Chapter 10

The Minimum Spanning Tree Problem

10.1 Introduction

Suppose that we have an undirected weighted graph on n vertices with m edges. Our goal is to find the minimum spanning tree (MST). We assume that all edge weights are distinct. It is known that if the graph is connected and all edge weights are distinct then there is a unique minimum spanning tree. If the graph is not connected, then there is a unique spanning forest. For simplicity of exposition, we will assume that the underlying graph is connected.

The problem of finding minimum spanning tree is a canonical problem for which a greedy approach works. In this chapter, we first present two greedy sequential algorithms: Kruskal's algorithm and Prim's algorithm. Both of these algorithms are sequential in nature because they pick one edge at a time. They are greedy in different ways. Kruskal's algorithm chooses the least cost edge that is feasible as its next edge. Prim's algorithm chooses the next vertex that can be added to the existing tree with the least cost. Next, we discuss parallel algorithms for the minimum spanning tree problem. We first discuss a strategy to parallelize Prim's algorithm. In particular, instead of adding and exploring one vertex to the tree at a time, we show a strategy which allows multiple vertices to be added and explored in parallel. We also discuss Boruvka's algorithm which also chooses multiple edges at a time. Boruvka's algorithm maintains multiple *fragments* and adds edges to all of them in every iteration. The notion of a *fragment* is crucial in understanding MST algorithms. A fragment is simply a subtree of the MST. Consider the graph in Fig. 10.1. The minimum spanning tree in this graph corresponds to the edges $\{2, 3, 4, 7\}$. The subtree formed by edges 3 and 4 is a fragment with three vertices $\{a, b, c\}$ and two edges $\{(a, c), (b, c)\}$.

A crucial property of MST is as follows.

Lemma 10.1 *Let F be a fragment. Let e be the edge with minimum weight that is outgoing from F. Then, $F \cup \{e\}$ is also a fragment.*

Proof: In the minimum spanning tree T, there must be at least one edge going out of the fragment F. Let that edge be f. If we add e to T and remove f, we get another tree T' with lower weight than T, a contradiction because we assumed that T is the minimum spanning tree.

■

In the fragment formed by edges with weights 3 and 4, there are three outgoing edges — edges with weight 7, 9 and 11. The edge 5 is not outgoing since it connect vertices that are part of the fragment. According to Lemma 10.1, the edge with weight 7 can be added to the edges with weight 3 and 4 to grow the fragment.

10.2 Kruskal's Algorithm

Kruskal's algorithm is a canonical greedy algorithm for the minimum spanning tree problem. It chooses edges one at a time in a greedy fashion. Let T be the set of edges chosen by the algorithm at any stage. The algorithm maintains the invariant that T does not have any cycle. Initially T is empty and therefore trivially satisfies the invariant. The algorithm considers the edges in the increasing order of weights. For this reason it is sometimes also known as the *shortest-edge-next* algorithm. Suppose that the algorithm has chosen edges in T so far. To grow T, it finds the least weight edge that does not form a cycle with existing edges in T. If any edge e forms a cycle with T, then it is rejected and the algorithm considers the least weight edge of the remaining edges.

Algorithm Kruskal: Kruskal's Algorithm

```
1  Input: Undirected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  var
4      T, Rejected: set of edges initially {};
5  while (|T| < n − 1) do
6      if T ∪ Rejected = E then return null; // no spanning tree in the graph
7      e := the least weight edge that is not in T ∪ Rejected
8      if e forms a cycle in T then
9          Rejected := Rejected ∪ {e}
10     else T := T ∪ {e}
11 endwhile
12 return T
```

In Kruskal, the variable T keeps the set of all edges that are chosen and the variable *Rejected* keeps the set of all edges that are rejected. Consider the graph shown in Fig. 10.1. Kruskal's algorithm will first choose the edge with weight 2 since it has the least weight. It then chooses the edges with weight 3 and 4 because these edges do not form any cycle. The next edge has weight 5. However, this edge needs to be rejected because it forms a cycle with the already chosen edges with weight 3 and 4. The algorithm then chooses the edge with weight 7 and terminates.

