AN EFFICIENT PARALLEL ALGORITHM FOR UPDATING MINIMUM SPANNING TREES *

Peter VARMAN and Kshitij DOSHI
Department of Electrical and Computer Engineering, George R. Brown School of Engineering, Rice University, Houston, TX 77251, U.S.A.

Abstract. A new parallel algorithm for updating the minimum spanning tree of an n-vertex graph following the addition of a new vertex is presented. The algorithm runs in O(log n) time using O(n) processors on a concurrent-read-exclusive-write parallel random access machine. The algorithm is superior to previous algorithms on this model, that either obtain O(log n) time performance using O(n*) processors, or employ O(n) processors but have a time complexity of O(log^2 n).

1. Introduction

An incremental algorithm is a procedure for recomputing the solution to a problem in response to a minor change in the inputs. Incremental algorithms are of interest in real-time or interactive computing environments, where typically, a previously computed solution needs to be updated to reflect the effects of a modification to the inputs. Incremental or update algorithms deal with the dynamic behavior of the system being modeled, in contrast to start-over algorithms that deal with the static behavior of the system.

In this paper we consider the problem of recomputing the minimum spanning tree (MST) of an n-vertex weighted undirected graph G that has been altered by the addition of a single new vertex, along with up to n new edges between it and the vertices of G. This problem is referred to as the vertex insertion problem for MSTs. We present an O(lg n) time algorithm for vertex insertion using O(n) processors on a concurrent-read-exclusive-write (CREW) parallel random access machine (PRAM).

A PRAM consists of a number of synchronous processors all of which have access to a common memory. A CREW PRAM is one of a family of such PRAM models [5]. Members of the family differ in the amount of parallelism allowed when several processors simultaneously access a single memory location. A CREW PRAM allows simultaneous access by any number of processors to a common memory location for reading but forbids simultaneous writes into a single memory location by two or more processors.

* This research was supported in part by an IBM Faculty Development Award.
1 We denote log, n by lg n.
We briefly examine the vertex insertion problem to place our results in context with extant work. It is easy to see that the MST on the \((n + 1)\) vertex graph obtained by introducing a single new vertex into a graph \(G\) would involve only the newly introduced edges and the edges of the previously computed MST on \(G\). Therefore, although \(G\) itself may have \(O(n^2)\) edges, an update algorithm that recomputes the MST when the new vertex is introduced need consider only a sparse graph with at most \((2n - 1)\) edges.

Spira and Pan [11] and Chin and Houck [2] have presented two different sequential algorithms for the vertex insertion problem. Both these algorithms require \(O(n)\) time.

The algorithm by Spira and Pan is based on the “vertex collapsing” technique attributed to Sollin [1]. A crucial observation made by Spira and Pan is that when Sollin’s algorithm is applied to the vertex insertion problem, both the number of vertices and the number of edges are at least halved in every iteration. For a graph with \(n\) vertices, the time complexity of the algorithm is therefore governed by the recurrence \(T(n) = T\left(\frac{n}{2}\right) + O(n)\), so that \(T(n) = O(n)\).

A parallel algorithm for the vertex insertion problem on a CREW PRAM, was proposed by Pawagi and Ramakrishnan [9]. This algorithm uses the observation that there are a maximum of \(O(n^2)\) cycles in the graph consisting of the old MST and the newly added edges, and that all these cycles can be broken in parallel by independently removing the maximum weighted edge on each cycle. The algorithm requires \(O(\log n)\) time and uses \(O(n^2)\) processors. The best time complexity achieved by any parallel start-over algorithm for computing the MST of an undirected graph on a CREW PRAM is \(O(\log^2 n)\) [3, 4, 6, 7, 10, 13].

The algorithm presented in this paper uses only \(O(n)\) processors and achieves a time complexity of \(O(\log n)\) for the vertex insertion problem. In contrast to the parallel algorithm in [9], the algorithm examines only a total of \(O(n \log \log n)\) cycles in computing the MST.

The rest of the paper is organized as follows. Section 2 presents some terminology from graph theory and summarizes previous results which we use in our algorithm. In Section 3, we present an informal description of the vertex insertion algorithm. In Sections 4, 5, and 6, we describe in detail the three phases into which the algorithm is organized. We end the paper with conclusions in Section 7.

2. Preliminaries

Let \(G = (V, E)\) be a graph with vertex set \(V\) and edge set \(E\). Edges of \(G\) may be directed or undirected; accordingly, \(G\) is called a directed or an undirected graph. We denote an undirected edge between vertices \(u\) and \(v\) by \((u, v)\). A directed edge (an arc) from vertex \(u\) to vertex \(v\) is denoted by \((u, v)\). In this paper the unqualified term graph refers to an undirected graph.

A subgraph of a graph \(G = (V, E)\) is a graph \(G_i = (V_i, E_i)\) such that \(V_i \subseteq V\) and for all vertices \(i, j \in V_i, (i, j) \in E_i\) exactly when \((i, j) \in E\). The minimum spanning tree
A parallel algorithm for updating minimum spanning trees

381

Cforest) (abbreviated MST (MSF)) of a weighted, undirected graph is a spanning tree (forest) such that the sum of the weights of its edges is not larger than that of any other spanning tree (forest) on the graph.

Let $T$ be a rooted tree, with a vertex $r$ as the root. Edges of $T$ can be oriented (or directed) by a pre-order traversal of $T$ beginning at $r$. Orientation of the edges defines a unique father for each vertex of $T$. We denote the father of a vertex $v$ by $F(v)$. Observe that $F(r)$ is $r$ itself. The least common ancestor of a pair of vertices $u$ and $v$ ($\text{LCA}(u, v)$) in $T$ is the vertex that is furthest from $r$ and lies on both the path from $u$ to $r$ and the path from $v$ to $r$.

