Chapter 14

Minimum Spanning Tree

In this chapter we will cover another important graph problem, Minimum Spanning Trees (MST). The MST of an undirected weighted graph is a tree that spans the graph and for which the sum of the edge weights is no more than any other such tree. We will first cover what it means to be a spanning tree, and an important cut property on graphs. We then cover three different algorithms for the problem: Kruskal's, Prim's, and Borůvka's. All of them make use of the cut property. The first two are sequential, and the third is parallel.

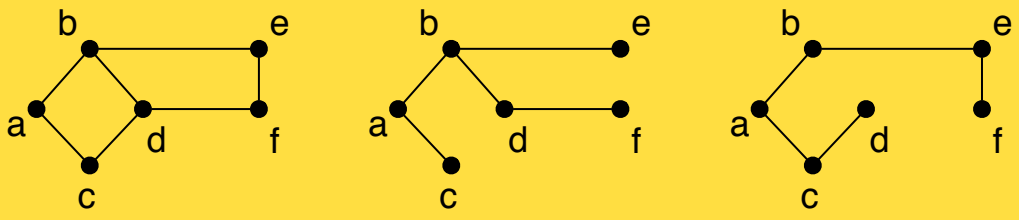
14.1 Spanning Trees

Recall that we say that an undirected graph is a forest if it has no cycles and a tree if it is also connected. Often in a general connected undirected graph we want to identify a subset of the edge that form a tree.

Definition 14.1. *For a connected undirected graph $G = (V, E)$, a* spanning tree *is a tree $T = (V, E')$ with $E' \subseteq E$.*

Note that a spanning tree of a graph G is a subgraph of G that *spans* the graph (includes all its vertices). A graph can have many spanning trees, but all have $|V|$ vertices and $|V| - 1$ edges.

Example 14.2. *A graph on the left, and two possible spanning trees.*



Exercise 14.3. *Prove that any tree with n vertices has $n - 1$ edges.*

One way to generate a spanning tree is simply to do a graph search, such as DFS or BFS. Whenever we visit a new vertex we add a new edge to it, as in DFS and BFS trees. DFS and BFS are work-efficient algorithms for computing spanning trees but they are not good parallel algorithms. Another way to generate a spanning tree is to use graph contraction, which as we have seen can be done in parallel. The idea is to use star contraction and add all the edges that are picked to define the stars throughout the algorithm to the spanning tree.

Exercise 14.4. *Work out the details of the algorithm for spanning trees using graph contraction and prove that it produces a spanning tree.*

14.2 Minimum Spanning Trees

Recall that a graph has many spanning trees. When the graphs are weighted, we are usually interested in finding the spanning tree with the smallest total weight (i.e. sum of the weights of its edges).

Definition 14.5. *The minimum (weight) spanning tree (MST) problem is given an connected undirected weighted graph $G = (V, E, w)$, find a spanning tree of minimum weight, where the weight of a tree T is defined as:*

$$w(T) = \sum_{e \in E(T)} w(e) \, .$$

Minimum spanning trees have many interesting applications.

Example 14.6. *Suppose that you are wiring a building so that all the rooms are connected. You can connect any two rooms at the cost of the wire connecting them. To minimize the cost of the wiring, you would find a minimum spanning tree of the graph representing the building.*

Bounding TSP with MST. There is an interesting connection between minimum spanning trees and the symmetric Traveling Salesperson Problem (TSP), an NP-hard problem. Recall that in TSP problem, we are given a set of n cities (vertices) and are interested in finding a tour that visits all the vertices exactly once and returns to the origin. For the symmetric case the edges are undirected (or equivalently the distance is the same in each direction). For the TSP problem, we

usually consider complete graphs, where there is an edge between any two vertices. Even if a graph is not complete, we can typically complete it by inserting edges with large weights that make sure that the edge never appears in a solution. Here we also assume the edge weights are non-negative.

Since the solution to the TSP problem visits every vertex once (returning to the origin), it spans the graph. It is however not a tree but a cycle. Since each vertex is visited once, however, dropping any edge would yield a spanning tree. Thus a solution to the TSP problem cannot have less total weight than that of a minimum spanning tree. In other words, the weight of a MST yields a lower bound on the solution to the symmetric TSP problem for graphs with non-negative edge weights.