The algorithm is dominated by the cost of sorting the edges $O(m \log n)$. Checking whether the edge e forms a cycle with T can be done efficiently using find-union data structure. The reader is referred to [CLRS01] for more details on the efficient implementation of the algorithm.

We give a parallel version for Kruskal's algorithm based on the simple boolean lattice of all edges. Our distributive lattice is the boolean lattice formed from all edges. We assume that edges are given to us in the increasing order. Then, an edge e_j between vertices v_k and v_l is in the minimum spanning tree iff there is no path between vertices v_k and v_l using edges $e_1 \ldots e_{j-1}$. Note that when j is 1, e_j corresponds to the lightest edge and is always in the minimum spanning tree. The algorithm is shown in Fig. LLP-Kruskal. The parallel time complexity of this algorithm is $O(m)$ where m is the number of edges in the graph if we are given edges in the sorted order.

Algorithm LLP-Kruskal: Finding the minimum spanning tree in a graph .

```
1 var G: array[1..m] of {0, 1};
2 // Edges G is assumed to be in increasing order of weights
3 init ∀j : G[j] := 0;
4 forbidden(j) ≡ e_j = (v_k, v_l) and there exists no path from v_k to v_l with edges 1..j − 1
5 advance(j) G[j] := 1
```

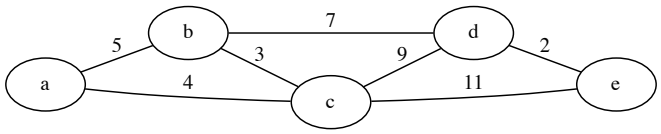

Figure 10.1: An undirected weighted graph

10.3 Prim's Algorithm

Prim's algorithm is also a greedy algorithm. It builds the minimum spanning tree by increasing the size of a single *fragment* by adding the minimum weight outgoing edge of the fragment. It simply exploits the Lemma 10.1 to increase the size of fragment until it becomes the MST. At any stage, Prim's algorithm has a fragment F. It finds the minimum outgoing edge from that fragment e. This edge can be viewed as the edge from the fragment to its nearest neighbor. Therefore, this algorithm is sometimes also known as the *nearest-neighbor-next* algorithm. To find the nearest neighbor, every vertex v maintains a label d which corresponds to the cost of adding v to the fragment. At every iteration, the algorithm chooses the vertex v with the minimum d value and adds it to the fragment. The array $fixed$ keeps track of the vertices in the fragment. Whenever, a new vertex v is *fixed* and added to the fragment, the d values for all adjacent vertex v' are updated as follows. We check whether the weight of the edge (v, v') is lower than the previous value of $d[v']$. If this is true, then $d[v']$ is updated to $w[v, v']$. We also use G pointer with each node which keeps track of the G node v that is responsible for the d value of v'.

Algorithm Prim: Finding a Minimum Spanning Tree (MST)

```
1  Input: Undirected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  var
4       d: array[0..n − 1] of real initially ∞; //d[v] is the cost to add vertex v
5       G: array[0..n − 1] of int initially −1; // G[v] is the node that corresponds to d[v] cost
6       fixed : array[0..n − 1] of boolean initially false; //vertices whose d value is fixed
7       T: set of edges initially {};
8  d[0] := 0;
9  while (|T| < n − 1) do
10      v := arg min_i{d[i] | ¬fixed[i]} ;
11      if d[v] = ∞ then return null;// no spanning tree in the graph
12      if G[v] ≠ −1 then add (v, G[v]) to T
13      fixed[v] := true;
14      forall (v, v') ∈ E:
15           if w[v, v'] < d[v'] then
16                d[v'] := w[v, v'];
17                G[v'] := v;
18      end // forall
19 end  //while
```

Consider the graph shown in Fig. 10.1 again. For Prim's algorithm, we start from a fixed node. Suppose we start from the vertex a. Then, the nearest neighbor is c with the cost of 4. The next nearest neighbor to the fragment with vertices $\{a, c\}$ is the vertex b. The cost of adding b is 3. At this point, we have vertices $\{a, b, c\}$ in the fragment. The cost to add vertex d is 7 and to add the vertex e is 11. We add the vertex d to our fragment with the cost 7. Finally, e is added with the cost 2. Hence, the edges are added to the tree in the order $4, 3, 7, 2$. Note that the set of edges chosen are identical to Kruskal's algorithm even though the order in which they are chosen is different. This is not surprising because there is a unique minimum spanning tree when all edge weights are unique (Problem 1).

