Chapter 18

Minimum Spanning Trees

In this chapter we cover a important graph problem, Minimum Spanning Trees (MST). The MST of an undirected, weighted graph is a tree that spans the graph while minimizing the total weight of the edges in the tree. We first define spanning tree and minimum spanning trees precisely and then present two sequential algorithm and one parallel algorithm, which are respectively Kruskal's, Prim's, and Borůvka's. All of these algorithms utilize an important cut property, which we also desribe.

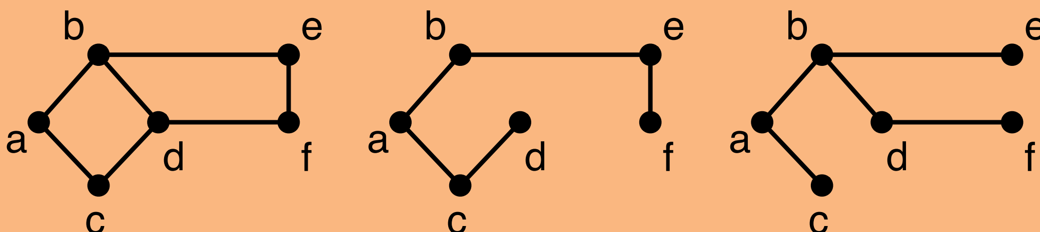
18.1 Minimum Spanning Trees

Recall that we say that an undirected graph is a forest if it has no cycles and a tree if it is also connected. Given a connected, undirected graph, we might want to identify a subset of the edges that form a tree, while "touching" all the vertices. We call such a tree a spanning tree.

Definition 18.1. *For a connected undirected graph $G = (V, E)$, a spanning tree is a tree $T = (V, E')$ with $E' \subseteq E$.*

Note that a graph can have many spanning trees, but all have $|V|$ vertices and $|V| - 1$ edges.

Example 18.2. *A graph on the left, and two possible spanning trees.*



Question 18.3. *Design 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. For example the DFS-tree of a DFS is a spannig tree, as it finds a path from a source to all the vertices. Similarly, we can construct a spanning tree based on BFS, by adding each edge that leads to the discovery of an unvisited vertex to the tree. DFS and BFS are work-efficient algorithms for computing spanning trees but as we discussed they are not good parallel algorithms.

Question 18.4. *Can you think of an algorithm with polylogarithmic span for finding a spanning tree of a connected undirected graph?*

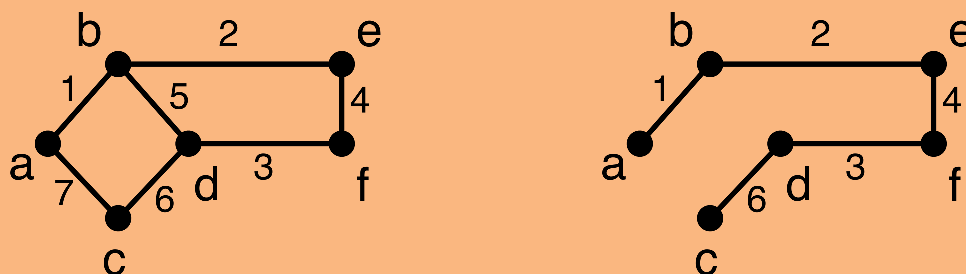
Another way to generate a spanning tree is to use graph contraction, which as we have seen can be done in parallel. The idea is to use star contraction and add all the edges that are selected to define the stars throughout the algorithm to the spanning tree.

