Chapter 15

Minimum Spanning Tree

In this chapter we will cover another important graph problem, Minimum Spanning Trees (MST). The goal is for a weighted connected graph to find 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.

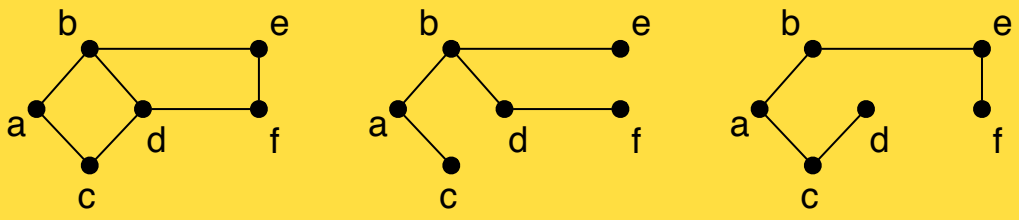
15.1 Spanning Trees

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 15.1. *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 15.2. *A graph on the left, and two possible spanning trees.*



Exercise 15.3. *Prove that any tree with n vertices has $n - 1$ edges.*

Question 15.4. *Can you think of an algorithm for finding a spanning tree of a connected undirected graph?*

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 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.

Question 15.5. *Can you think of an algorithm with polylogarithmic span for finding a spanning tree of a connected undirected graph?*

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

15.2 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 15.7. *The minimum (weight) spanning tree (MST) problem is given an connected undirected weighted graph $G = (V, E, w)$ with non-negative weights, 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 15.8. *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 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.

Question 15.9. *Can you think of a way to bound the solution to a TSP problem on an undirected connected graph using minimum spanning trees.*

Since the solution to TSP 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, MST yields a lower bound on symmetric TSP.

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 we require that all distances satisfy the triangle inequality. In particular, for any three vertices a, b, and c the triangle inequality requires that $w(a, c) \leq w(a, b) + w(b, c)$. This restriction makes sense and is referred to as the *metric TSP* problem. We would now like a way to use the MST to generate a path to take for the TSP. Let us first consider paths that are allowed to visit a vertex more than once.

Question 15.10. *Given an undirected graph G, suppose that you compute a minimum spanning tree T. Can you use the tree to visit each vertex in the graph from a given origin?*

Since a minimum spanning tree T spans the graph, we can start at the origin and just do a DFS, treating the undirected edges as edges in both directions. This way we will be able to visit each vertex in the graph twice.

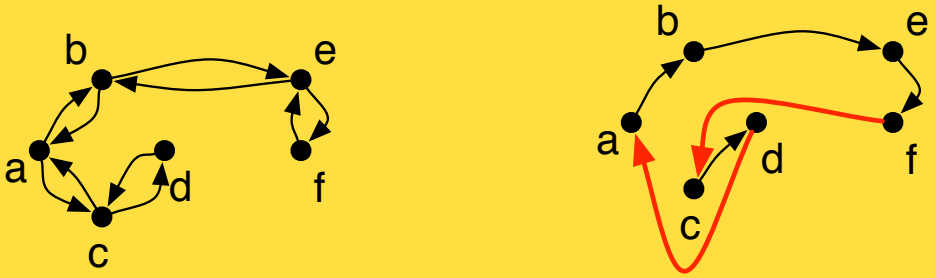
Example 15.11. *The figure on the right shows such a traversal using the spanning tree on the left. Starting at a, we can visit $a, b, e, f, e, b, a, c, d, c, a$.*

Now, recall that in the TSP problem, we have edges between any two vertices.

Question 15.12. *Can you find a way to derive a non-optimal solution to TSP using the particular approach to visiting vertices? Let's first try to eliminate multiple visits.*

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 15.13. *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.*



Question 15.14. *Assuming that edges are distances between cities, can we say anything about the lengths of the shortcut edges?*

By the triangle inequality the shortcut edges are shorter than the paths that they replace. Thus by taking shortcuts, we are not increasing the total distance.