The step of finding the vertex v with minimum d value can be done either by simply traversing the array d or by maintaining d in a heap. If we simply traverse the array d to find the minimum, the work complexity of the above algorithm is $O(n^2 + e)$. In every iteration of the while loop, we perform $O(n)$ work for finding the minimum and there are $O(n)$ iterations of the loop. The work for processing edges over all iterations is $O(e)$ because every edge is processed at most once. If we use a heap to store d values of all the vertices that are not fixed, we require $O(m)$ insertions on the heap resulting in $O(m \log n)$ work complexity. This approach results in better work complexity when the graph is sparse and m equals $O(n)$.

10.4 An LLP based algorithm for Prim

We now show an LLP based algorithm to find a minimum spanning tree rooted at a fixed vertex v_0. As before, we assume that all edge weights are unique. Every node other than v_0 has to choose a G edge. It keeps all its edge in the sorted order and begins by choosing its least edge as the proposed edge as its G edge. For the graph in Fig. 10.1, with root as the vertex a, we get the following choices of edges for other vertices.

```
b: 3, 5, 7
c: 3, 4, 9, 11
d: 2, 7, 9
e: 2, 11
```

We let $G[i]$ be the edge chosen by the node i. Thus, initially G vector is $\{3, 3, 2, 2\}$. Observe that the set of all possible combinations of edges forms a distributive lattice. This lattice is ordered by our desired objective function, i.e., if there are two distinct vectors G and H that form spanning trees rooted at v_0 and G is the minimum spanning tree then G is less than H in the lattice. Since every node except v_0 has exactly one outgoing edge, by following the chosen edge from every node we either reach v_0, or we end up in a cycle. We define a vertex to be *fixed* if by traversing the path starting from the edge proposed by that vertex leads to v_0. The vertex v_0 is trivially fixed. It is clear that the edge proposed by a non-fixed vertex can only lead to a non-fixed vertex and the edge proposed by a fixed vertex can only lead to a fixed vertex. Thus, any G partitions the set of vertices into *fixed* and *non-fixed* vertices. In our example, the initial vector $\{3, 3, 2, 2\}$ makes vertex a as fixed and all other vertices $\{b, c, d, e\}$ as non-fixed.

We define the set of cut edges between these two partitions as follows:

$$E'(G) := \{(i, k) \in E \mid fixed(i, G) \wedge \neg fixed(k, G)\}$$

In our example, we get the following edges as $E'(G) = \{(a, b), (a, c)\}$.

Let $x = (i, j)$ be the edge in E' with minimum $w[i, j]$. For our example, it is (a, c) with edge weight of 4. We claim that process j is forbidden in G. Consider any H such that $H[j] < w[i, j]$. Suppose if possible H forms the minimum spanning tree. Any spanning tree must have an edge y between the set $fixed(G)$ and *non-fixed*(G). We claim that H cannot be the minimum spanning tree. This claim holds because the edge set $H - \{y\} + \{x\}$ is also a spanning tree and has a lower weight than H. Therefore, unless process j advances to the edge (i, j), G cannot be the minimum spanning tree.

When we advance $G[j]$ to that edge, v_j becomes fixed. Observe that additional nodes may become fixed if their proposed edge lead to v_j directly or indirectly. The algorithm continues to advance G until either all the vertices become fixed or the edge set $E'(G)$ is empty. In the latter case, the graph is not connected and there is no minimum spanning tree.

Algorithm LLP-Prim: Finding the minimum spanning tree in a graph .

```
1 var G: array[1..n − 1] of real initially ∀i : G[i] = minimum edge adjacent to i;
2 always
3       fixed(j, G) ≡ there exists a directed path from v_j to v_0 using edges in G
4       E' := { (i, k) ∈ E | fixed(i, G) ∧ ¬fixed(k, G)};
5 forbidden(j)≡ ∃i : (i, j) ∈ E' such that it has minimum weight w[i, j] of all edges in E'
6 advance(j) G[j] := min{w[i, j] | (i, j) ∈ E'}
```

Algorithm LLP-Prim shows the forbidden and the advance condition.

