# A Linear Time Algorithm for Finding Minimum Spanning Tree Replacement Edges

David A. Bader
*Department of Computer Science*
*New Jersey Institute of Technology*
*Newark, NJ 07102*
*Email: bader@njit.edu*

Paul Burkhardt
*Research Directorate*
*National Security Agency*
*Fort Meade, MD 20755*
*Email: pburkha@nsa.gov*



**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 O$(m\alpha(m,n))$ time, where $\alpha(m,n)$ is the inverse Ackermann's function. Given the MST and sorted non-tree edges, our algorithm is the first that runs in O$(m+n)$ time and O$(m+n)$ space to find all replacement edges. Moreover, it is easy to implement and our experimental study demonstrates fast performance on several types of graphs. 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 $T=\text{MST}(G)$ is a subset of $n-1$ edges with the minimal sum of weights that connects the $n$ vertices.

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 $\text{O}(m \log n)$ time from the non-tree edges, or $\text{O}(m + n \log n)$ 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 $\text{O}(m)$ unique cycles and each cycle can have $\text{O}(n)$ 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 since the 1970s (e.g., [15], [4], [18], [13]). The best algorithm is from 1979 due to Tarjan [18] and runs in $\text{O}(m\alpha(m,n))$, where $\alpha(n,m)$ is the inverse Ackermann's function. Given edges sorted by weight, our algorithm is asymptotically faster and is surprisingly simple, taking $\text{O}(m+n)$ time and $\text{O}(m+n)$ space.

The main result of this paper is a simple and fast algorithm for the MST replacement edge problem. Given the minimum spanning tree and non-tree edges sorted by weight, our algorithm is the first to find all replacement edges in $\text{O}(m+n)$ time and $\text{O}(m+n)$ space. 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 uses simple arrays and alleviates the need to use least common ancestor (LCA) algorithms. We demonstrate its performance on power-law graphs.

## 2. Linear-Time Algorithm

Given $T$ and the remaining non-tree edges $E-T$ sorted from lowest to highest weight, then Algorithm 1 gives the minimum spanning tree replacement edges algorithm. Observe that each of the $m-n+1$ edges in $E-T$ induces a fundamental cycle with the edges in $T$. Then for any MST edge there is a subset of cycles on that edge, and the cycle induced by the lightest non-MST edge is the replacement for it. This follows from the Cut Property [5] where the 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 [14] MST algorithm.

**Algorithm 1** Linear Time MST Replacement Edges

**Input:** Graph $G$, MST edges labeled, and sorted list of non-MST edges

**procedure** PATHLABEL($s$, $t$, $e$)
  **if** IN[$s$] $<$ IN[$t$] $<$ OUT[$s$] **then** ▷ $s$ is ancestor of $t$
    **return**
  **if** IN[$t$] $<$ IN[$s$] $<$ OUT[$t$] **then** ▷ $t$ is ancestor of $s$
    PLAN $\leftarrow$ ANC, $k_1 \leftarrow$ IN[$t$], $k_2 \leftarrow$ IN[$s$]
  **else**
    **if** IN[$s$] $<$ IN[$t$] **then**
      PLAN $\leftarrow$ LEFT, $k_1 \leftarrow$ OUT[$s$], $k_2 \leftarrow$ IN[$t$] ▷ $s$ is left of $t$
    **else**
      PLAN $\leftarrow$ RIGHT, $k_1 \leftarrow$ OUT[$t$], $k_2 \leftarrow$ IN[$s$] ▷ $s$ is right of $t$
  $i \leftarrow 0$
  $\hat{v} \leftarrow s$
  **while** $k_1 < k_2$ **do** ▷ Detecting when below LCA($s$, $t$)
    **if** RippleTree[$\hat{v}$] $= P[\hat{v}]$ **then** ▷ If true, inspect the MST edge from $\hat{v}$ to parent
      $\hat{e} \leftarrow \langle \hat{v}$, P[$\hat{v}$]$\rangle$ ▷ $\hat{e} \in$ MST
      **if** $R_{\hat{e}} = \emptyset$ **then** ▷ Replacement edge has not been found yet
        $R_{\hat{e}} \leftarrow e$ ▷ Set the replacement edge
    $L[i] \leftarrow \hat{v}$ ▷ Add $\hat{v}$ to *Ripple* tree subpath compression list
    $i \leftarrow i + 1$
    $\hat{v} \leftarrow$ RippleTree[$\hat{v}$]
    **switch** PLAN **do**
      **case** ANC
        $k_2 \leftarrow$ IN[$\hat{v}$]
      **case** LEFT
        $k_1 \leftarrow$ OUT[$\hat{v}$]
      **case** RIGHT
        $k_2 \leftarrow$ IN[$\hat{v}$]
  **for** $j \leftarrow 0, i - 1$ **do** ▷ Update the *Ripple* tree
    RippleTree[$L[j]$] $\leftarrow \hat{v}$