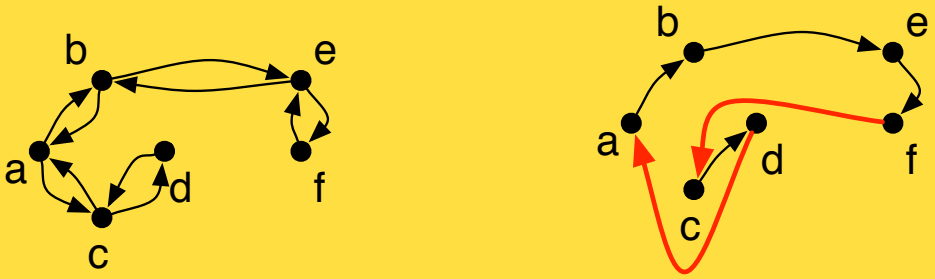
Approximating TSP with MST. It turns out that minimum spanning trees can also be used to find an approximate solutions to the TSP problem, effectively finding an upper bound. This, however, requires one more condition on the MST problem. In particular in addition to requiring that weights are non-negative we require that all distances satisfy the triangle inequality—i.e., for any three vertices a, b, and c, $w(a, c) \leq w(a, b) + w(b, c)$. This restriction holds for most applications of the TSP problem and is referred to as the *metric TSP* problem. It also implies that edge weights are non-negative. We would now like a way to use the MST to generate a path to take as an approximate solution to the TSP problem. To do this we first consider a path based on the MST that can visit a vertex multiple times, and then take shortcuts to ensure we only visit each vertex once.

Given a minimum spanning tree T we can start at any vertex s and take a path based on the depth-first search on the tree from s. In particular whenever we visit a new vertex v from vertex u we traverse the edge from u to v and when we are done visiting everything reachable from v we then back up on this same edge, traversing it from v to u. This way every edge in our path is traversed exactly twice, and we end the path at our initial vertex. If we view each undirected edge as two directed edges, then this path is a so-called *Euler tour* of the tree—i.e. a cycle in a graph that visits every edge exactly once. Since T spans the graph, the Euler tour will visit every vertex at least once, but possibly multiple times.

Example 14.7. *The figure on the right shows an Euler tour of the tree on the left. Starting at* `a`*, the tour visits* `a`, `b`, `e`, `f`, `e`, `b`, `a`, `c`, `d`, `c`, `a`.

Now, recall that in the TSP problem it is assumed that there is an edge between every pair of vertices. Since it is possible to take an edge from any vertex to any other, we can take shortcuts to avoid visiting vertices multiple times. More precisely what we can do is when about to go back to a vertex that the tour has already visited, instead find the next vertex in the tour that has not been visited and go directly to it. We call this a shortcut edge.

Example 14.8. *The figure on the right shows a solution to TSP with shortcuts, drawn in red. Starting at* `a`*, we can visit* `a`, `b`, `e`, `f`, `c`, `d`, `a`.



By the triangle inequality the shortcut edges are no longer than the paths that they replace. Thus by taking shortcuts, the total distance is not increased. Since the Euler tour traverses each edge in the minimum spanning tree twice (once in each direction), the total weight of the path is exactly twice the weight of the TSP. With shortcuts, we obtain a solution to the TSP problem that is at most the weight of the Euler tour, and hence at most twice the weight of the MST. Since the weight of the MST is also a lower bound on the TSP, the solution we have found is within a factor of 2 of optimal. This means our approach is an approximation algorithm for TSP that approximates the solution within a factor of 2. This can be summarized as:

$$W(\text{MST}(G)) \leq W(\text{TSP}(G)) \leq 2W(\text{MST}(G)) \ .$$

Remark 14.9. *It is possible to reduce the approximation factor to 1.5 using a well known algorithm developed by Nicos Christofides at CMU in 1976. The algorithm is also based on the MST problem, but is followed by finding a vertex matching on the vertices in the MST with odd-degree, adding these to the tree, finding an Euler tour of the combined graph, and again shortcutting. Christofides algorithm was one of the first approximation algorithms and it took over 40 years to improve on the result, and only very slightly.*