We now give an efficient implementation of the above LLP-Prim algorithm. The reader may have observed that Prim's algorithm is very similar to Dijkstra's algorithm. In each iteration, we choose the vertex v with the least value of d from all non-fixed vertices. We then explore all the neighbors of v. The only difference is the way d is updated. Many techniques that are applicable to Dijkstra's algorithm are also applicable to Prim's algorithm. For example, similar to Dijkstra's algorithm we can use a heap to maintain d values for all non-fixed vertices and then use the *removeMin()* operation of the heap to find the vertex with the minimum value. Whenever a vertex is removed from the heap, all its outgoing edges can be explored in parallel. On the flip side, Prim's algorithm also suffers from the sequential bottleneck. In every iteration of the *while* loop, exactly one vertex becomes fixed. We now show a parallel algorithm that allows multiple vertices to be fixed in one iteration.

Note that Prim's Algorithm grows the fragment of MST one vertex at a time. We call all the vertices that are part of the fragment as *fixed*. All of these vertices know their final G in the MST rooted at v_0. Prim's algorithm determines the next vertex to be fixed as the vertex that is nearest to fixed vertices. We now claim that any non-fixed vertex such that it is connected to one of the fixed vertices with a minimum weight edge (MWE) can also be fixed. Observe that initially all nodes such that their minimum weight edge points to v_0 are fixed. In every iteration, a node k can become fixed in either of the two ways

1. It is the the nearest neighbor to the fragment. This is the usual way in Prim's algorithm.
2. It is connected to a fixed node z via a minimum weight edge. This edge could be the minimum weight edge for z or for k.

Note that when a node becomes fixed, it may result in additional nodes becoming fixed due to the second way of becoming fixed. We continue to add nodes to the fixed set until we are not able to add any more nodes by this method.

The algorithm starts with the insertion of the source vertex with its d value as 0 in the heap. Instead of removing the minimum vertex from the heap in each iteration and then exploring it, the algorithm consists of two *while* loops. The outer while loop removes one vertex from the heap. If this vertex is fixed, then it has already been explored and therefore it is skipped; otherwise, it is marked as fixed and inserted in R to start the inner while loop. The inner loop keeps processing the set R till it becomes empty.

We do not require that vertices in R be explored in the order of their cost. If R consists of multiple vertices then all of them can be explored in parallel. During this exploration other non-fixed vertices may become fixed. These are then added to R. Some vertices may initially be non-fixed but R may become fixed later when R is processed. To avoid the expense of inserting these vertices in the heap, we collect all such vertices which may need to be inserted or adjusted in the heap in a separate set called Q. Only, when we are done processing R, we call $H.insertOrAdjust$ on vertices in Q.

The vertices $z \in R$ are explored as follows. We process all out-going adjacent edges (z, k) of the vertex z to non-fixed vertices k. This step is called *processEdge1* in Algorithm Parallel-Prim. First, we check if this edge is in the set MWE. If this is the case, then we know that this edge can be added to the tree and the node k can be marked as fixed. Setting $fixed[k]$ to true removes it effectively from the heap because whenever a fixed vertex is extracted in the outer while loop it is skipped. The vertex k can be added to R for future processing. If (z, k) is not in MWE, then we check if the existing distance $d[k]$ is bigger than the weight of the edge (z, k). If this is the case, we update