**procedure** FINDREPLACEMENTEDGES
  Root the MST $T$ at arbitrary vertex $v_r$ and store parents in $P[.]$
  $P[v_r] \leftarrow v_r$ ▷ root's parent points to root
  Run DFS on $T$, setting IN[$v$] and OUT[$v$] to the sequence when $v$ is first and last visited, respectively.
  **for all** vertices $v \in V$ **do**
    RippleTree[$v$] $\leftarrow P[v]$ ▷ Initialize the *Ripple* tree to the MST
  **for** $k \leftarrow 1$, $m - n + 1$ **do** ▷ Scan the $m - n + 1$ sorted non-MST edges
    $\langle v_i, v_j \rangle \leftarrow e_k$
    PATHLABEL($v_i, v_j, \langle v_i, v_j \rangle$)
    PATHLABEL($v_j, v_i, \langle v_i, v_j \rangle$)

The major steps of our approach are 1) rooting the MST, 2) using the rooted MST to compute several key vertex-based values, 3) scanning the non-tree edges, and 4) for each non-tree edge, inspecting the tree edges in the fundamental cycle. With O$(m)$ non-tree edges and O$(n)$ edges in each cycle, the naïve approach has $\Omega(mn)$ time complexity. This paper introduces an algorithm that reduces the cost to O$(m + n)$ time by using a novel tree data structure we call a *Ripple* tree.*

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 (DFS) 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 least common ancestor vertex in the rooted MST for each non-tree edge. To do so, we used a pair of

*. The name *Ripple* is motived by the music lyric "And if you go no one may follow / That path is for your steps alone" [10].

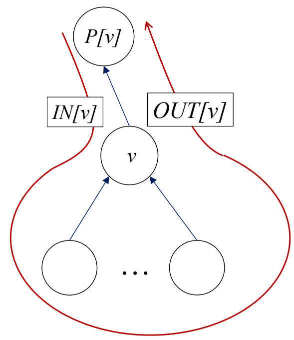


Figure 1. Depth-first traversal of the minimum spanning tree $T$, setting each parent $P[v]$, and IN$[v]$, and OUT$[v]$ to the counter when vertex $v$ is first and last visited, respectively.

vertex-based values, IN$[v]$ and 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, IN$[v]$ is assigned the current counter value. When the traversal backtracks up an edge from vertex $v$, OUT$[v]$ is then assigned the current counter value. Figure 1 shows the depth-first traversal of the MST.

The $m - n + 1$ remaining edges in $E - T$ 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 is included in a fundamental cycle, its replacement is set to the non-tree edge from that cycle. As we will describe, the *Ripple* tree provides subpath compression as replacement edges are assigned to MST edges. In Algorithm 1, RippleTree$[\cdot]$ stores the *Ripple* tree.