14.3 Algorithms for Minimum Spanning Trees

There are several algorithms for computing minimum spanning trees. They all, however, are based on the same underlying property about cuts in a graph, which we will refer to as the

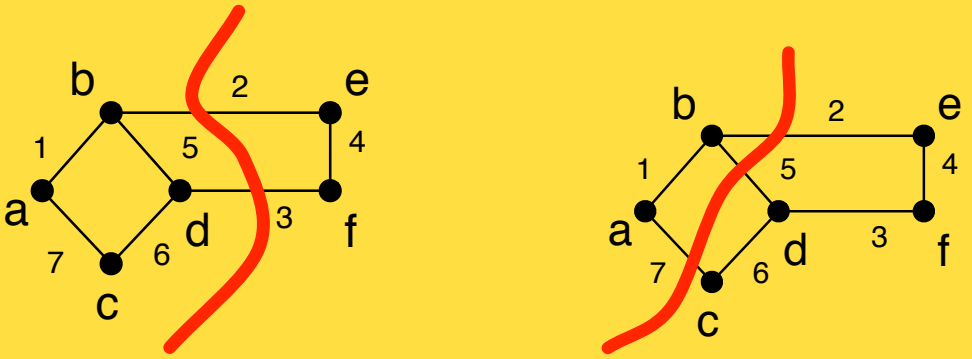
light-edge property. Roughly, the light-edge property states that if you partition the graph into two, the minimum edge between the two parts has to be in the MST. A more formal definition is given below. The light-edge property gives a way to identify edges of the MST, which can then be repeatedly added to form the tree. In our discussion we will assume without any loss of generality that all edges have distinct weights. It is without loss-of-generality since we are free to break ties in a consistent way (e.g. if two edges have the same weight, order them by where they appear in the input). Given distinct weights, the minimum spanning tree of any graph is unique.

Exercise 14.10. *Prove that a graph with distinct edge weights has a unique minimum spanning tree.*

Definition 14.11. *For a graph $G = (V, E)$, a* cut *is defined in terms of a non-empty proper subset $U \subsetneq V$. This set U partitions the graph into $(U, V \setminus U)$, and we refer to the edges between the two parts as the cut edges $E(U, \overline{U})$, where as is typical in literature, we write $\overline{U} = V \setminus U$.*

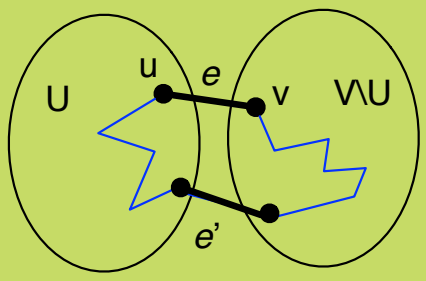
The subset U used in the definition of a cut might include a single vertex v, in which case the cut edges would be all edges incident on v. But the subset U must be a proper subset of V (i.e., $U \neq \emptyset$ and $U \neq V$). We sometimes say that a cut edge *crosses* the cut.

Example 14.12. *Two example cuts. For each cut, we can find the lightest edge that crosses that cut, which in this case is 2 for both cuts shown.*



We are now ready for a formal statement and proof of the light-edge property, which is given in Lemma 14.13. As indicated in the implication, all three algorithms we consider take advantage of the light-edge property. Kruskal's and Prim's algorithms are based on selecting a single lightest weight edge on each step and are hence sequential, while Borůvka's selects multiple edges and hence can be parallelized. We briefly review Kruskal's and Prim's algorithm and will spend most of our time on a parallel variant of Borůvka's algorithm.

Lemma 14.13 (Light-Edge Property)**.** *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any cut of G, the minimum weight edge that crosses the cut is in the minimum spanning tree MST(G) of G.*



Proof. The proof is by contradiction. Assume the minimum-weighted edge $e = (u, v)$ is not in the MST. Since the MST spans the graph, there must be some simple path P connecting u and v in the MST (i.e., consisting of just edges in the MST). The path must cross the cut between U and $V \setminus U$ at least once since u and v are on opposite sides. Let e' be an edge in P that crosses the cut. By assumption the weight of e' is larger than that of e. Now, insert e into the graph—this gives us a cycle that includes both e and e'—and remove e' from the graph to break the only cycle and obtain a spanning tree again. Now, since the weight of e is less than that of e', the resulting spanning tree has a smaller weight. This is a contradiction and thus e must have been in the tree. □

Implication: Any edge that is a minimum weight edge crossing a cut can be immediately added to the MST. For example the overall minimum edge (Kruskal's algorithm), the minimum edge incident on each vertex (Borůvka's algorithm), or when doing a graph search, the minimum edge between the visited set and the frontier (Prim's algorithm).