Algorithm Parallel-Prim: Early Fixing Algorithm $MST1$

```
1  Input: Undirected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  var d: array[0...n-1] of integer initially ∀i : d[i] = ∞;
4      H: binary heap of (j, dist) initially empty;
5      fixed: array[0...n-1] of boolean initially ∀i : fixed[i] = false;
6      Q, R: set of vertices initially empty;
7      T: set of edges initially {};
8      G: array[0...n-1] of integer initially ∀i : G[i] = −1;
9      MWE: set of edges; // minimum weight edges for all vertices;
10 d[0] := 0;
11 H.insert((0, d[0]));
12 while ¬H.empty() do
13     (j, dist) := H.removeMin();
14     if (¬fixed[j]) then
15         R.insert(j);
16         fixed[j] := true;
17         if G[j] ≠ −1 then add (j, G[j]) to T
18         while R ≠ {} do
19             forall z, k : (z ∈ R) ∧ ¬fixed(k) ∧ (z, k) ∈ E in parallel:
20                 processEdge1(z, k);
21         endwhile;
22         forall z ∈ Q : ¬fixed[z] : H.insertOrAdjust (z, d[z])
23 endwhile;
24 if (|T| = n − 1) then return T;
25 else return null; // no spanning tree exists;
26 procedure processEdge1(z, k);
27 if (z, k) ∈ MWE then
28     fixed[k] := true;
29     add (k, z) to T; // z is the G of k
30     R.insert(k);
31 else if (d[k] > w[z, k]) then
32     d[k] := w[z, k];
33     G[k] := z;
34     if (k ∉ Q) then
35         Q.insert(k);
```

$d[k]$ and make z as the G of k. Finally, if $d[k]$ has decreased, we insert it in Q so that once R becomes empty we can call H.insertOrAdjust() method on vertices in Q. This algorithm can be terminated as soon as $n-1$ edges have been chosen.

Let us run this algorithm on the graph in Fig. 10.1. We first insert the vertex a in the heap with distance 0. We remove a from the heap, fix it and insert it into R. We then process all edges of vertices in R. When we process the edge (a, c), we find that it is the mwe for a. Hence, c is fixed and added to R. We continue processing edges of a. The next vertex discovered is b. It is added to Q. We then continue processing R. The next vertex is c. When we process the edge (c, b), we find that it is mwe for b (and c). Hence, b is now added to R. We now remove the vertex b from R and process its edges. Since vertices a and c are already fixed, we discover d and insert it in Q. Now, R is empty and we start inserting vertices from Q into the heap. Since c is already fixed, we insert only the vertex d in the heap. We are now ready to remove the minimum from the heap. The vertex d is removed and fixed. When we explore the edges of d, we find that the edge (d, c) is mwe and the vertex c is also fixed. At this point all vertices have been fixed and we get the edges $\{4, 3, 7, 2\}$ as the MST.

Note that this algorithm requires every vertex to know its minimum weight edge. If this information is not available, then every node can determine this information in parallel by processing all its adjacent edges. In a sequential algorithm, the set MWE can be computed when the graph is input.

To analyze the work complexity of this algorithm when a heap is used, we consider the simple variant in which we simply insert the vertex and its label instead of calling $H.insertOrAdjust$. Just as in Dijkstra's algorithm, any edge is explored at most once in this algorithm. Any such edge exploration results in one heap insertion operation. Thus, we have $O(m \log n)$ complexity due to heap insertions and therefore also heap removals.

10.5 Boruvka's Algorithm: Sequential Implementation

In Prim's algorithm, we started with a trivial fragment including just the vertex v_0. We kept increasing the size of the fragment till it became a spanning tree. In Boruvka's algorithm, we may have more than one fragment. We increase the size of all fragments by adding the minimum outgoing edge for each fragment.

We use T to denote the set of tree edges. Initially, T is empty. When we determine the components in (V, T) using BFS, we get that there are n components as each vertex is a component by itself when T is empty. The algorithm finds minimum weight outgoing edge for each component as follows. At any iteration, we use BFS to find the least numbered vertex that any vertex is connected to in the graph (V, T). This vertex serves as the identifier for the component of the node i, and we use the variable $cid[i]$ to store it. Once we have determined the component identity of all nodes, we move to the next step of determining the minimum weight outgoing edge for each component. We traverse all edges and for each edge that connects two different components we check whether it is cheaper than previously known outgoing edge for the component on either side. Once we have determined all minimum weight edges for every component, we add these to T and start the next iteration.

For example, consider the graph in Fig. 10.1. Initially, T is empty and there are 5 components. We we compute mwe for each component, we get the edges $4, 3, 3, 2, 2$ as the minimum weight edges of a, b, c, d, e, respectively. Once, these edges are added we have two components: $\{a, b, c\}$ and $\{d, e\}$. We then find mwe of these two components as the edge 7. On adding this edge, we have chosen $(n-1)$ edges and the algorithm terminates with the edges $\{2, 3, 4, 7\}$.