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

Definition 18.5. *Given a connected, undirected weighted graph $G = (V, E, w)$, the minimum (weight) spanning tree (MST) problem requires finding a spanning tree of minimum weight, where the weight of a tree T is defined as:*

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

Example 18.6. *A graph (left) and its MST (right).*



Example 18.7. *Minimum spanning trees have many interesting applications. One example concerns the design of a network. Suppose that you are wiring a building so that all the rooms are connected via bidirectional communication wires. Suppose that you can connect any two rooms at the cost of the wire connecting the rooms, which depends on the specifics of the building and the rooms but is always a positive real number. We can represent the possible connection between rooms as a graph, where vertices represent rooms and weighted edges represent possible connections along with their cost (weight). To minimize the cost of the wiring, you could find a minimum spanning tree of the graph.*

18.2 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***. Intuitively, the light-edge property (precisely defined below) states that if you partition the graph into two, the minimum edge between the two parts has to be in the MST. The light-edge property gives a way to identify algorithmically the edges of an MST.

In our discussion we will assume that all edges have distinct weights. This assumption causes no loss-of-generality, because light-edge property allows us to break ties arbitrarily, which we can take advantage of by for example breaking ties based on some arbitrary ordering of edges such as their position in the input.

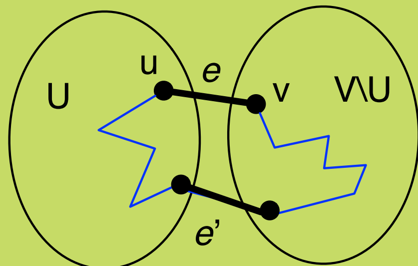
Question 18.8. *Consider a graph where each edge has a distinct weight, how many MST's can the graph have?*

A simplifying consequence of this assumption is that the MST of a graph with distinct edge weigths unique.

Definition 18.9. *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 written $E(U, \overline{U})$, where $\overline{U} = V \setminus U$.*

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., $U \neq V$). We sometimes say that a cut edge *crosses* the cut.

Lemma 18.11 (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. $\square$

Example 18.10. *Two example cuts. For each cut, we can find the lightest edge that crosses that cut, which are the edges with weight* 2 *(left) and* 4 *(right) respectively.*

We now state and prove the light-edge property (Lemma 18.11).

An important implication of Lemma 18.11 is that any minimum-weight edge that crosses a cut can be immediately added to the MST. In fact, all of the three algorithms that we will consider in this chapter take advantage of this implication. For example, Kruskal's algorithm constructs the MST by greedily adding the overall minimum edge. Prim's algorithm grows an MST incrementally by considering a cut between the current MST and the rest of graph. Borůvka's algorithm constructs a tree in parallel by considering the cut defined by each and

every vertex. In the next section, we briefly review Kruskal's and Prim's algorithm and spend most of our time on a parallel variant of Borůvka's algorithm.

Remark 18.12. *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.*

18.2.1 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.