Remark 14.14. *Even though Borůvka's algorithm is the only parallel algorithm, it was the earliest, invented in 1926, as a method for constructing an efficient electricity network in Moravia in the Czech Republic. It was re-invented many times over.*

Kruskal's Algorithm

As described in Kruskal's original paper, the algorithm is:

> "Perform the following step as many times as possible: Among the edges of G not yet chosen, choose the shortest edge which does not form any loops with those edges already chosen" [Kruskal, 1956]

In more modern terminology we would replace "shortest" with "lightest" and "loops" with "cycles".

Kruskal's algorithm is correct since it maintains the invariant on each step that the edges chosen so far are in the MST of G. This is true at the start. Now on each step, any edge that forms a cycle with the already chosen edges cannot be in the MST. This is because adding it would would violate the tree property of an MST and we know, by the invariant, that all the other edges on the cycle are in the MST. Now considering the edges that do not form a cycle, the minimum weight edge must be a "light edge" since it is the least weight edge that connects the connected subgraph at either endpoint to the rest of the graph. Finally we have to argue that all the MST edges have been added. Well we considered all edges, and only tossed the ones that we could prove were not in the MST (i.e. formed cycles with MST edges).

We could finish our discussion of Kruskal's algorithm here, but a few words on how to implement the idea efficiently are warranted. In particular checking if an edge forms a cycle might be expensive if we are not careful. Indeed it was not until many years after Kruskal's original paper that an efficient approach to the algorithm was developed. Note that to check if an edge (u, v) forms a cycle, all one needs to do is test if u and v are in the same connected component as defined by the edges already chosen. One way to do this is by contracting an edge (u, v) whenever it is added—i.e., collapse the edge and the vertices u and v into a single supervertex. However, if we implement this as described in the last chapter we would need to update all the other edges incident on u and v. This can be expensive since an edge might need to be updated many times.

To get around these problem it is possible to update the edges lazily. What we mean by lazily is that edges incident on a contracted vertex are not updated immediately, but rather later when the edge is processed. At that point the edge needs to determine what supervertices (components) its endpoints are in. This idea can be implemented with a so-called union-find data type. The ADT supports the following operations on a union-find type U: $\texttt{insert}(U, v)$ inserts the vertex v, $\texttt{union}(U, (u, v))$ joins the two elements u and v into a single supervertex, $\texttt{find}(U, v)$ returns the supervertex in which v belongs, possibly itself, and $\texttt{equals}(u, v)$ returns true if u and v are the same supervertex. Now we can simply process the edges in increasing order. This idea gives Algorithm 14.15.

To analyze the work and span of the algorithm we first note that there is no parallelism, so the span equals the work. To analyze the work we can partition it into the work required for sorting the edges and then the work required to iterate over the edges using union and find. The sort requires $O(m \log n)$ work. The union and find operations can be implemented in $O(\log n)$ work each requiring another $O(m \log n)$ work since they are called $O(m)$ times. The overall work is therefore $O(m \log n)$. It turns out that the union and find operations can actually be implemented with less than $O(\log n)$ amortized work, but this does not reduce the overall work since we still have to sort.

Prim's Algorithm

Prim's algorithm performs a priority-first search to construct the minimum spanning tree. The idea is that if we have already visited a set X, then by the light-edge property the minimum