The subtree of $T$ at a vertex $u$, denoted by $T(u)$, is a subgraph of $T$ consisting of the vertices whose ancestor is $u$. Vertex $u$ is the root of $T(u)$. Evidently, $T(r) = T$. Let $\text{size}(u)$ be the number of vertices in $T(u)$.

An Euler tour of a tree is the set of arcs obtained by replacing each tree edge by a pair of directed antiparallel edges. If the tree is rooted, then the arcs in the Euler tour may be ordered so that together they form a closed walk beginning and ending at the root of the tree. The Euler number of an arc $e$ of the Euler tour is the position at which $e$ occurs in it, and is denoted by $\text{EulerNum}(e)$. The arcs with Euler numbers $\text{EulerNum}(e) + 1$ and $\text{EulerNum}(e) - 1$ are denoted by $\text{EulerNext}(e)$ and $\text{EulerPrev}(e)$ respectively. For every tree edge $(u, v)$, the Euler tour contains two arcs $(u, v)$ and $(v, u)$. We refer to $(u, v)$ as a forward arc (respectively reverse arc) if $\text{EulerNum}((u, v))$ is less than (respectively greater than) $\text{EulerNum}((v, u))$.

We assume familiarity with parallel algorithms for finding the Euler tour of a tree and for computing various functions on trees, as described by Tarjan and Vishkin in [12].

3. Overview of the algorithm

Let $T = (V, E)$ represent the old MST, $w$ the newly introduced vertex, and $E' = \{(w, u) | u \in V\}$ the set of newly introduced edges. For convenience we assume that there is an edge between every vertex of $T$ and the newly introduced vertex $w$. If this is not the case, we may assume the existence of dummy edges having the maximum possible weight (denoted by $+\infty$) in their place. The vertex insertion problem is equivalent to computing the minimum spanning tree of the graph $G = (V \cup \{w\}, E \cup E')$.

Our algorithm is based on a recursive divide-and-conquer strategy. $T$ is partitioned into $\sqrt{n}$ subtrees $T_0, T_1, \ldots, T_{\sqrt{n} - 1}, T_i = (V_i, E_i)$, by removing $(\sqrt{n} - 1)$ edges of $T$. The partitioning is done in such a way that each $T_i$ contains a significant fraction of the vertices of $T$. If $E_{\text{split}}$ is the set of edges of $T$ removed during the partition, then

$$V = \bigcup_{i=0, \ldots, \sqrt{n}-1} V_i \quad \text{and} \quad E = \left\{ \bigcup_{i=0, \ldots, \sqrt{n}-1} E_i \right\} \cup E_{\text{split}}.$$  

For convenience, assume that $\sqrt{n}, \sqrt[n]{n}$ etc. are all integers.
For the subsequent steps of the algorithm, define $G_i$, $0 \leq i \leq \sqrt{n} - 1$, to be the graph obtained by "splitting" vertex $w$ into $\sqrt{n}$ copies $w_i$, and replacing edge $(w, u)$ by the edge $(w_i, u)$ whenever $u \in V_i$. More formally,

$$G_i = (V_i \cup \{w_i\}, E_i \cup \{(w_i, u) | u \in V_i, (w, u) \in E'\}).$$

Finding the MST on $G_i$ is equivalent to solving the vertex insertion problem on $T_i$ (the new vertex being $w_i$). We compute the MST on each $G_i$ recursively. On

Edges between $w$ and the vertices of $T$ have been omitted for clarity.

Fig. 1(a). Degree-constrained tree $T$ and the new vertex $w$.

Fig. 1(b). Graphs obtained by partitioning $T$ and assigning a copy of $w$ to each tree.
A parallel algorithm for updating minimum spanning trees

Fig. 1(c). Forest of MSTs on graphs $G_0$, $G_1$, $G_2$ and $G_3$.

Fig. 1(d). (i) Tree $\hat{T}$ obtained by reintroducing the edges in $E_{\text{split}}$.

Cycle between

| $w_0$, $w_1$  | $(w_0 \ a \ e \ w_1)$   |
| $w_0$, $w_2$  | $(w_0 \ a \ e \ i \ m \ w_2)$   |
| $w_0$, $w_3$  | $(w_0 \ a \ e \ i \ k \ l \ m \ w_3)$   |
| $w_1$, $w_2$  | $(w_1 \ a \ i \ w_2)$   |
| $w_1$, $w_3$  | $(w_1 \ e \ i \ k \ l \ m \ w_3)$   |
| $w_2$, $w_3$  | $(w_2 \ i \ k \ l \ m \ w_3)$   |

LCA of pair

| $w_0$, $w_1$  | $w_0$   |
| $w_0$, $w_2$  | $w_0$   |
| $w_0$, $w_3$  | $w_0$   |
| $w_1$, $w_2$  | $w_1$   |
| $w_1$, $w_3$  | $e$     |
| $w_2$, $w_3$  | $i$     |

Fig. 1(d). (ii) Cycles when $w_i$ are merged

Fig. 1(d). (iii) LCAs of pairs $w_i$ and $w_j$, assuming $w_0$ as the root of $\hat{T}$. 
return from the recursive call, we have a forest of $\sqrt{n}$ MSTs, one on each $G_i$. The $\sqrt{n}$ trees are merged simultaneously to obtain the MST on $G$ as follows.

The edges in $E_{\text{split}}$ that had been removed in the process of partitioning are reintroduced into the forest of $\sqrt{n}$ MSTs just computed. This results in a graph we will call $\hat{T}$. It is easily observed that $\hat{T}$ is a tree. Vertices $w_i$ are next coalesced into a single vertex $w$ to yield a graph $\hat{G}$; i.e., each edge $(w_i, u)$, $0 < i < \sqrt{n} - 1$ of $\hat{T}$ is replaced by $(w, u)$ in $\hat{G}$. One may observe that $\hat{G}$ has the same set of vertices as the original graph $G$ and a subset of the edges of $G$. An edge of $G$ that is not in $\hat{G}$ cannot occur in the MST on $G$. Hence we complete the identification of the MST of $G$ by finding the MST on $\hat{G}$.