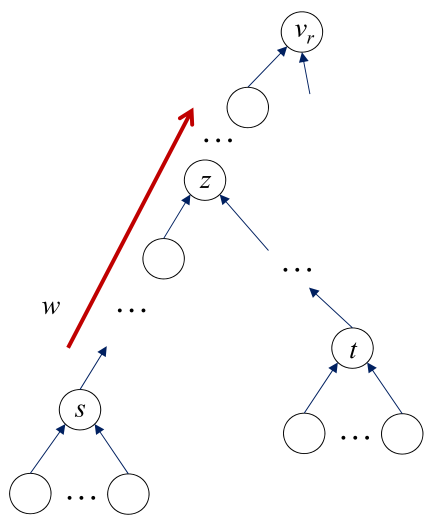


Figure 2. The PathLabel algorithm detects when vertex $w$ on the path from $s$ to the root $v_r$ is an ancestor of the vertex $z =$ LCA$[s, t]$, without determining $z$.

For each non-tree edge $\langle s, t \rangle$, if vertex $t$ is an descendant of $s$, (if and only if IN$[s] <$ IN$[t] <$ OUT$[t] <$ OUT$[s]$), we make a single PathLabel call for the edges from $t$ up to $s$. 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 least common ancestor (LCA) of $s$ and $t$ in the tree. We assume, without loss of generality, that $s$ is visited in the depth-first seach traversal before $t$. Let's call $z =$ 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 IN$[z] <$ IN$[s] <$ OUT$[s] <$ IN$[t] <$ OUT$[t] <$ OUT$[z]$ by definition of the depth-first traversal. As illustrated in Figure 2, 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 OUT$[w] <$ IN$[t]$), or from $z$ to the root $v_r$ (where OUT$[w] >$ IN$[t]$). We use this fact to detect when PathLabel reaches the LCA without computing it.

As mentioned earlier, the *Ripple* tree provides subpath compression as replacement edges are assigned to MST edges.

The *Ripple* tree is initialized to the MST. While traversing edges in a cycle that have not yet been assigned a replacement, the *Ripple* tree compresses the subpath for these tree edges so they cannot be followed again. Unlike the path compression in Tarjan's Union-Find [18], the *Ripple* tree compresses subpaths in conjunction with the original (MST) tree. Another significant difference is that Tarjan's Find operation always compresses the entire path from a vertex to the root, while the *Ripple* tree efficiently compresses tree subpaths.

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 [17] shows that counting the number of bridges in the graph $G$ takes $O(m+n)$ 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 $G$.

## 2.1. Illustrative Example

In this section we give a simple walkthrough of the algorithm on the graph in Figure 3. This illustrative example exercises all three plans in the algorithm.

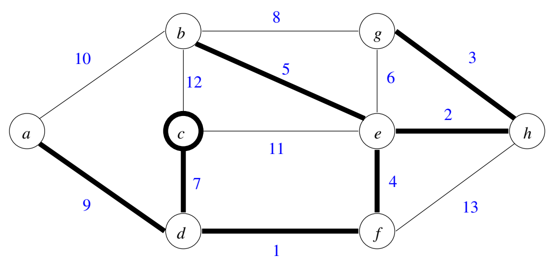


Figure 3. An example graph on 8 vertices $(a, \ldots, h)$ and 13 weighted edges (label/weight in blue). The MST root vertex $c$ and MST edges are highlighted by thicker lines.

The MST edges are $e_1, e_2, e_3, e_4, e_5, e_7, e_9$ and say the root of the MST tree is vertex $c$. In the following walkthrough 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 $\langle g, e\rangle$ edge at line 38.