Algorithm 14.15 (Union-Find Kruskal)**.**

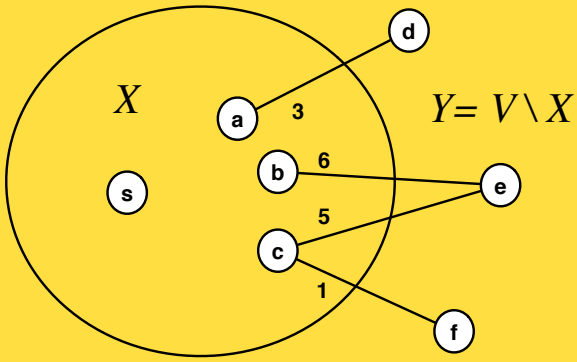
```
1  function kruskal(G = (V, E, w)) =
2  let
3     val U = iter UF.insert UF.∅ V    % insert vertices into union find structure
4     val E' = sort(E, w)              % sort the edges
5     function addEdge((U, T), e = (u, v)) =
6     let
7        val u' = UF.find(U, u)
8        val v' = UF.find(U, v)
9     in
10       if (u' = v') then (U, T)              % if u and v are already connected then skip
11       else (UF.union(U, u', v'), T ∪ e)     % contract edge e in U and add e to T
12    end
13 in
14    iter addEdge (U, ∅) E'
15 end
```

weight edge with one of its endpoint in X and the other in $V \setminus X$ must be in the MST (it is a minimum cross edge from X to $V \setminus X$). We can therefore add it to the MST and include the other endpoint in X. This leads to the following definition of Prim's algorithm:

Algorithm 14.16 (Prim's Algorithm)**.** *For a weighted undirected graph $G = (V, E, w)$ and a source s, Prim's algorithm is priority-first search on G starting at an arbitrary $s \in V$ with $T = \emptyset$, using priority $p(v) = \min_{x \in X} w(x, v)$ (to be minimized), and setting $T = T \cup \{(u, v)\}$ when visiting v where $w(u, v) = p(v)$.*

When the algorithm terminates, T is the set of edges in the MST.

Example 14.17. *A step of Prim's algorithm. Since the edge (c, f) has minimum weight across the cut (X, Y), the algorithm will "visit" f adding (c, f) to T and f to X.*



Remark 14.18. *This algorithm was invented in 1930 by Czech mathematician Vojtech Jarnik and later independently in 1957 by computer scientist Robert Prim. Edsger Dijkstra's rediscovered it in 1959 in the same paper he described his famous shortest path algorithm.*

Exercise 14.19. *Carefully prove the correctness of Prim's algorithm by induction.*

Interestingly this algorithm is quite similar to Dijkstra's algorithm for shortest paths. The only differences are (1) we start at an arbitrary vertex instead of at a source, (2) that $p(v) = \min_{x \in X}(x, v)$ instead of $\min_{x \in X}(d(x) + w(x, v))$, and (3) we maintain a tree T instead of a table of distances $d(v)$. Because of the similarity we can basically use the same priority-queue implementation as in Dijkstra's algorithm and it runs with the same $O(m \log n)$ work bounds.

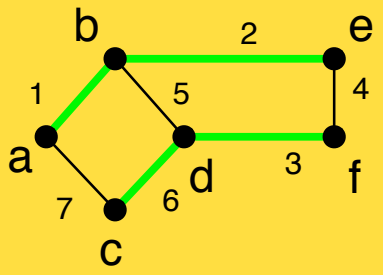
Exercise 14.20. *Write out the pseudocode for a Priority Queue based implementation of Prim's algorithm that runs in $O(m \log n)$ work.*

14.4 Parallel Minimum Spanning Tree

As we discussed, Kruskal and Prim's algorithm are sequential algorithms. We now focus on developing an MST algorithm that runs efficiently in parallel using graph contraction.

We will be considering at a parallel algorithm based on an approach by Borůvka, which we may even be able to invent on the fly. The two algorithms so far picked light edges belonging to the MST carefully one by one. It is in fact possible to select many light edges at the same time. Recall than all light edges that cross a cut must be in the MST. The most trivial cut is simply to consider a vertex v and the rest of the vertices in the graph. The edges that cross the cut are the edges incident on v. Therefore, by the light edge rule, for each vertex, the minimum weight edge between it and its neighbors is in the MST. We will refer to these edges as the *minimum-weight edges* of the graph.