The MST on $\hat{G}$ is determined by removing the edge of maximal weight in each cycle in $\hat{G}$. Since $\hat{G}$ is obtained from $\hat{T}$ by coalescing vertices $w_i$ and since $\hat{T}$ is acyclic, every cycle of $\hat{G}$ is a path in $\hat{T}$ between some two vertices $w_j$ and $w_k$, $j \neq k$, $0 \leq j, k \leq \sqrt{n} - 1$. The $O(n)$ cycles so induced in $\hat{G}$ are removed in a two-phase algorithm as described in Section 5. In the first phase, the $O(n + \sqrt{n})$ vertex tree $\hat{T}$ is compressed to a tree with at most $2\sqrt{n}$ vertices. The second phase consists of identifying the edges of maximal weight on selected paths in the compressed tree. The latter edges correspond to maximal weight edges in cycles introduced when vertices $w_i$ of $\hat{T}$ are coalesced.

Figure 1 illustrates the operation of the algorithm by an example. A proof of the correctness of the procedure is presented in Section 5. The complete algorithm for the problem is presented in algorithm Vertex Insertion below.
Algorithm Vertex Insertion

Step 0: Preprocess the tree so that no vertex has degree greater than 3.

Step 1: Partition the degree-constrained tree $T$ of $n$ vertices into $\sqrt{n}$ subtrees $T_i$, $i = 0, \ldots, \sqrt{n} - 1$, by removing exactly $\sqrt{n} - 1$ edges so that each subtree is also a degree-constrained tree, with at most $n^{3/2-\varepsilon}$ vertices. $E_{\text{split}}$ denotes the set of edges of $T$ removed during the partition.

Step 2: Let $G_i$ be the graphs defined in (1) above. Compute recursively (and in parallel) the MST on each graph $G_i$, $i = 0, \ldots, \sqrt{n} - 1$, using the same algorithm (beginning at Step 1).

Step 3: Let $F$ be the forest of $\sqrt{n}$ MSTs on graphs $G_i$, $0 \leq i \leq \sqrt{n} - 1$. Let $\hat{T}$ be the graph obtained by augmenting $F$ with the edges in $E_{\text{split}}$ and let $\hat{G}$ be the graph obtained by coalescing the vertices $w_i$, $0 \leq i \leq \sqrt{n} - 1$, of $\hat{T}$. Obtain the MST on $\hat{G}$ by removing, simultaneously, the edge of maximal weight in each of the cycles of $\hat{G}$. (Up to $n - 2$ dummy edges may be introduced into the tree by step 0. However, no edge removed in Step 3 is a dummy edge. In fact, the edges indicated for removal by this step are removed directly from the original set of edges $\{E \cup E'\}$ to yield the updated MST.)

Step 0 is done to ensure that the tree $T$ may be broken into subtrees each containing an appropriate number of vertices. The details of this step are presented in Algorithm Regularize, in Section 6. The details of Step 1 are presented in Algorithm Partition in Section 4 and the details of Step 3 in Algorithm MergeTrees in Section 5. All of these steps can be implemented in $O(\lg n)$ time using $O(n)$ processors. The time complexity for Algorithm Vertex Insertion is thus bounded by

$$T \leq T_{\text{preprocessing}} + T(n),$$

where $T(n)$ is the solution of the recurrence:

$$T(n) \leq T_{\text{split}} + T(n^{3/2-\varepsilon}) + T_{\text{merge}}, \quad \varepsilon = \frac{1}{2} \lg 3,$$

$$T(1) = O(1).$$

$T_{\text{preprocessing}}$, $T_{\text{split}}$ and $T_{\text{merge}}$ are the time complexities for Steps 0, 1 and 3 respectively. Since the time complexities of each of these steps are bounded (see Sections 4-6) by $c \lg n$, $c > 0$, we obtain $T(n) = O(\lg n)$.

4. Split phase

Algorithm Vertex Insertion described in the previous section consists of three phases. The first of these phases transforms the input MST into a degree-constrained tree; and this phase is performed just once for the entire algorithm. We defer the description of this transformation to Section 6. In the present section, we describe the second phase, the split phase, which is concerned with splitting the degree-constrained tree into a forest of $\sqrt{n}$ subtrees by deleting $\sqrt{n} - 1$ edges. Edges are chosen for deletion in such a way that each subtree contains a significant fraction
of the vertices of the original tree. The resulting forest remains degree-constrained.
More precisely, we have the following specification for Algorithm *Partition*.

**Input:** A rooted degree-constrained tree, \( T = (V, E), |V| = n \); each vertex in \( V \) has degree 3 or less.

**Output:** A forest \( \{T_i = (V_i, E_i) | i = 0, \ldots, \sqrt{n} - 1\} \), and a set of edges \( E_{\text{split}} \) such that

(i) \( T_i \) is a subtree of \( T \),

(ii) \( n^{1-\epsilon} < |V_i| < n^{3/2-\epsilon} \), where \( \epsilon = \frac{1}{2} \log 3 \), and

(iii) \((\bigcup_i E_i) \cup E_{\text{split}} = E, E_i \cap E = \emptyset\);

(iv) each vertex in \( T_i \) has degree 3 or less.