Algorithm 18.13 (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
```

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 18.13.

Question 18.14. *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 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.

18.2.2 Prim's Algorithm

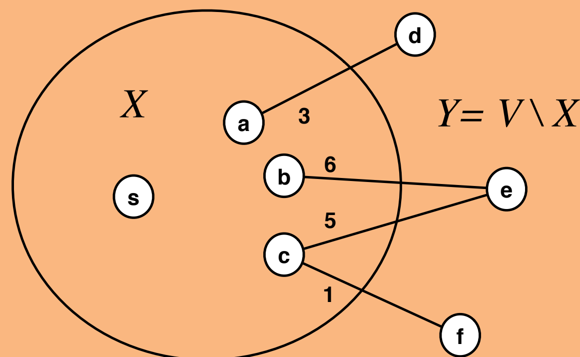
Prim's algorithm performs a priority-first search to construct the minimum spanning tree. The idea is that if we have already visited a set X, then by the light-edge property the minimum weight edge with one of its endpoint in X and the other in $V \setminus X$ must be in the MST (it is a

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 18.15 (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 18.16. *A step of Prim's algorithm. Since the edge $(\texttt{c}, \texttt{f})$ has minimum weight across the cut (X, Y), the algorithm will "visit" $\texttt{f}$ adding $(\texttt{c}, \texttt{f})$ to T and $\texttt{f}$ to X.*



Exercise 18.17. *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.

Remark 18.18. *Prim's 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.*

18.2.3 Borůvka's Algorithm

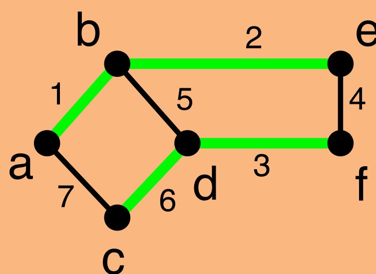
As discussed in previous sections, Kruskal and Prim's algorithm are sequential algorithms. In this section, we present an MST algorithm that runs efficiently in parallel using graph contraction. This parallel algorithm is based on an approach by Borůvka. As Kruskal's and Prim's, Borůvka's algorithm constructs the MST by inserting light edges but unlike them, it inserts many light edges at once.

To see how we can select multiple light edges, recall than all light edges that cross a cut must be in the MST.

Question 18.19. *What is the most trivial cut you can think of? What edges cross it?*

Consider now a cut that is defined by a vertex v and the rest of the vertices in the graph. The edges that cross this cut are exactly the edges incident on v. Therefore, by the light edge rule, for v, the minimum weight edge between it and its neighbors is in the MST. Since this argument applies to all vertices at the same time, the minimum weight edges incident an any vertex is in the MST. We call such edges *vertex-joiners*.

Example 18.20. *The vertex joiners 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`$\}$*.*



Question 18.21. *Have we found all the MST edges? Can we stop?*

Sometimes just one round of picking vertex-joiners will select all the MST edges and would generate a complete solution. However, in most cases, the minimum-weight edges on their own do not form a spanning tree. In the example above, we the edge $($`e`$,$ `f`$)$ is not selected (neither `e` nor `f` pick it).