Example 14.21. *The minimum-weight edges of the graph are highlighted. The vertices* `a` *and* `b` *both pick edge* $\{a, b\}$*,* `c` *picks* $\{c, d\}$*,* `d` *and* `f` *pick* $\{d, f\}$*, and* `e` *picks* $\{e, b\}$*.*

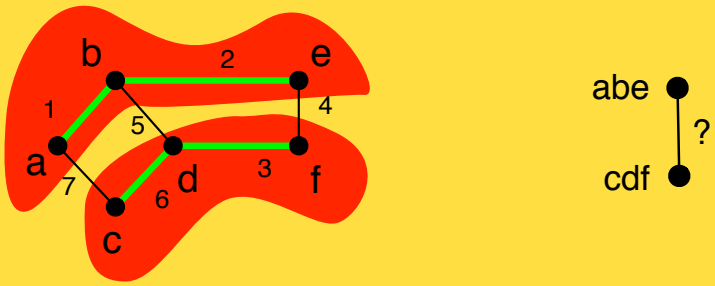


This gives us a way to identify many MST edges, and since each edge can probably find its own minimum edge, it is likely this can be done in parallel. Sometimes just one round of picking minimum-weight edges will select all the MST edges and thus would complete the algorithm. However, in most cases, the minimum-weight edges on their own do not form a spanning tree. Indeed, in our example, we are missing the edge $(\texttt{e}, \texttt{f})$ since neither $\texttt{e}$ nor $\texttt{f}$ pick it. Note that we no longer have to consider edges that are within a component since adding any such edge would create a cycle with edges that we know are in the MST, and therefore cannot be in the MST. Therefore if we can contract the graph along the edges that we selected, we can proceed to consider the cuts that have not been covered.

Borůvka's Algorithm

The idea of Borůvka's algorithm is to use graph contraction to collapse each component that is connected by a set of minimum-weight edges into a single vertex. Recall that in graph contraction, all we need is a partition of the graph into disjoint connected subgraphs. Given such a partition, we then replace each subgraph (partition) with a supervertex and relabel the edges. This is repeated until no edges remain.

Example 14.22. *Contraction along the minimum edges. Note that there are redundant edges between the two partitions.*



One property of graph contraction is that it can create redundant edges. When discussing graph contraction in the last chapter the graphs were unweighted so we could just keep one of the redundant edges, and it did not matter which one. When the edges have weights, however, we have to decide what the "combined" weight will be. How the edges are combined will depend on the application, but in the application to MST in Borůvka's algorithm we note that any future cut will always cut all the edges or none of them. Since we are always interested in finding minimum weight edges across a cut, we only need to keep the minimum of the redundant edges. In the example above we would keep the edge with weight 4.

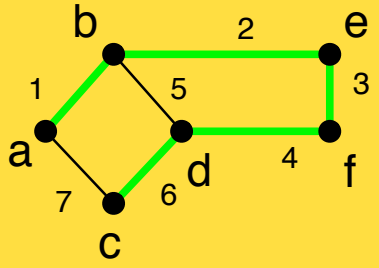
What we just covered is exactly Borůvka's idea. He did not discuss implementing the contraction in parallel. At the time, there were not any computers let alone parallel ones. We are glad that he has left us something to do. In summary, Borůvka's algorithm can be described as follows.

Algorithm 14.23 (Borůvka)**.** *While there are edges remaining: (1) select the minimum weight edge out of each vertex and contract each connected component defined by these edges into a vertex; (2) remove self edges, and when there are redundant edges keep the minimum weight edge; and (3) add all selected edges to the MST.*

We now consider the efficiency of this algorithm. We first focus on the number of rounds of contraction and then consider how to implement the contraction. We note that since contracting an edge removes exactly one vertex, if k edges are selected then k vertices are removed. We might now be tempted to say that every vertex will be removed since every vertex selects an edge. This is not the case since two vertices can select the same edge. Therefore there can be half as many edges as vertices, but there must be at least half as many. We therefore remove half the vertices on each round. This implies that Borůvka's algorithm will take at most $\log_2 n$ rounds.

We now consider how to contract the graph on each round. This requires first identifying the minimum-weight edges. How this is done depends on the graph representation, so we will defer this for the moment, and look at the contraction itself once the minimum-weight edges have been identified. In general the components identified by selecting minimum-weight edges are neither single edges nor single stars.

Example 14.24. *An example where minimum-weight edges give a non-star tree. Note that we have in fact picked a minimum spanning tree by just selecting the minimum-weight edges.*



In general, the minimum-weight edges will form a forest (a set of trees).