Question 15.15. *What can you say about the weight of the TSP that we obtain in this way?*

Since we traverse each edge in the minimum spanning tree twice, the total weight of the approach with multiple visits is twice as much as that of the tree. With shortcuts, we obtain a solution to the TSP problem that is no more than the twice the weight of the tree (and possibly less). Since 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.

Remark 15.16. *It is possible to reduce the approximation factor to 1.5 using a well known algorithm by developed by Nicos Chistofides at CMU in 1976. The algorithm is also based on MST, and was one of the first approximation algorithms. Only recently were the bound improved, and only very slightly.*

15.3 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. This 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. This is easy to do since we can break ties in a consistent way. Given distinct weights, the minimum spanning tree of any graph is unique.

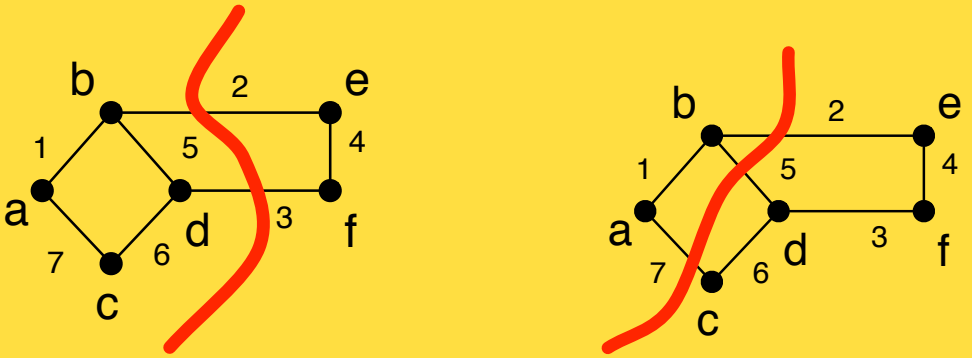
Exercise 15.17. *Prove that a graph with distinct edge weights has a unique minimum spanning tree.*

Definition 15.18. *For a graph $G = (V, E)$, a* cut *is defined in terms of a 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 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 $U \neq V$).

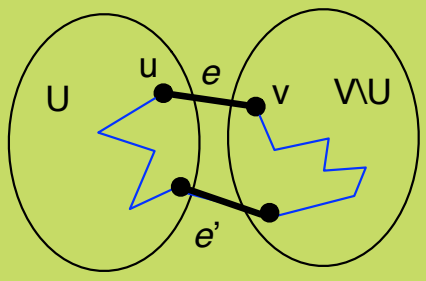
We sometimes say that a cut edge *crosses* the cut.

Example 15.19. *Some example cuts are illustrated. For each edge, we can find the lightest edge that crosses that cut.*



We are not ready for a formal statement and proof of the light-edge property.

Lemma 15.20 (Light-Edge Property)**.** *Let $G = (V, E, w)$ be a connected undirected weighted graph with distinct edge weights. For any nonempty $U \subsetneq V$, the minimum weight edge e between U and $V \setminus U$ 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 if we have done a graph search, the minimum edge between the visited set and the frontier (Prim's algorithm).

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.

Remark 15.21. *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 Czech Republic. It was re-invented many times over, the latest one as late as 1965.*

Kruskal's Algorithm

As described in the 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". The algorithm is correct since (1) since we are looking for a tree we need only consider edges that do not form cycles, and (2) among 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.

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 selected. 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. Furthermore the contraction can create duplicate edges with different weights. We would need to allow for duplicate edges, or remove all but the lightest of them.

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,

Algorithm 15.22 (Union-Find Kruskal)**.**

```
1  function kruskal(G = (V, E, w)) =
2  let
3    U = iter UF.insert UF.∅ V     (* insert vertices into union find structure *)
4    E' = sort(E, w)               (* sort the edges *)
5    function addEdge((U, T), e = (u, v)) =
6    let
7      u' = UF.find(U, u)
8      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 to T *)
12   end
13 in
14   iter addEdge (U, ∅) E'
15 end
```

$\texttt{find}(U, v)$ returns the supervertex in which v belongs, possibly itself, and $\texttt{equals}(U, u, v)$ returns true if u and v are the same. Now we can simply process the edges in increasing order. This idea gives Algorithm 15.22.