After every iteration, the number of connected components in (V, T) reduces by at least a factor of two. This is because every component gets attached to some other component. The worst case is when every least weight edge chosen by any component is also chosen as the least weight edge by the component on the other side of the edge. Hence, the algorithm takes at most $O(\log n)$ iterations of the while loop. It is easy to see that the least weight edge outgoing from each component is found in $O(m)$ work. Thus, the algorithm takes $O(m \log n)$ work.

10.6 Boruvka's Algorithm: Parallel Implementation

An advantage of Boruvka's algroithm is that we get one edge for every component. Thus, if there are c components we get at least $c/2$ new tree edges in one iteration of the algorithm. In the sequential implementation, we used the

Algorithm BoruvkaSeq: Finding MST

```
1  Input: Undirected connected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  var
4      T: { set of edges } initially {};
5      cid: array[1..n] of 0..n initially all 0;
6      mwe: array[1..n] of edge initially all null;
7      dist: array[1..n] of 0..n initially all ∞;
8  while (|T| < n − 1) do
9      visited: array[1..n] of boolean initially all false;
10     for i := 1 to n do
11         if (¬visited[i])
12             // do a BFS in the graph (V, T) from vertex i setting cid of every visited vertex to i
13             BFS(i);
14     for (i, j) ∈ E such that (cid[i] ≠ cid[j]) do
15             if w[i, j] < dist[cid[i]]
16                 dist[cid[i]] = w[i, j]
17                 mwe[cid[i]] = (i, j)
18             if w[i, j] < dist[cid[j]]
19                 dist[cid[j]] = w[i, j]
20                 mwe[cid[j]] = (i, j)
21     forall i do:
22         T := T ∪ mwe[cid[i]];
23 endwhile
24 return T
```

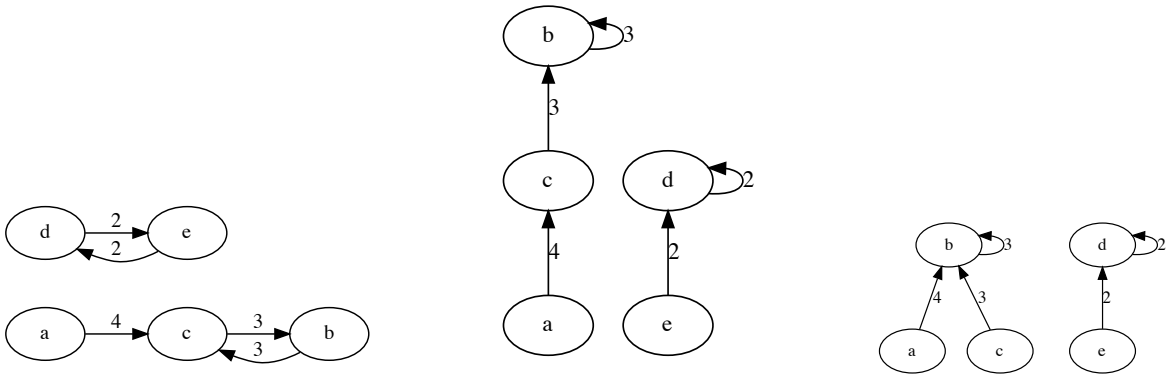

Figure 10.2: (a) Pseudo-tree from the graph in Fig. 10.1 . (b) Rooted-trees (c) Rooted-stars

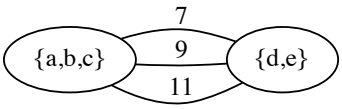

Figure 10.3: Graph after all rooted stars are contracted.

graph BFS traversal to determine the component ids for each node. The BFS traversal is work-efficient but takes as much time as the diameter of the graph. We now show a strategy that takes $O(\log n)$ time to carry out this step. This will give us a parallel algorithm with $O(\log^2 n)$ time complexity.

The heart of the technique is *pointer jumping*. Given any undirected graph (V, E), we first construct a *directed graph* (V, H) in which we every vertex points to the neighbor connected with the least weight edge. In this graph, each component is a **pseudo-tree**, i.e., a tree with a 2-cycle at the root.