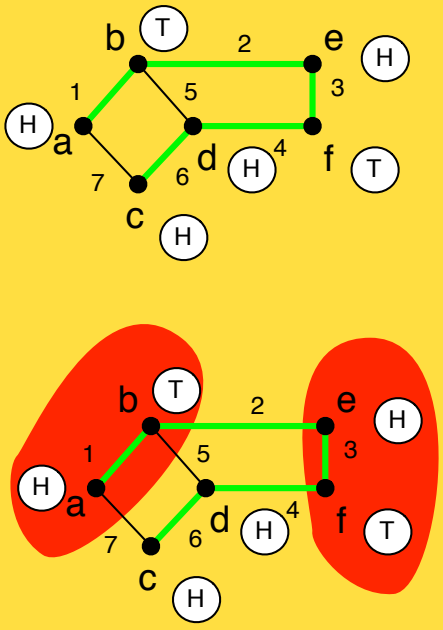
Exercise 14.25. *Prove that the minimum-weight edges will indeed form a forest. Recall that we are assuming that no two edge weights are equal.*

Therefore what we want to contract are trees. A tree can be contracted by repeatedly contracting disjoint stars within the tree. Indeed this can be done with `contractGraph` using star contraction from the last chapter. Furthermore since when doing a star contraction on a forest, it remains a forest on each step, the number of edges goes down with the number of vertices. Therefore the total work to contract all the stars will be bounded by $O(n)$ if using array sequences. The span remains $O(\log^2 n)$.

After contracting each tree, we have to update the edges. As discussed earlier for redundant edges we want to keep the minimum weight such edge. There are various ways to do this, including keeping the redundant edges. Keeping the edges turns out to be an effective solution, and allows the updating the edges to be done in $O(m)$ work. Assuming redundant edges, the minimum into each component can still be done with $O(m)$ work, as described below. Since there are at most $\log n$ rounds, Borůvka's algorithm will run in $O(m \log n)$ work and $O(\log^3 n)$ span.

Borůvka's algorithm, improved with star contraction. We will now improve the span of Borůvka by a logarithmic factor by interleaving steps of star contraction with steps of finding the minimum-weight edges, instead of fully contracting the trees defined by the minimum-weight edges. The idea is to apply randomized star contraction on the minimum-weight edges, and then select a new set of minimum-weight edges. We repeat this simpler contraction step until there are no edges. As we will show, at each step, we will still be able to reduce the number of vertices by a constant factor (in expectation), leading to logarithmic number of rounds. The advantage of this second approach is that we will reduce the overall span for finding the MST from $O(\log^3 n)$ to $O(\log^2 n)$ while maintaining the same work.

Example 14.26. *An example of Borůvka with star contraction.*



More precisely, for a set of minimum-weight edges $minE$, let $H = (V, minE)$ be a subgraph of G. We will apply one step of the star contraction algorithm on H. To do this we modify our `starContract` routine so that after flipping coins, the tails only hook across their minimum-weight edge. The modified algorithm for star contraction is as follows. In the code w stands for the weight of the edge (u, v).

Pseudo Code 14.27 (Star Contraction along Minimum-Weight Edges).

```
1  function minStarContract(G = (V, E), i) =
2  let
3      val minE = minEdges(G)
4      val P = {u ↦ (v, w) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
5      val V' = V \ domain(P)
6  in (V', P) end
```

where $\texttt{minEdges}(G)$ *finds the minimum edge out of each vertex* v.

Before we go into details about how we might keep track of the MST and other information, let us try to understand what effects this change has on the number of vertices contracted away. If we have n non-isolated vertices, the following lemma shows that the algorithm still removes $n/4$ vertices in expectation on each step:

Lemma 14.28. *For a graph* G *with* n *non-isolated vertices, let* X_n *be the random variable indicating the number of vertices removed by* $\texttt{minStarContract}(G, r)$. *Then,* $\mathbf{E}[X_n] \geq n/4$.