The existence of such a partitioning of \( T \) is guaranteed by the separator theorem for the family of trees with a maximum degree 3 [8], which ensures that by removing exactly one edge from the \( n \)-vertex tree \( T \), we can split \( T \) into two subtrees \( T_i \) and \( T_j \) such that both \( T_i \) and \( T_j \) contain between \( \frac{1}{3}n \) and \( \frac{2}{3}n \) vertices. Algorithm *Partition* operates in \( O(\log n) \) time using \( n \) processors. Following a preprocessing phase which builds an Euler tour around the input tree \( T \), the algorithm consists of \( k = \frac{1}{2} \log n \) iterations, each requiring \( O(1) \) time. In each iteration every subtree obtained up to that iteration is split into two subtrees so that, after \( k \) iterations, \( T \) has been partitioned into \( \sqrt{n} \) subtrees, where the number of vertices in any subtree is between \( (\frac{1}{3})^k n \) and \( (\frac{3}{3})^k n \), i.e., between \( n^{1-\epsilon} \) and \( n^{3/2-\epsilon} \), \( \epsilon = \frac{1}{2} \log 3 \).

**Algorithm Partition**

\{Input: \( n \)-vertex rooted tree \( T_0 \). Every vertex in \( T_0 \) has degree 3 or less\}

1. \((a)\) build an Euler tour around \( T_0 \);

   \((b)\) determine \( \text{EulerNum}(e) \), the Euler tour number for each arc \( e \) on the tour, identify forward and reverse arcs, and determine the father \( F(u) \) of every vertex \( u \).

2. \((m = 0 \to \frac{1}{2} \log n - 1)\) do

   \((a)\) split \( T_i \) into trees \( X_i \) and \( Y_i \) of appropriate size using Procedure *SplitForest*;

   \((b)\) rebuild the Euler tours around \( X_i \) and \( Y_i \);

   \((c)\) rename \( X_i \) as \( T_{i \cdot 2^m} \) and \( Y_i \) as \( T_{i + 2^m} \)

end for

end for

Step 1(a) constructs an Euler tour around the given tree in \( O(\log n) \) time with \( O(n) \) processors using the technique described in [12]. All the information required in Step 1(b) may be readily computed from the Euler tour as described in [12].

Splitting the degree-constrained tree \( T_i \) of \( n_i \) vertices (Step 2(a)) involves identifying a single tree edge whose removal splits \( T_i \) into two subtrees of sizes between \( \frac{1}{3}n_i \) and \( \frac{2}{3}n_i \) vertices. This is done in \( O(1) \) time, to meet the \( O(\log n) \) time bound for
this phase of the algorithm, as shown in Procedure \textit{SplitForest} below. Every vertex $u$ of $T_i$ determines whether deleting the edge $(F(u), u)$ of $T_i$ will effect the desired split; if so, vertex $u$ flags itself as a \textit{candidate}. Obviously, $u$ is a candidate if and only if the number of vertices in its subtree lies between $\frac{1}{3}n_i$ and $\frac{2}{3}n_i$. In general, there may be several $(\frac{1}{3}n_i$ in the worst case) vertices that identify themselves as candidates. We need to choose exactly one such candidate vertex.

Theorem 4.1 below provides a method for resolving in $O(1)$ time the contention among vertices that have identified themselves as candidates. The theorem ensures that there exist \textit{at most two} vertices that are candidates and whose fathers are \textit{not} candidates. Furthermore, the two candidate vertices must be children of the same father.

\textbf{Theorem 4.1.} \textit{Let $T = (V, E)$ be a tree, rooted at $r \in V$, and of maximum degree 3. Let $|V| = n$. Then there exists at least one vertex $v_1$, and at most one other vertex $v_2$ ($v_1, v_2 \in V$) such that}

(i) $\frac{1}{3}n \leq \text{size}(v_1), \text{size}(v_2) \leq \frac{2}{3}n$, and

(ii) $\frac{2}{3}n < \text{size}(F(v_1)), \text{size}(F(v_2))$.

\textit{Furthermore, if two such vertices $v_1$ and $v_2$ exist, then $F(v_1) = F(v_2)$.}

\textbf{Proof.} Let $U = \{v \in V | \frac{1}{3}n \leq \text{size}(v) \leq \frac{2}{3}n\}$. The separator theorem \cite{8} ensures that $U$ is nonempty. Let $v_1 \in U$ be the vertex in $U$ that is closest to the root (ties between vertices at the same distance from the root may be broken arbitrarily). Since $\text{size}(v_1) \leq \frac{2}{3}n$, $v_1 \neq r$. Let $F(v_1) = u_1$. Evidently, $v_1$ and $u_1$ are distinct; and $\text{size}(u_1) > \text{size}(v_1)$. From the way $v_1$ is selected, (a member of $U$ closest to $r$), it follows that $u_1$ cannot be in $U$. Consequently, $\text{size}(u_1) > \frac{2}{3}n$. This establishes the existence of a vertex with the required property.

Let $v_2 \neq v_1$ be another vertex of $U$ such that $\text{size}(u_2 = F(v_2)) > \frac{2}{3}n$. Then $u_2$ cannot belong to $T(v_1)$, and, similarly $u_1$ cannot belong to $T(v_2)$. Let, if possible, $u_1$ and $u_2$ be distinct. Consider the possibility that $u_2$ is a descendant of $u_1$. Since $u_2 \notin T(v_1)$, $\text{size}(u_1) > \text{size}(u_2) + \text{size}(v_1)$, which is not possible as the latter sum exceeds $n$. Therefore, $u_1$ cannot be a descendant of $u_1$. Similarly, $u_1$ cannot be a descendant of $u_2$. That is, if $u_1$ and $u_2$ are distinct, then the subtrees of $T$ rooted at $u_1$ and $u_2$ are disjoint. However, this is impossible since both these subtrees have a size greater than $\frac{2}{3}n$. Therefore, $u_1 = u_2$ is the common father of the two vertices $v_1$ and $v_2$. \hfill $\square$

The procedure for splitting each tree $T_i$ of a forest into two parts (Step 2(a) of Algorithm \textit{Partition}) is described below.

