds all replacement edges of the minimum spanning tree, the most vital edge takes $\mathrm{O}(n)$ time by simply finding the tree edge with maximum difference in weight from its replacement edge. Thus, our approach will also find the most vital edge in $\mathrm{O}(m + n)$ time, and is the first linear algorithm for finding the most vital edge of the minimum spanning tree given the non-tree edges sorted by weight.

References

[1] H. Booth and J. Westbrook. A linear algorithm for analysis of minimum spanning and shortest-path trees of planar graphs. *Algorithmica*, 11(4):341–352, 1994. doi:10.1007/BF01187017.

[2] G. Cattaneo, P. Faruolo, U. Ferraro Petrillo, and G. Italiano. Maintaining dynamic minimum spanning trees: An experimental study. *Discrete Applied Mathematics*, 158(5):404–425, 2010. doi:10.1016/j.dam.2009.10.005.

[3] F. Chin and D. Houck. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences*, 16(3):333–344, 1978. doi:10.1016/0022-0000(78)90022-3.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, Inc., Cambridge, MA, 2009.

[5] G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985. doi:10.1137/0214055.

[6] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *Journal of Computer and System Sciences*, 48:533–551, 1994. doi:10.1016/S0022-0000(05)80064-9.

[7] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 246–251, New York, NY, USA, 1983. ACM. doi:10.1145/800061.808753.

[8] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

[9] K. Hanauer, M. Henzinger, and C. Schulz. Recent advances in fully dynamic graph algorithms – a quick reference guide. *ACM Journal of Experimental Algorithmics*, 2022. doi:10.1145/3555806.

[10] M. Henzinger and V. King. Maintaining minimum spanning trees in dynamic graphs. In *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 594–604, 1997.

[11] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, July 2001. doi:10.1145/502090.502095.

[12] L.-H. Hsu, R.-H. Jan, Y.-C. Lee, C.-N. Hung, and M.-S. Chern. Finding the most vital edge with respect to minimum spanning tree in weighted graphs. *Information Processing Letters*, 39(5):277–281, 1991. doi:10.1016/0020-0190(91)90028-G.

[13] K. Iwano and N. Katoh. Efficient algorithms for finding the most vital edge of a minimum spanning tree. *Information Processing Letters*, 48(5):211–213, 1993.

[14] A. Kooshesh and R. Crawford. Yet another efficient algorithm for replacing the edges of a minimum spanning tree. In *Proceedings of the 1996 ACM 24th Annual Conference on Computer Science*, CSC '96, pages 76–78, New York, NY, USA, 1996. ACM. doi:10.1145/228329.228337.

[15] J. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

[16] S. Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-ackermann time. In *16th International Symposium on Algorithms and Computation (ISAAC)*, volume 3827 of *Lecture Notes in Computer Science*, pages 964–973, Sanya, Hainan, China, 2005. Springer.

[17] S. Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-ackermann time. *Journal of Graph Algorithms and Applications*, 19(1):375–391, 2015. doi:10.7155/jgaa.00365.

[18] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1):16–34, 2002. doi:10.1145/505241.505243.

[19] P. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM Journal on Computing*, 4(3):375–380, 1975. doi:10.1137/0204032.

[20] F. Suraweera, P. Maheshwari, and P. Battacharya. Optimal algorithms to find the most vital edge of a minimum spanning tree. Technical Report CIT-95-21, School of Comput. and Inf. Tech., Griffith University, 1995.

[21] R. E. Tarjan. A note on finding the bridges of a graph. *Information Processing Letters*, 2(6):160–161, 1974.

[22] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, Oct. 1979. doi:10.1145/322154.322161.

[23] R. E. Tarjan. Sensitivity analysis of minimum spanning trees and shortest path trees. *Information Processing Letters*, 14(1):30–33, 1982. doi:10.1016/0020-0190(82)90137-5.