1) Vertex $e$ is the ancestor of $g$, then at line 4 we get the ancestor (ANC) 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 12). Since $g$ has not yet been visited then lines 15-17 assign the current non-MST edge $\langle g, e\rangle$ to $\langle g, h\rangle$, where $h$ is the parent of $g$.
3) The next vertex is $h$ since it is the parent of $g$ (line 20) and then $k_2$ is updated to $\text{IN}[h]$.
4) Continuing the traversal with $h$ (line 12), again lines 15-17 assign $\langle g, e\rangle$ to $\langle h, e\rangle$ where $e$ is the parent of $h$.
5) Now the next vertex is $e$ so $k_2$ gets $\text{IN}[e]$ making it equal to $k_1$, thus ending the while loop.
6) The *Ripple* tree is updated (lines 28-29) so both $\text{RippleTree}[g]$ and $\text{RippleTree}[h]$ get $e$. This compresses the subpath $(g, h, e)$.
7) The counter-oriented edge $\langle e, g\rangle$ input at line 39 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 $\langle b, g\rangle$ and input at line 38, but suppose $g$ was reached before $b$ in the DFS.

8) We get the RIGHT branch plan with $k_1 = \text{OUT}[g]$, $k_2 = \text{IN}[b]$ and so again $k_1 < k_2$.
9) The traversal begins with $b$ (line 12) and since $b$ has not yet been visited then $\langle b, e\rangle$ gets the non-MST edge $\langle b, g\rangle$, where $e$ is the parent of $b$.
10) The next vertex is $e$ (the parent of $b$) and thus $k_2$ is updated to $\text{IN}[e]$ (lines 26-27) making $k_2 < k_1$ and thus ending the while loop.
11) The *Ripple* tree is updated (lines 28-29) so $\text{RippleTree}[b]$ gets $e$ and the subpath $(b, e)$ is compressed.
12) The counter-oriented edge $\langle g, b\rangle$ is input at line 39.
13) 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 $g$.
14) Now observe that the *Ripple* tree had previously compressed the subpath $(g, h, e)$. Thus $\text{RippleTree}[g] \neq P[g]$. This jumps the walk to the LCA, which is vertex $e$, and updates $k_2$ to $\text{OUT}[e]$ to end the while loop.

Observe for edge $\langle b, g\rangle$ 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.

## 2.2. 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 [5].*

**Claim 2** *Algorithm 1 traverses the cycle induced by a non-MST edge from descendent to ancestor.*

**Proof of claim 2** First observe that the parent is set for each vertex in DFS order so that the traversal carried out by lines 13–27 follows a single path from descendent to ancestor. The path is an upwards traversal of the compressed subpaths in the *Ripple* tree. For each $\langle s, t \rangle$ edge, $s$ may be the ancestor of $t$ or vice versa, or there is a least common ancestor (LCA) between $s$ and $t$. 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 12 and each traversal up a *Ripple* tree edge leads to $t$. Otherwise, there is an LCA and from lines 38–39 each branch is traversed from $s$ and $t$ up to the LCA.

The subpath compression of the *Ripple* tree occurs at lines 28–29 where all of the *Ripple* tree edges traversed from the descendent to ancestor are replaced by edges that point from each vertex to the top-most vertex reached by this *Ripple* tree traversal. □

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

**Proof of claim 3** We prove this by loop invariance for a single cycle. Let $\langle s, t \rangle$ 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 such that each vertex at the start of the while loop at lines 13–27 must be a vertex in the cycle induced by $\langle s, t \rangle$.

The base step at line 12 holds trivially since the starting vertex is $s$.

The inductive step maintains the loop invariant as follows. At each iteration each vertex's parent in the *Ripple* tree is set and by Claim 2 this must be a predecessor in the path from descendent to ancestor, thus every iteration produces the sequence $v_i, v_{i+1}, \ldots, t$ where $t$ is an ancestor or LCA.