Proof. The proof is pretty much identical to our proof for `starContract` except here we're not working with the whole edge set, only a restricted one $\texttt{minE}$. Let $v \in V(G)$ be a non-isolated vertex. Like before, let H_v be the event that v comes up heads, T_v that it comes up tails, and R_v that $v \in \texttt{domain}(P)$ (i.e, it is removed). Since v is a non-isolated vertex, v has neighbors—and one of them has the minimum weight, so there exists a vertex u such that $(v, u) \in \texttt{minE}$. Then, we have that $T_v \wedge H_u$ implies R_v since if v is a tail and u is a head, then v must join u. Therefore, $\mathbf{Pr}[R_v] \geq \mathbf{Pr}[T_v]\,\mathbf{Pr}[H_u] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

$$\mathbf{E}\left[\sum_{v:v \text{ non-isolated}} \mathbb{I}\{R_v\}\right] = \sum_{v:v \text{ non-isolated}} \mathbf{E}[\mathbb{I}\{R_v\}] \geq n/4$$

since we have n vertices that are non-isolated. □

This means that this MST algorithm will take only $O(\log n)$ rounds, just like our other graph contraction algorithms.

Final Things. There is a little bit of trickiness since, as the graph contracts, the endpoints of each edge changes. Therefore, if we want to return the edges of the minimum spanning tree, they might not correspond to the original endpoints. To deal with this, we associate a unique label with every edge and return the tree as a set of labels (i.e. the labels of the edges in the spanning tree). We also associate the weight directly with the edge. The type of each edge is therefore $(\texttt{vertex} \times \texttt{vertex} \times \texttt{weight} \times \texttt{label})$, where the two vertex endpoints can change as the

Algorithm 14.29 (Borůvka's based on Star Contraction)**.**

```
1  function minEdges (E) =
2  let
3     val ET = {(u,v,w,l) ↦ {u ↦ (v,w,l)} : (u,v,w,l) ∈ E}
4     function joinEdges((v1,w1,l1),(v2,w2,l2)) =
5        if (w1 ≤ w2) then (v1,w1,l1) else (v2,w2,l2)
6  in
7     reduce (merge joinEdges) {} ET
8  end

9  function minStarContract(G = (V,E), i)
10 let
11    val minE = minEdges(G)
12    val P = {(u ↦ (v,w,ℓ)) ∈ minE | ¬heads(u,i) ∧ heads(v,i)}
13    val V' = V \ domain(P)
14 in (V',P) end

15 function MST((V,E), T, i) =
16 if (|E| = 0) then T
17 else let
18    val (V',PT) = minStarContract((V,E),i)
19    val P = {u ↦ v : u ↦ (v,w,ℓ) ∈ PT} ∪ {v ↦ v : v ∈ V'}
20    val T' = {ℓ : u ↦ (v,w,ℓ) ∈ PT}
21    val E' = {(P[u],P[v],w,l) : (u,v,w,l) ∈ E | P[u] ≠ P[v]}
22 in
23    MST((V',E'), T ∪ T', i+1)
24 end
```

graph contracts but the weight and label stays fixed. This leads to the slightly-updated version of `minStarContract` that appears in Algorithm 14.29.

The function `minEdges`(G) in Line 11 finds the minimum edge out of each vertex v and maps v to the pair consisting of the neighbor along the edge and the edge label. By Theorem 14.13, since all these edges are minimum out of the vertex, they are safe to add to the MST. Line 12 then picks from these edges the edges that go from a tail to a head, and therefore generates a mapping from tails to heads along minimum edges, creating stars. Finally, Line 13 removes all vertices that are in this mapping to star centers.

This is ready to be used in the MST code, similar to the `graphContract` code studied last time, except we return the set of labels for the MST edges instead of the remaining vertices. The code is given in Algorithm 14.29 The MST algorithm is called by running `MST`$(G, \emptyset, r)$. As an aside, we know that T is a spanning forest on the contracted nodes.

Finally we describe how to implement `minEdges`(G), which returns for each vertex the

minimum edge incident on that vertex. There are various ways to do this. One way is to make a singleton table for each edge and then merge all the tables with an appropriate function to resolve collisions. Algorithm 14.29 gives code that merges edges by taking the one with lighter edge weight.

If using sequences for the edges and vertices an even simpler way is to presort the edges by decreasing weight and then use `inject`. Recall that when there are collisions at the same location `inject` will always take the last value, which will be the one with minimum weight.

.