\textbf{Procedure \textit{SplitForest}}

\{Split each of the $2^n$ trees $T_i$, $i = 0, 1, \ldots, (2^n - 1)$, into two subtrees of appropriate size\}

\textbf{for each vertex $u$ do in parallel}

\hspace{1cm} let $u \in T_i$; let $n_i$ be the number of vertices in $T_i$;
if \( \frac{1}{3} n_i \leq \text{size}(u) \leq \frac{2}{3} n_i \) then \( \text{candidate}(u) = \text{TRUE} \);

end for

for each vertex \( u \) do in parallel

if \( \text{candidate}(u) \land \neg \text{candidate}(F(u)) \) then begin

if \( (u \text{ is the only child or the left-child of } F(u)) \) then split \( T_i \) be removing the edge \( (F(u), u) \)

else if \( \neg \text{candidate}(\text{left-child of } F(u)) \) then split \( T_i \) by removing the edge \( (F(u), u) \).

end

end for

Each execution of Procedure SplitForest takes \( O(1) \) time, as shown below. Let \( e_f \) and \( e_r \) be the forward and reverse Euler tour arcs respectively, corresponding to the tree edge \( (u, F(u)) \). Then, we state the following:

1. The number of vertices in the subtree of \( u \), \( \text{size}(u) \), is readily calculated as follows:

\[
\text{size}(u) = \frac{1}{2} (\text{EulerNum}(e_f) - \text{EulerNum}(e_r) + 1)
\]

2. It is easily checked, using the Euler tour, whether a vertex is a right-child of its father.

3. Splitting \( T_i \) by removing edge \( (F(u), u) \) involves making \( u \) the root of the newly created tree, \( T_{i+2^m} \). All vertices not in the new subtree retain their old tree identifier \( T_i \) (Fig. 2(a)). Following the split, every vertex updates the information about the i.d. and the size of the tree to which it belongs. If the Euler number of the forward Euler tour arc directed into a vertex is less than \( \text{EulerNum}(e_f) \) or greater

\[ e_f = < b, d > \]
\[ e_r = < d, b > \]

\[ \text{Size} \left( T_i \right) = 8 \]

Fig. 2(a). \( T_i \) (before split).
than \( \text{EulerNum}(e_i) \), then the vertex belongs to the tree \( T_i \) with no change in the root; else it belongs to the newly created subtree \( T_{i+2^m} \) rooted at \( u \). The sizes of the two trees are given by

\[
\text{size of tree } T_{i+2^m} = \frac{1}{2}(\text{EulerNum}(e_i) - \text{EulerNum}(e_f) + 1),
\]
\[
\text{size of tree } T_i = \text{previous size of tree } T_i - \text{size of tree } T_{i+2^m}.
\]

(4) The final step of Algorithm Partition involves rebuilding the Euler tours around \( T_i \) and \( T_{i+2^m} \). The Euler tours of the trees \( T_i \) and \( T_{i+2^m} \) can be easily updated by the following operations (see Fig. 2(b)):

\[
\text{EulerNext}(\text{EulerPrev}(e_i)) = \text{EulerNext}(e_i),
\]
\[
\text{EulerPrev}(\text{EulerNext}(e_i)) = \text{EulerPrev}(e_i),
\]
\[
\text{EulerPrev}(\text{EulerNext}(e_i)) = \text{null},
\]
\[
\text{EulerNext}(\text{EulerPrev}(e_i)) = \text{null}.
\]

(5) The Euler numbers of the arcs are updated as follows:

(a) For an arc \( e \) in tree \( T_{i+2^m} \):

\[
\text{EulerNum}(e) = \text{EulerNum}(e) - \text{EulerNum}(e_f);
\]

(b) For an arc \( e \) in tree \( T_i \):

\[
\text{if } (\text{EulerNum}(e) > \text{EulerNum}(e_f)) \text{ then }
\]
\[
\text{EulerNum}(e) = \text{EulerNum}(e) - 2 \times \text{size of } T_{i+2^m}
\]
\[
\text{else } \text{EulerNum}(e) = \text{EulerNum}(e).
\]

All the update operations following a split, and the split itself, can thereby be accomplished in \( O(1) \) time using \( O(n) \) processors. The time complexity of Algorithm Partition is therefore \( O(\lg n) \).

![Fig. 2(b). \( T_i \) after split.](image-url)
5. Merging the $\sqrt{n}$ (sub)solutions

This phase of the algorithm is concerned with merging the $\sqrt{n}$ MSTs available upon return from a recursive call. The input to the Algorithm \textit{MergeTrees}, consists of the following:

(i) A forest of $\sqrt{n}$ trees $T_i = (V_i, E_i)$, $i = 0, 1, \ldots, \sqrt{n} - 1$. Each tree $T_i$ has a distinguished vertex $w_i$, and the total number of vertices in the forest equals $n + \sqrt{n}$.

(ii) A set $E_{\text{split}}$ of $\sqrt{n} - 1$ edges. An edge in $E_{\text{split}}$ connects a vertex in $V_i$ to a vertex in $V_j$, $i \neq j$, in such a way that the graph $\hat{T} = (V, E)$, $V = (\bigcup_i V_i)$, $E = (\bigcup_i E_i) \cup E_{\text{split}}$ is a tree.

(iii) A unique weight associated with each edge in $E$.

The output of Algorithm \textit{MergeTrees} is the minimum spanning tree on the graph $\hat{G}$ obtained from $\hat{T}$ by coalescing all distinguished vertices, $w_i$, $i = 0, 1, \ldots, \sqrt{n} - 1$, into a single vertex $w$. Coalescing of vertices $w_i$ is performed by replacing every edge $(w_i, u)$ in $\hat{T}$ by the edge $(w, u)$ in $\hat{G}$. The construction of the minimum spanning tree on $\hat{G}$ is accomplished by identifying the cycles in $\hat{G}$, and then breaking these cycles by removing the edge of maximum weight (abbreviated MWE) in each of them, as described in Algorithm \textit{MergeTrees} below.