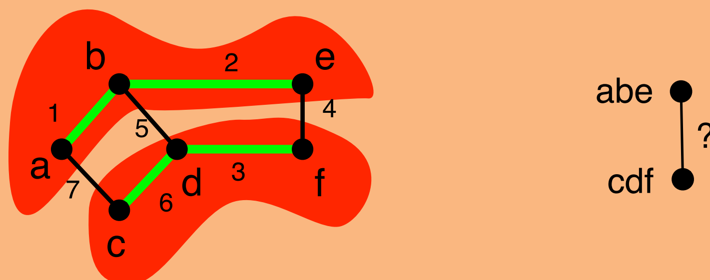
Question 18.22. *Given that we have found some of the edges, how can we proceed, can we eliminate some edges from consideration?*

To see how we can proceed, note that the vertex joiners define a partitoning of the graph—all the vertices are in a partition. Consider now the edges that remain internal to a partition. Such an edge is cannot be in the MST, because inserting it into the MST would create a cycle. The edges that cross the partitions, however, must be considered as they can indeed be in the MST.

Question 18.23. *How can we eliminate the internal edges?*

One way to eliminate the internal edges from consideration, while keeping the cross edges is to perform a graph contraction based on the partitioning defined by the vertex joiners. Recall that in graph contraction, all we need is a partitioning of the graph into disjoint connected subgraphs. Given such a partitioning, we then replace each subgraph (partition) with a supervertex and relabel the edges. This is repeated until no edges remain.

Example 18.24. *Contraction along the minimum edges. Note that there are redundant edges between the two partitions.*



When performing graph contraction, we have to be careful about redundant edges. In our discussion of graph contraction in Chapter 17, used unweighted graphs, we mentioned that we may treat redundant edges differently based on the application. In unweighted graphs, the task is usually simple because we can just keep any one of the redundant edges, and it usually does not matter which one. When the edges have weights, however, we have to decide to keep all the edges or select some of the edges to keep.

Question 18.25. *Which edge should we keep for computing the MST?*

For the purposes of MST, in particular, we can keep all the edges or keep just the edge with the minimum weight, because the others, cannot be in the MST. 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 18.26 (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.*

Cost of Borůvka by using tree contraction. 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 18.27. *Suppose that we picked k minimum-weight edges, how many vertices will we remove?*

Since contracting an edge removes exactly one vertex (contraction of and edge can be viewed as folding one endpoint into the other), if k edges are selected then k vertices are removed.

Question 18.28. *Can we then remove all the vertices?*

It is possible for $k = n$ and to remove all the vertices but k will generally be less than n, because two vertices can select the same edge.