Termination of the loop is determined by new values for either $k_1$ or $k_2$ between lines 21–27. If the case was that $t$ was the ancestor of $s$, then $k_2$ decreases in value as the *Ripple* tree path traversal approaches $t$. Otherwise there is an LCA and if $s$ is older, meaning it precedes $t$ in the ancestry tree, then it is in the *left* branch and $k_1$ increases in value as the upwards *Ripple* tree path traversal approaches $t$, otherwise we have the *right* branch and similarly the loop ends as the *Ripple* tree path traversal 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 36–39. Then the $\langle s, t \rangle$ 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 16, 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 36–39 that all MST edges in a cycle will get a replacement edge.*

*At the end of a cycle, the traversed* Ripple *tree edges in the subpath are compressed with parent set to the ancestor or LCA so that any edge from this cycle cannot be traversed again.* □

## 2.3. Complexity analysis

**Claim 4** *Algorithm 1 references at most $2m = O(m)$ edges.*

**Proof of claim 4** The algorithm inspects $\mathrm{O}(m)$ non-tree edges using the dynamically updated *Ripple* tree. To count the number of *Ripple* tree edges traversed, we partition the calls into two groups: when *Ripple* tree edges coincide with MST edges, and when these edges are in a compressed path. For the first group, since there are $n-1$ tree edges, a *Ripple* tree edge is traversed twice (in and out of the tree edge) before the path compression in PATHLABEL in lines 28–29 of Algorithm 1. Hence, at most $2(n-1)$ *Ripple* tree edges are referenced in this group. In the second group, we have a *Ripple* tree edge in a compressed subpath in PATHLABEL for each of the two endpoints of the $E - T$ remaining edges. While a traversal up the ancestry of the *Ripple* tree in lines 13–27 may traverse other compressed subpaths as the traversal potentially alternates with the first group of initial *Ripple* tree edges, we have already counted these with the first group. Hence, there are at most $2(m-n+1)$ *Ripple* tree edges referenced in this group. The maximum number of *Ripple* tree edges referenced is the sum $2(n-1) + 2(m-n+1) = 2m = \mathrm{O}(m)$. □

**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 $P$ and* Ripple *tree takes $O(n)$ time. Since there are $n - 1$ edges in $T$ then running Depth-First Search (DFS) on $T$ (line 33) to initialize the IN and OUT arrays takes $O(n)$ time. It remains to show that finding all $O(n)$ replacement edges takes $O(m)$ time.*

*There are $m - n + 1 = O(m)$ non-MST edges read in ascending order by weight, taking $O(m)$ 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. At the end of a walk, the* Ripple *edges for the vertices in the cycle are compressed and updated at lines 28–29 to point towards the ancestor (or LCA). Thus on subsequent walks from edges lower in the DFS ordering, the algorithm jumps up to the ancestor or root.*

*It follows from Claim 3 and this specific ordering of* Ripple *tree edges that the algorithm cannot follow a path that does not close the cycle. Claim 4 establishes that it takes $O(m)$ time to reference all edges. Then each endpoint $v$ of a compressed path in the* Ripple *tree is accessed at most $d(v)$ times leading to $O(m)$ references overall. In total it takes $O(m)$ time to find all replacement edges in $T$. All data structures are simple arrays, taking $O(n)$ space, and all non-MST edges take $O(m)$ space. Therefore it takes $O(m + n)$ time and $O(m + n)$ space as claimed.* □

### 2.4. Related Work

Spira and Pan [15] presented an $O(n^2)$ algorithm to update the MST when new vertices are added, and could find all replacement edges in $O(n^3)$ time. Chin and Houck [4] improved this bound to $O(n^2)$ using a more efficient approach to insert and delete vertices from the graph. Tarjan [18] gave an $O(m\alpha(m, n))$ time algorithm using path compression, where $\alpha(n, m)$ is the inverse Ackermann's function. Kooshesh and Crawford [13] proposed an algorithm that computes the replacement for every edge in the minimum spanning tree that runs in $O(\max(C_{\text{mst}}, n \log n))$, where $C_{\text{mst}}$ is the cost of computing a minimum spanning tree of $G' = (V, E - T)$. Their approach is based on computing efficiently the possible replacement edges from the remaining edge set.