\textbf{Algorithm \textit{MergeTrees}}

\begin{algorithmic}
  \FOR {each pair $i, j$, $i, j = 0, 1, \ldots, \sqrt{n} - 1$, $i \neq j$} \DO \IN \PARALLEL
    \STATE (i) find the edge of maximum weight on the path in $\hat{T}$ between $w_i$ and $w_j$
    \STATE (ii) delete the edge identified in Step (i) from $\hat{T}$
  \ENDFOR
\end{algorithmic}

In the following discussion we will show that the Algorithm \textit{MergeTrees} does result in a minimum spanning tree of $\hat{G}$. Following that, we describe the details of its implementation.

\textbf{Lemma 5.1.} Every edge that is removed by \textit{MergeTrees} is the MWE on a cycle of $\hat{G}$.

\textbf{Proof.} The path in $\hat{T}$ between vertices $w_i$ and $w_j$, $i \neq j$, must be of the form $(w_{k_0}, \ldots, w_{k_t}, \ldots, w_{k_s}, \ldots, w_{k_{s+1}})$ where $s > 0$, $k_0 = i$ and $k_{s+1} = j$. The MWE on this path must occur in the subpath between $w_{k_t}$ and $w_{k_{t+1}}$ for some $t$, $0 \leq t < s$, and must be the MWE on this subpath as well. But every path $(w_{k_t}, x_1, \ldots, x_r, w_{k_{t+1}})$ in $\hat{T}$ corresponds to the cycle $(w, x_1, \ldots, x_r, w)$ in $\hat{G}$. \hfill $\square$

\textbf{Lemma 5.2.} Every cycle in $\hat{G}$ consists of exactly the edges in the path in $\hat{T}$ between some pair of vertices $w_i$ and $w_j$, $i \neq j$.

\textbf{Proof.} Every cycle in $\hat{G}$ must include the vertex $w_i$; otherwise this cycle would also be present in $\hat{T}$, which is impossible since $\hat{T}$ is a tree. Therefore, without loss of
generality, let \((w, x_1, \ldots, x_i, w)\) be a cycle in \(\hat{G}\), with \(x_j \neq w, \forall i, 1 \leq i \leq t\). Let \(x_i \in V_i\). Then \(x_i \in V_j\) for some \(j, j \neq i\); for otherwise \((w_i, x_1, \ldots, x_i, w_i)\) constitutes a cycle in \(\hat{T}\). We have shown then that the above cycle consists of the edges \((w_i, x_1, \ldots, x_i, w_j)\), i.e., the edges constituting a path between \(w_i\) and \(w_j\) for some \(j \neq i\). Since all of these edges occur in \(\hat{T}\), the path occurs in \(\hat{T}\), and it is the only such path in \(\hat{T}\) since \(\hat{T}\) is acyclic. \(\Box\)

Theorem 5.3 below provides the final arguments for the correctness of Algorithm MergeTrees.

**Theorem 5.3.** Algorithm MergeTrees produces an MST of \(\hat{G}\).

**Proof.** (i) We first show that all cycles in \(\hat{G}\) are removed by the algorithm. From Lemma 5.2 it follows that every cycle in \(\hat{G}\) consists of the edges in the path between some \(w_i\) and \(w_j\), \(i \neq j\), in \(\hat{T}\). The algorithm removes one edge from the path between every pair of distinct vertices \(w_i\) and \(w_j\) and therefore breaks all cycles in \(\hat{G}\).

(ii) We now show that vertices of \(\hat{G}\) remain connected at the end of the algorithm. From Lemma 5.1, it follows that every edge removed from \(\hat{G}\) was the MWE in some cycle in \(G\). Suppose that the removal of the edges resulted in the formation of more than one connected component. Consider one such connected component, say \(C\). Let \(X\) be a set of edges in \(\hat{G}\) that connect vertices of \(C\) to vertices of \(\hat{G} - C\). Since \(\hat{G}\) was initially a connected graph, the algorithm must have removed all the edges of \(X\) from \(\hat{G}\). Consider the edge \(e_{\min}\) with the smallest weight among the edges of \(X\). Since this edge was removed by the algorithm, all edges on any cycle in which it occurred must have had a smaller weight. In particular, there must have been an edge in \(X\) that had a weight smaller than that of \(e_{\min}\), which is not possible. Therefore, it is not possible for vertices of \(\hat{G}\) to be separated into more than one connected component. Thus at the end of the algorithm, the remaining graph is a connected, acyclic graph, i.e., a tree.

(iii) Since every edge removed by the algorithm has a larger weight than all the other edges in any cycle in which it occurs, it follows that the tree is indeed a minimum spanning tree. \(\Box\)

**Theorem 5.4.** In an \(n\)-vertex rooted tree with \(k\) marked vertices, \(2 < k \leq n\), the number of vertices that are the least common ancestors of some pair of marked vertices is at most \((k-1)\).

**Proof.** We prove a somewhat stronger result, namely that the theorem holds for three marked vertices and that each new marked vertex results in the addition of at most one vertex to the set of LCA vertices.

It can be easily verified that the theorem is true for \(k = 3\). Let \(a_1, a_2, \ldots, a_m\) be \(m\) marked vertices. If possible, let the introduction of \(a_{m+1}\) cause two (or more) vertices to be added to the LCA set. Specifically, consider \(l_i = LCA(a_{m+1}, a_p)\) and
$l_2 = \text{LCA}(a_{m+1}, a_q)$, $p \neq q$, $1 \leq p$, $q \leq m$, that are two of the new LCAs over the set of marked vertices. But then $l_1$, $l_2$ and LCA($a_p$, $a_q$) form a 3-vertex LCA set over the set of three marked vertices $\{a_p, a_q, a_{m+1}\}$, which is not possible.