We convert the pseudo-tree into a rooted tree and then by using pointer jumping convert the rooted tree to a rooted star. To convert a pseudo-tree into a rooted tree, we simply need to break the 2-cycle into a self-loop and a directed edge. We make the smaller of the nodes as the root with the self-loop and the bigger node points to the smaller node. Fig. 10.2 shows this conversion. The 2-cycle between b and c is converted to c pointing to b and b as the root. To convert a rooted tree to a rooted star, we simply use pointer jumping. Fig. 10.2(c) shows the rooted stars for the graph.

All the vertices in the rooted star are *contracted* into a single vertex in the graph (V, E). At this point, we have a graph in which each vertex represents a fragment of the original graph. We can now recursively find tree edges in the new graph. The algorithm terminates when there is exactly one vertex in the graph.

It is easy to check that the only step in one level of recursive call of Boruvka's algorithm that takes $O(\log n)$ time is the pointer jumping step. Every other step can be done in $O(1)$ time with m processors. Since each level takes $O(\log n)$ time and there are at most $O(\log n)$ levels, we get the parallel time complexity of $O(\log^2 n)$. Since each level takes $O(m)$ work, we get the work complexity of $O(m \log n)$. We remark here that even though we have

Algorithm BoruvkaPar: Finding MST

```
1  Input: Undirected connected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  function Boruvka(V, E)
4  var
5       T: { set of edges } initially {};
6       G: array[1..n] of 0..n initially all 0;
7  if |V| = 1 return {};
8  // Get Pseudo-tree from the graph
9  forall v ∈ V in parallel do
10      G[v] = w such that (v, w) is the minimum weight edge of v;
11      T := T ∪ {(v, w)};
12 // Convert it into a rooted tree
13 forall v ∈ V in parallel do
14      if G[G[v]] = v and v < G[v] then
15      G[v] := v;
16 // Convert every rooted tree into a rooted star
17 while (∃j : G[j] ≠ G[G[j]])
18      forall v ∈ V in parallel do
19          G[v] := G[G[v]]
20 // Contract all rooted stars into a single vertex
21 V' := {v ∈ V | G[v] = v}
22 E' := {(G[v], G[w])|((v, w) ∈ E) ∧ (G[v] ≠ G[w])}
23 return T ∪ Boruvka(V', E')
```

given the algorithm in a recursive fashion, it is easy to convert it into an iterative algorithm. We leave that as an exercise (Problem 3).

10.7 LLP-Boruvka Algorithm

In this section, we continue with the recursive version of the algorithm, but implement each recursive instance of Boruvka's algorithm with the LLP algorithm. As before, we assume that input to our algorithm is a connected graph without any self-loops. The LLP algorithm uses the single variable G. For each instance, we let the minimum weight edge (mwe) adjacent to each vertex v be denoted by $mwe[v]$. We denote this directed graph by H: the set of vertices is V and the set of edges is $\{(v,w)|(v,w) = mwe[v]\}$. We initialize $G[v]$ with the node w when $mwe[v] = (v,w)$ except when $mwe[w] = (w,v)$ and $v < w$. For the latter case, we make $G[v]$ equal to v. As a result of this initialization, the structure $G[v]$ forms a rooted tree. This initialization is simply the steps of getting the rooted tree in Algorithm BoruvkaPar. We say that the edge (v,w) is *incident* to v when $G[v]$ equals w for w different from v.

Algorithm LLP-Boruvka: Finding MST

```
1  Input: Undirected connected Weighted Graph: (V, E, w).
2  Output: Minimum Weight Spanning Tree
3  function Boruvka(V, E)
4  var
5       G: array[1..n] of 1..n initially G[v] equals w such that mwe[v] = (v, w) and mwe[w] ≠ (v, w),
     or mwe[v] = mwe[w] = (v, w) and v < w
6       v, otherwise.
7  forbidden(j)≡ G[j] ≠ G[[G[j]]
8  advance(j) ≡ G[j] := G[G[j]]
9  if |V| = 1 return {};
10 else return T ∪ Boruvka(V', E') where
11      T := {(v, w) | (v, w) is the minimum weight edge of v};
12      V' := {v ∈ V | G[v] = v}
13      E' := {(G[v], G[w])|((v, w) ∈ E) ∧ (G[v] ≠ G[w])}
```

Once we have rooted trees for the graph, we convert the rooted trees to rooted stars using pointer-jumping. A node j is considered forbidden if $G[j] \neq G[G[j]]$. It is advanced by setting $G[j]$ to $G[G[j]]$. The algorithm stops this iteration when no node is forbidden. We show that each instance is indeed an LLP algorithm.