In 1994, Katajainen and Träff [12] designed a parallel algorithm that runs in $O(\log n)$ time and $O(m)$ space using $m$ processors on a MINIMUM CRCW PRAM machine. Their approach uses path product and path labelling techniques. However, this approach does not improve the sequential running time.

Das and Loui [6] solved a similar problem of *node replacement* for deleted vertices in the MST, that runs in $O(m \log n)$ sequential time, or $O(m\alpha(m, n))$ when the edges $E$ are pre-sorted by weight; and a parallel algorithm that takes $O(\log^2 n)$ time and $m$ processors on a CREW PRAM.

## 3. Experimental Study

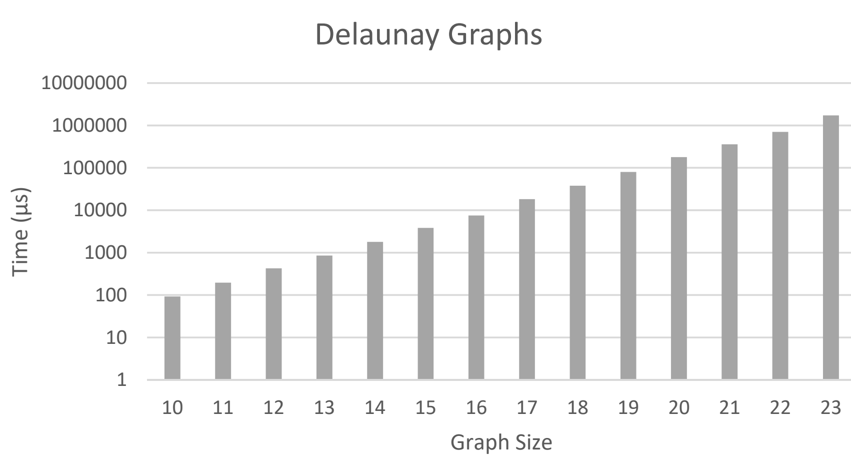


Figure 4. Execution time to find the minimum spanning tree replacement edges in the Delaunay graphs.

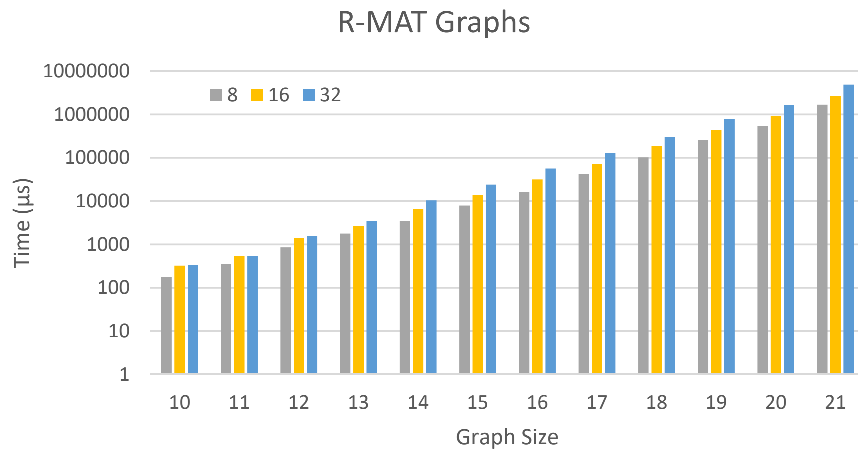


Figure 5. Execution time to find the minimum spanning tree replacement edges in the R-MAT graphs.

| **Graph** | $n$ | $m$ |
|---|---|---|
| delaunay_n10 | 1024 | 3056 |
| delaunay_n11 | 2048 | 6127 |
| delaunay_n12 | 4096 | 12264 |
| delaunay_n13 | 8192 | 24547 |
| delaunay_n14 | 16384 | 49122 |
| delaunay_n15 | 32768 | 98274 |
| delaunay_n16 | 65536 | 196575 |
| delaunay_n17 | 131072 | 393176 |
| delaunay_n18 | 262144 | 786396 |
| delaunay_n19 | 524289 | 1572954 |
| delaunay_n20 | 1048577 | 3145817 |
| delaunay_n21 | 2097153 | 6291539 |
| delaunay_n22 | 4194305 | 12583000 |
| delaunay_n23 | 8388609 | 25165915 |

TABLE 1. NUMBER OF VERTICES AND EDGES IN THE DELAUNAY GRAPHS FROM THE 10TH DIMACS IMPLEMENTATION CHALLENGE.