We now describe the details involved in implementing Algorithm MergeTrees. In particular, we are interested in determining the edge of maximum weight on each of the $(^{n}_2)$ paths between the pairs of vertices $w_i, w_j, i \neq j$ in $T$. The procedure employed consists of two parts. In the first part (Steps (A.1) through (A.6)), the tree $\hat{T}$ containing $(n + \sqrt{n})$ vertices is compressed to a tree having at most $2\sqrt{n}$ vertices, including the $\sqrt{n}$ vertices $w_i, i = 0, \ldots, \sqrt{n} - 1$. The compressed tree is such that the MWE on the path between any two vertices in it equals the MWE on the path between the two vertices in $\hat{T}$. In the second part (Step (B)), the MWE on the path between every pair of vertices in the compressed tree is determined. The details are presented in Procedure DetermineMWE below.

**Procedure DetermineMWE**

(A.1) Let $r \in V$ be an arbitrarily chosen root of $\hat{T}$.

(A.2) Construct an Euler tour around $\hat{T}$. For $u \in V$, $u \neq r$, determine $F(u)$, the father of $u$. Let $F(r) = \text{null}$.

(A.3) For each pair of distinct vertices $w_i, w_j, i \neq j$ in $\hat{T}$, determine LCA($w_i, w_j$), the least common ancestor of that pair in $\hat{T}$.

(A.4) Flag all vertices $w_i, i = 0, \ldots, \sqrt{n} - 1$, the root $r$ and all vertices found to be LCAs of some pair of vertices in Step (A.3) as special vertices.

(A.5) For each vertex in $\hat{T}$, find the MWE on the path between the vertex and the first special vertex on the path between the vertex and the root using the procedure below.

```
for (each vertex $u \in \hat{T}$) do in parallel
begin
  NEXT(u) = F(u)
  WT(u) = \text{weight-of-edge}((u, F(u))).
  Max-Wt-Edge(u) = (u, F(u)).
  for ($k = 1$ to $\frac{1}{2} \lg n$) do
  begin
    if (($NEXT(u) \neq \text{null}$) $\wedge$ ($NEXT(u)$ is not a special vertex))
    then begin
      if $WT(NEXT(u)) > WT(u)$
      then begin
        $WT(u) = WT(NEXT(u))$.
        $Max-Wt-Edge(u) = (NEXT(u), F(NEXT(u)))$.
      end
      $NEXT(u) = NEXT(NEXT(u))$
    end
  end
endfor
```
A parallel algorithm for updating minimum spanning trees

At the end of the above procedure, \( \text{NEXT}(u) \) is the first special vertex on the path from \( u \) to the root. \( \text{Max-Wt-Edge}(u) \) is the MWE on the path from \( u \) to \( \text{NEXT}(u) \). The weight of the MWE is recorded in \( W_T(u) \).

(A.6) Construct the rooted weighted tree \( T' = (S, W) \) with root \( r \), where

\[
S = \{ u \in V | u \text{ is a special vertex of } \hat{T} \},
\]

\[
W = \{(u, \text{NEXT}(u))\},
\]

weight-of-edge\((u, \text{NEXT}(u))\) = \( W_T(u) \).

Additionally, define for each edge \((u, \text{NEXT}(u))\) in \( W \), an alias (an edge in tree \( \hat{T} \)):

\[
\text{alias}((u, \text{NEXT}(u))) = \text{Max-Wt-Edge}(u).
\]

(B) Determine \( \text{MWE}(w_i, w_j) \), the edge of maximum weight between each pair \( w_i, w_j, i \neq j \) of vertices in \( T' \). Record \( \text{alias}(\text{MWE}(w_i, w_j)) \) as the edge of maximum weight on the path in \( \hat{T} \) between vertices \( w_i, w_j \).

Comments: No more than \( 2\sqrt{n} \) vertices are marked special in Step (A.4), as evidenced by Theorem 5.4. At the termination of Step (A.5), we have effectively "compressed" \( \hat{T} \). Every path with intermediate vertices that were not special has been replaced with a single edge between the special vertices that are the endpoints of the path. All vertices that were not marked special are now leaves of the compressed tree, and we may delete all of them without affecting the connectivity of the tree. Tree \( T' \) is this compressed tree. The weight of an edge between a vertex and its father in \( T' \) equals that of the edge of maximum weight on the path between the two vertices in \( \hat{T} \). It follows by a simple inductive argument that the (alias of the) edge of maximum weight on the path between any two vertices in \( T' \) is the edge of maximum weight on the path between the two vertices in \( \hat{T} \).

Steps (A.2) and (A.3) of the procedure can be done in \( O(\lg n) \) time using the technique in [12]. By assigning one processor to each pair of distinct \( w_i, w_j \), and performing a binary search on the arcs of the Euler tour, the LCA of the pair can be determined in \( O(\lg n) \) time [12, 14]. It is easy to see that steps (A.5) and (A.6) can also be done in \( O(\lg n) \) time using \( O(n) \) processors.

Tree \( T' \) constructed in Step (A.6) contains at most \( 2\sqrt{n} \) vertices. The MWE between all \( O(n) \) pairs of vertices in the tree can be computed, in Step (B), using the well-known technique described in [10, 13] in \( O(\lg n) \) time. The entire procedure can therefore be done in \( O(\lg n) \) time using \( O(n) \) processors.

6. Transformation to a degree-constrained tree

This section describes the transformation of a given \( n \)-vertex tree with vertices of arbitrary degree into an \( O(n) \)-vertex degree-constrained tree in which no vertex
has degree more than 3. The transformation is achieved with the addition of at most $n - 2$ dummy vertices and $n - 2$ dummy edges (of weight $-\infty$) to the given tree.