Question 15.23. *What is the work and the span of Kruskal's algorithm based on union-find?*

To analyze the work and span of the algorithm we first note that their 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.

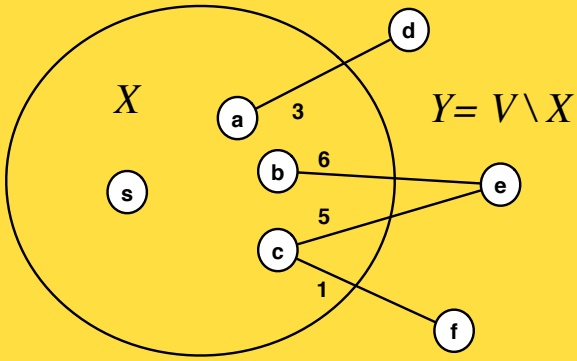
Prim's Algorithm

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 weight edge with one of its endpoint in 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 15.24 (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 and $w(u, v) = p(v)$.*

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

Example 15.25. *A step of Prim's algorithm. Since the edge (a, b) has minimum weight, the algorithm will "visit" f adding (c, f) to T and f to X.*



Remark 15.26. *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 15.27. *Carefully prove 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 15.28. *Write out the pseudocode for a Priority Queue based implementation of Prim's algorithm that runs in $O(m \log n)$ work.*

15.4 Parallel Minimum Spanning Tree

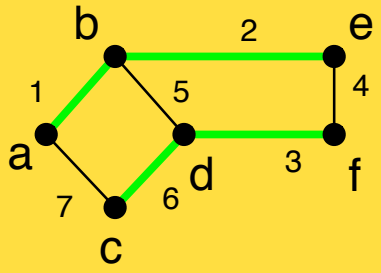
As we discussed, Kruskal and Prim's algorithm are sequential algorithms. We now focus on developing an MST algorithm that runs efficiently in parallel using graph contraction.

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 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.

Question 15.29. *What is the most trivial cut you can think of? What edges cross it?*

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 15.30. *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 find some edges in the MST. Furthermore since each edge can probably find its own minimum edge, it is likely this can be done in parallel.

Question 15.31. *Have we found all the MST edges? Can we stop?*

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 (e, f) since neither e nor f pick it.

Question 15.32. *Given that we have found some of the edges, how can we proceed?*

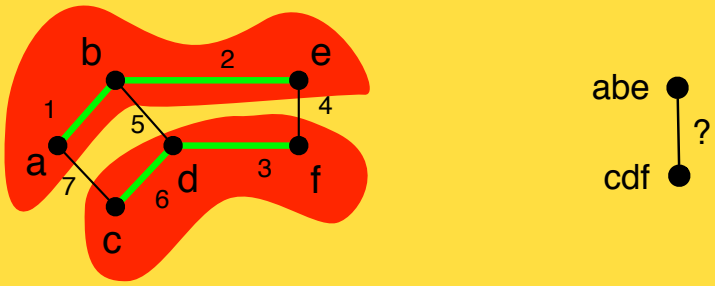
The edges that we have selected cover some of the cuts in the graph. We no longer have to consider edges that are within a component,

If we can somehow collapse the graph along the edges that we selected, we can proceed to consider the cuts that have not been covered.

Question 15.33. *How can we collapse the graph along the edges?*

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 15.34. *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 weren't 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 15.35 (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.

Question 15.36. *Suppose that we picked k minimum-weight edges, how many vertices will we remove?*

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 that every vertex will be removed since every vertex selects an edge.

Question 15.37. *Why is this not the case?*

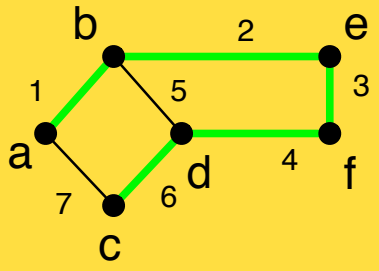
This is not the case since two vertices can select the same edge. Therefore there can be half as many edges as vertices, but there must be at least half as many. We therefore remove half the vertices on each round. This implies that Borůvka's algorithm will take at most $\log n$ rounds.