The minimum spanning tree replacement edge algorithm is both easy to implement and has fast sequential running time. In this section, we perform an empirical study using graphs from the 10th DIMACS Implementation Challenge on Graph Partitioning and Graph Clustering [1].

The first category of graphs uses Delaunay triangularization of random points in the plane [7]. Table 1 gives the size of each graph, and the edge weights are uniformly random.

The second category of graphs is a synthetic real-world network using the Kronecker generator R-MAT [3] and the same parameters $A = 0.57, B = 0.19, C = 0.19, D = 0.05$ selected in the DIMACS Challenge and the Graph500 Benchmark. The R-MAT graph generator produces random graphs, such as social networks, that are designed to fit power-law graphs in which the degree distribution follows an inverse power-law. We modify the generator so that the graph is connected, and assign uniformly random edge weights. For these R-MAT graphs on $n$ vertices, we varied the number of edges using an edge factor of 8, 16, and 32, where the number $m$ of edges is $n$ times this edge factor. For instance, an edge factor of 8 implies that $m = 8n$.

We test our implementation on an Intel Xeon CPU E5-2680 with clock speed of 2.70GHz, using GNU C compiler (`gcc`) version 9.1.0 and optimization level 3 ("`-O3`"). The performance results are given in Figures 4 and 5, for the Delaunay triangularization and R-MAT graphs, respectively. Each plot is on a log-log scale. The horizontal axis labelled "Graph Size" is the logarithm in base 2 of the number of vertices and the vertical axis is the running time in microseconds ($\mu$s). Clearly, the algorithm runs in fast linear time with respect to the graph size.

## 4. Most Vital Edge Algorithm

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 [9]. When the graph contains bridges (which can be found in linear time [17]), the most vital edge is undefined. Hsu et al. [8] designed algorithms to find the most vital edge in $\mathrm{O}(m \log m)$ and $\mathrm{O}(n^2)$ time.

Iwano and Katoh [11] improve this with $\mathrm{O}(m + n \log n)$ and $\mathrm{O}(m\alpha(m, n))$ time algorithms. Suraweera et al. [16] 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.

## References


[1] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, editors. *Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop. February 13-14, 2012.* Contemporary Mathematics 588. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013.

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

[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining (SDM)*, Orlando, FL, April 2004. SIAM.

[4] F. Chin and D. Houck. Algorithms for updating minimal spanning trees. *Journal of Computer and System Sciences*, 16(3):333 – 344, 1978.

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

[6] B. Das and M.C. Loui. Reconstructing a minimum spanning tree after deletion of any node. *Algorithmica*, 31(4):530–547, 2001.

[7] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, Atlanta, GA, April 2010.

[8] L.-H. Hsu, R.-H. Jan, Y.-C. Lee, and C.-N. Hung. Finding the most vital edge with respect to minimum spanning tree in weighted graphs. *Information Processing Letters*, 39(5):277–281, 1991.

[9] 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.

[10] R. Hunter and J. Garcia. Ripple. Ice Nine Publishing Co., Inc., 1970.

[11] 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.

[12] J. Katajainen and J.L. Träff. Simple parallel algorithms for the replacement edge problem and related problems on minimum spanning trees. Technical Report DIKU-94/18, Department of Computer Science, University of Copenhagen, 1994.

[13] A.A. Kooshesh and R.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.

[14] J.B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

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

[16] 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.

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

[18] R.E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, October 1979.