We first show a basic lemma.

Lemma 10.2 *The following is an invariant of the program: $G[v]$ is reachable from v in the directed graph H.*

Proof: The invariant is true initially because $G[v]$ is set to w where (v,w) is an edge in H. The only statement executed in the program is $G[v] := G[G[v]]$ which preserves the invariant because $G[G[v]]$ is reachable from $G[v]$.

■

We also observe that H is a collection of rooted trees and if w is reachable from v, then there is a unique path from v to w. Furthermore, the weight of edges along any path is strictly decreasing. This is because if v points to w and w points to u then the weight of the edge (w,u) must be strictly smaller than the the weight of the edge (v,w) by the property that each vertex points to the minimum weight edge incident to it. We are now ready to show the following lemma.

Lemma 10.3 *Each instance of finding connected components is an LLP algorithm that terminates.*

Proof: We consider the lattice of vectors. For component j, we keep the weight of the minimum weight edge in the path from j to $G[j]$. We define the predicate B as

$$B \equiv \forall j : G[j] = G[G[j]]$$

We show that predicate is lattice-linear. Suppose the predicate B is not true. This implies that there exists j such that $G[j]$ is different from $G[G[j]]$. We show that the component j is forbidden. $G[j]$ cannot be the root of a tree in H because the root points to itself (and therefore $G[j]$ equals $G[G[j]]$. Let k be equal to $G[j]$. We just showed that k is not equal to the root. Therefore, $G[k] \neq k$ even if $G[k]$ changes. Hence, $G[j] \neq G[G[j]]$ continues to hold. The component j in G vector is advanced by setting $G[j]$ to be $G[G[j]]$.

We now show that the LLP algorithm terminates. When $G[j]$ is advanced, the weight of the last edge from j to $G[j]$ can only strictly decrease. Since there are at most $n-1$ edges in H, the algorithm must eventually terminate with B as true.

■

Algorithm Figure LLP-Boruvka shows the entire algorithm. Observe that there are no synchronization requirements for any instance of the LLP algorithm. Any thread j that finds $G[j]$ different from $G[G[j]]$ can set it to $G[G[j]]$. In particular, $G[G[j]]$ may have changed between the observation of $(G[j] \neq G[G[j]])$ and setting of $G[j]$ to $G[G[j]]$, but the algorithm still works correctly.

10.8 Summary

The following table lists all the algorithms discussed in this chapter.

Problem	**Algorithm**	**Parallel Time**	**Work**
Minimum Spanning Tree	Kruskal's	$O(m)$	$O(m \log n)$
Minimum Spanning Tree	LLP-Kruskal1	$O(m)$	$O(m^2)$
Minimum Spanning Tree	LLP-Kruskal2	$O(\log^2 n)$	$O(n^3 \log n)$
Minimum Spanning Tree	Prim	$O(m)$	$O(m \log n)$
Minimum Spanning Tree	LLP-Prim	$O(m)$	$O(m \log n)$
Minimum Spanning Tree	Boruvka-Par	$O(\log^2 n)$	$O(m \log n)$
Minimum Spanning Tree	LLP-Boruvka	$O(\log^2 n)$	$O(m \log n)$

10.9 Problems

1. Show that there is a unique minimum spanning tree in any weighted undirected graph when all edge weights are distinct.
2. Suppose that you are given an undirected graph (V, E, w) such that all edge weights are distinct. You are also given a subset of edges E' such that (V, E') is acyclic. Show that there exists a unique minimum spanning tree T that is constrained to include all edges in E'.
3. Give a parallel iterative implementation of Boruvka's algorithm.

10.10 Bibliographic Remarks

Sequential algorithms for minimum spanning tree can be found in [CLRS01]. Parallel Boruvka algorithm is available in [SMDD19].