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.

Question 15.38. *What kind of contraction can we use, edge contraction, star contraction?*

In general the components identified by selecting minimum-weight edges are neither single edges nor single stars.

Example 15.39. *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).

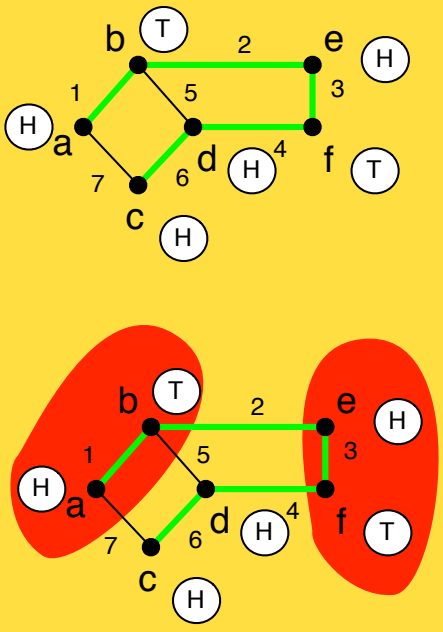
Exercise 15.40. *Prove that the minimum-weight edges will indeed form a forest. Recall that we are assuming that no two edge weights are equal.*

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 allow us to keep redundant edges. This can be done with $O(m)$ work as can finding the minimum edge into each component (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 fusing tree contraction into the algorithm directly instead of layering it. The idea is to apply randomized star contraction on the minimum-weight edges. This way, we will use star contraction instead of tree contraction. 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 factor of 2, leading to logarithmic number of rounds.

Example 15.41. *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 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 advantage of this second approach is that we will reduce the overall span for finding the MST from $O(\log^3 n)$ to $O(\log^2 n)$.

The modified algorithm for star contraction is as follows. In the code w stands for the weight of the edge (u, v).

Pseudo Code 15.42 (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 `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's 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 we're still contracting away $n/4$ vertices in expectation:

Lemma 15.43. *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by* `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 `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 $v \in$ `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 $(v, u) \in$ `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.

Algorithm 15.44 (Borůvka's based on Star Contraction)**.**

```
1  function minEdges (E) =
2  let
3     ET = {(u,v,w,l) ↦ {u ↦ (v,w,l)} : (u,v,w,l) ∈ E}
4     function joinEdges((v1,w1,l1),(v2,w2,l1)) =
5        if (w1 ≤ w2) then (v1,w1,l1) else (v2,w2,l1)
6  in
7     reduce (merge joinEdges) {} ET
8  end
9
10 function minStarContract(G = (V,E), i)
11 let
12    minE = minEdges(G)
13    P = {(u ↦ (v,w,ℓ)) ∈ minE | ¬heads(u,i) ∧ heads(v,i)}
14    V' = V \ domain(P)
15 in (V',P) end
16
17 function MST((V,E), T, i) =
18 if (|E| = 0) then T
19 else let
20    (V',PT) = minStarContract((V,E),i)
21    P = {u ↦ v : u ↦ (v,w,ℓ) ∈ PT} ∪ {v ↦ v : v ∈ V'}
22    T' = {ℓ : u ↦ (v,w,ℓ) ∈ PT}
23    E' = {(P[u],P[v],w,l) : (u,v,w,l) ∈ E | P[u] ≠ P[v]}
24 in
25    MST((V',E'), T ∪ T', i+1)
26 end
```

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

The function $\texttt{minEdges}(G)$ in Line 12 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 15.20, since all these edges are minimum out of the vertex, they are safe to add to the MST. Line 13 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 14 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 15.4 The MST algorithm is called by running $\texttt{MST}(G, \emptyset, r)$. As an aside, we know that T is a spanning forest on the contracted nodes.

Finally we describe how to implement $\texttt{minEdges}(G)$, 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 15.4 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.