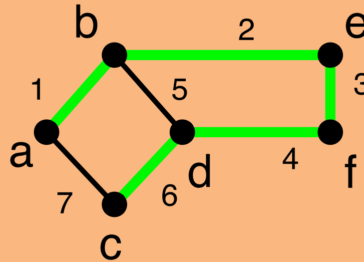
Question 18.29. *At least how many vertices can be remove?*

Therefore there must be at least $n/2$ edges and thus $n/2$ vertices will be removed. Consequently, Borůvka's algorithm will take at most $\log_2 n$ rounds of selecting vertex-joiners and contracting based on the partitioning defined by them.

Question 18.30. *How can be perform a round of contraction based on the partitioning defined by the vertex-joiners? Can we use edge contraction or star contraction?*

To contract the partition defined by the vertex joiners, we cannot use edge or star contraction, because the partitions may not correspond to edge or star partitions. In general each partition identified by selecting the vertex joiners are neither single edges nor single stars.

Example 18.31. *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.*

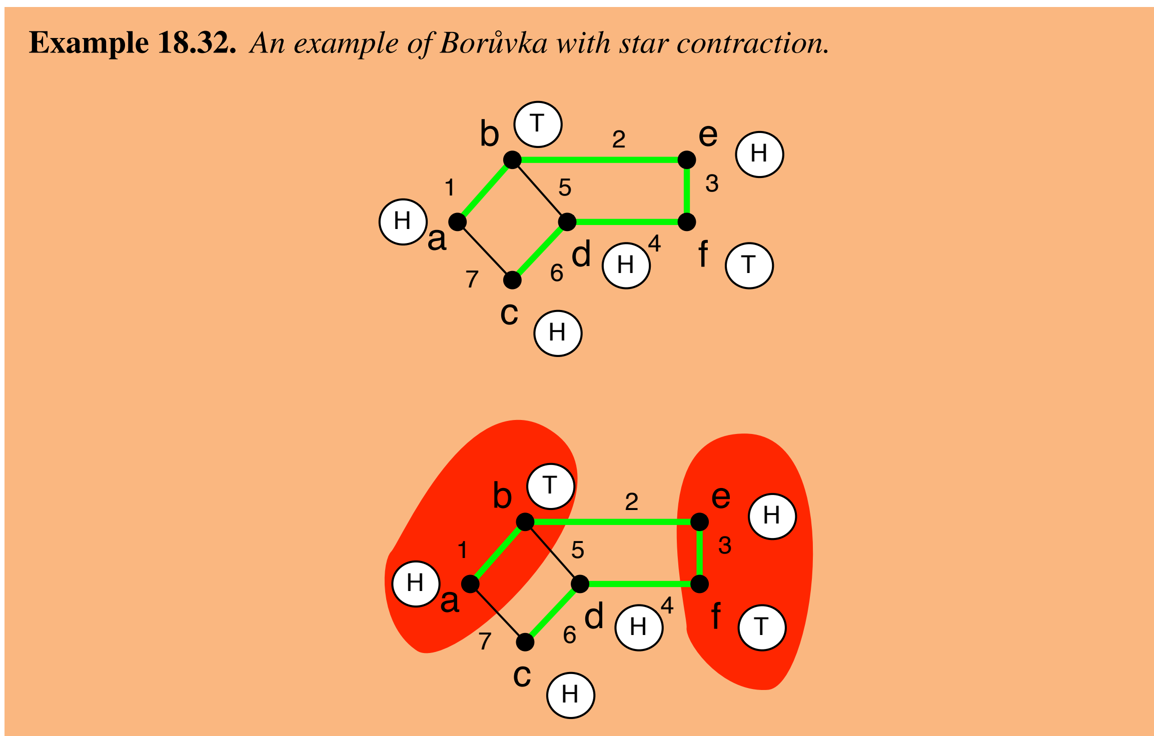


It turns out, the minimum-weight edges will form a forest (a set of trees). Therefore, the partitions are defined by general trees and thus we want to contract trees. By removing all edges that are not vertex joiners, we can contract a partition by applying star contraction to the partition. Furthermore since when doing a star contraction on a tree, it remains a tree on each step, the number of edges goes down with the number of vertices. Therefore the total work to contract all the partitions is 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.

Cost of Borůvka by using star contraction. We now describe how to improve the span of Borůvka by a logarithmic factor by interleaving steps of star contraction with steps of finding the vertex joiners, instead of fully contracting the trees defined by the vertex-joiners. The idea is to apply randomized star contraction on the subgraph induced by the vertex joiners, instead of considering the whole graph as in conventional star contraction. Intuitively, this is correct because we only have to care about vertex joiners (all other edges cannot be in the MST). As we will show, on each round, we will still be able to reduce the number of vertices by a constant factor (in expectation), leading to logarithmic number of total rounds. Consequently, we will reduce the overall span for finding the MST from $O(\log^3 n)$ to $O(\log^2 n)$ and maintain the same work.

Example 18.32. *An example of Borůvka with star contraction.*



For a set of vertex-joiners jE, consider the subgraph $H = (V, jE)$ of G and apply one step of the star contraction on H. To apply star contraction, we can 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).

Algorithm 18.33 (Star Contraction along Vertex Joiners)**.**