In Fig. 3(a), we have a sample input tree $T_{in}$. The transformation of $T_{in}$ into a degree-constrained tree $T_{out}$, shown in Fig. 3(b), involves the addition of 2 dummy vertices—7' and 8'—and 2 dummy edges—$(10, 7')$ and $(7', 8')$. In addition, the edges
(10, 7), (10, 8), and (10, 9) of $T_{in}$ are replaced by the new edges $(7', 7)$, $(8', 8)$ and $(8', 9)$ respectively in $T_{out}$.

Let $u$ be a vertex in $T_{in}$ with children $u_1, u_2, \ldots, u_k$, $k > 2$. Perform the following operations:

(i) create dummy vertices $u_2', u_3', \ldots, u_{k-1}'$,
(ii) add dummy edges (of weight $-\infty$): $(u, u_2'), (u_p, u_{p+1})$ $\forall p: 2 \leq p \leq k - 2$, and
(iii) replace edges $(u, u_2)$ and $(u, u_k)$ by edges $(u_2', u_2)$ and $(u_{k-1}', u_k)$ respectively, and the edges $(u, u_p)$ by the edges $(u_p', u_p)$ $\forall p: 2 \leq p \leq k - 1$.

Effectively, we make $k - 2$ copies of the original vertex $u$ and link $u$ and all its copies in a path consisting of dummy edges. The $k - 2$ copies are the dummy vertices $u_2', \ldots, u_{k-1}'$. Each child vertex $u_i$ of $u$ is linked to either $u$ or to one of these dummy vertices by an edge of the same weight as the edge between $u$ and $u_i$ in $T_{in}$.

We now present the procedure for transforming a given tree $T_{in}$ to a degree-constrained tree $T_{out}$.

Algorithm Regularize

(1) Construct an Euler tour around $T_{in}$. For each vertex $u$ in $T_{in}$:

(i) identify $F(u)$, and,
(ii) determine preorder($u$), the preorder number of $u$.

Denote the child of $F(u)$ that is visited in the Euler tour just before $u$ by $\hat{u}$.

(2) Comment: Identify the first and last child of every vertex. The first (respectively last) child is the vertex that is visited by the Euler tour before (respectively after) any of the other children are visited. A vertex that is neither the first, nor the last child of its father is termed an intermediate child. If a vertex is the only child, it is classified as a first child. This information is recorded in an array $MARK[\ ]$.

for each arc $(v, u)$ in the Euler tour where $F(u) = v$ do in parallel

(a) $MARK[u] = \text{intermediate-child}$ \{initialize all children as intermediate\}
(b) if $(u, v) = \text{EulerPrev}((v, F(v)))$ then $MARK[u] = \text{last-child}$
(c) if $(\text{EulerPrev}((v, u)) = (F(v), v))$ then $MARK[u] = \text{first-child}$
endfor..

(3) Comment: Build an adjacency list representation for $T_{out}$. The output is recorded in a table $ADJ[\ ]$. Each entry, $ADJ[u]$, has three fields, $ADJ[u].F$, $ADJ[u].LC$, and, $ADJ[u].RC$, which will contain the i.d.s of the father, the left-child, and the right-child of vertex $u$ respectively. Each vertex $u$ that is an intermediate child is replaced by two vertices, $u$ and $u'$. I.d. $u'$ is uniquely chosen to be preorder($u$) + $n$.

for each arc $(v, u)$ in the Euler tour of $T_{in}$, $F(u) = v$, do in parallel

begin

Comment: determine new father of $u$ in $T_{out}$. Record this in $temp[u]$.

case $MARK[u]$ of:

first-child: $temp[u] = u$

intermediate-child or last-child:
if \((MARK[\hat{u}] = \text{first-child})\) then \(temp[u] = v\) else \(temp[u] = (\hat{u})'\)

end case

Comment: build the adjacency table, \(ADJ[ ]\).

case \(MARK[u]\) of:

  first-child:
  \(ADJ[u].F = temp[u]\)
  \(ADJ[temp[u]].LC = u\)

  last-child:
  \(ADJ[u].F = temp[u]\).
  \(ADJ[temp[u]].RC = v.\)

  intermediate-child:
  Comment: connect \((u, u')\) pair
  \(ADJ[u].F = u'\)
  \(ADJ[u'].LC = u\)
  \(ADJ[u'].F = temp[u]\)
  \(ADJ[temp[u]].RC = u'\)

end case

end for

Comments: Step 1 can be done in \(O(\lg n)\) time using \(O(n)\) processors [12]. Steps 2 and 3 take \(O(1)\) time with \(O(n)\) processors.

7. Conclusions

In this paper, we described a new parallel algorithm for updating the MST of an \(n\)-vertex graph following the addition of a new vertex. The algorithm computes the MST by breaking cycles in the underlying graph. By employing a recursive divide-and-conquer technique, the algorithm reduces the number of cycles that it must examine from the maximum possible \(O(n^2)\) to only \(O(n \lg \lg n)\). By carefully balancing the depth of the recursion with the number of cycles that the algorithm examines, a time complexity of \(O(\lg n)\) using \(O(n)\) processors was achieved on a CREW PRAM. To implement the algorithm efficiently, several subproblems, which are interesting in their own right, were addressed and efficient algorithms for them were presented. The performance of the algorithm compares favorably with both the start-over algorithm for MST computation on a sparse graph, which requires \(O(n)\) processors and \(O(\lg^2 n)\) time, as well as the known incremental algorithm [9] that requires \(O(n^2)\) processors and \(O(\lg n)\) time on the same model.

References

A parallel algorithm for updating minimum spanning trees