```
1 fun joinerStarContract(G = (V, E), i) =
2 let
3    val jE = vertexJoiners(G)
4    val P = {u ↦ (v, w) ∈ jE | ¬heads(u, i) ∧ heads(v, i)}
5    val V′ = V \ domain(P)
6 in (V′, P) end
```

where `vertexJoiners`(G) *finds the vertex joiners 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 18.34. *For a graph G with n non-isolated vertices, let X_n be the random variable indicating the number of vertices removed by* `joinerStarContract`(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 `jE`. 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.

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 (`vertex` × `vertex` × `weight` × `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 `joinerStarContract` that appears in Algorithm 18.35.

The function `vertexJoiner`(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 Lemma 18.11, 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 18.35 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

Algorithm 18.35 (Borůvka's based on Star Contraction).

```
1  function vertexJoiners (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,l2)) =
5        if (w1 ≤ w2) then (v1,w1,l1) else (v2,w2,l2)
6  in
7     reduce (merge joinEdges) {} ET
8  end

9  function joinerStarContract(G = (V,E),i)
10 let
11    minE = vertexJoiners(G)
12    P = {(u ↦ (v,w,ℓ)) ∈ minE | ¬heads(u,i) ∧ heads(v,i)}
13    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    (V′,PT) = joinerStarContract((V,E),i)
19    P = {u ↦ v : u ↦ (v,w,ℓ) ∈ PT} ∪ {v ↦ v : v ∈ V′}
20    T′ = {ℓ : u ↦ (v,w,ℓ) ∈ PT}
21    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
```

a singleton table for each edge and then merge all the tables with an appropriate function to resolve collisions. Algorithm 18.35 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.

18.3 Minimum Spanning Trees and the Travel Salesperson Problem

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.

Question 18.36. *Can you think of a way to bound the solution to a TSP problem on an undirected connected graph using minimum spanning trees.*

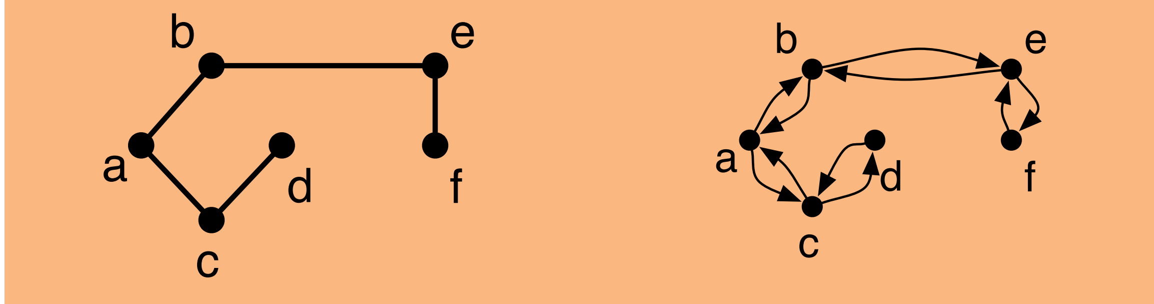
Since the solution to the TSP problem visits every vertex once (returning to the origin), it spans the graph. It is however not a tree but a cycle. Since each vertex is visited once, however, dropping any edge would yield a spanning tree. Thus a solution to the TSP problem cannot have less total weight 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.

Question 18.37. *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?*

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 18.38. *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`.

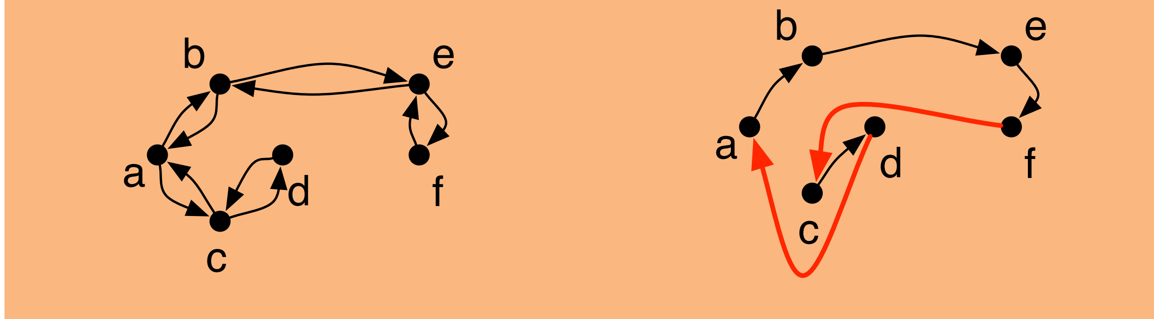


Now, recall that in the TSP problem it is assumed that there is an edge between every pair of vertices.

Question 18.39. *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 18.40. *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 18.41. *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 no longer than the paths that they replace. Thus by taking shortcuts, the total distance is not increased.

Question 18.42. *What can you say about the weight of the TSP that we obtain in this way?*

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 18.43. *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.*

18.4 Exercises and Problems

Exercise 18.44. *Prove that any tree with n vertices has $n - 1$ edges.*

Exercise 18.45. *Work out the details of the algorithm for spanning trees using graph contraction with star partitions (as mentioned in Section 18.1) and prove that it produces a spanning tree.*

Exercise 18.46. *Prove that the network with the minimum cost in Example 18.7 is indeed an MST of the graph.*

Exercise 18.47. *Write out the pseudocode for a Priority Queue based implementation of Prim's algorithm that runs in $O(m \log n)$ work.*

Problem 18.48. *Prove that a graph with distinct edge weights has a unique minimum spanning tree.*

Problem 18.49. *Prove that the vertex-joiners selected in any round Borůvka's algorithm form a forest. Recall that we are assuming that no two edge weights are equal.*