Lecture 17 — Minimum Spanning Trees

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2013)

Lectured by Umut Acar — 24 October 2013

1 Spanning Trees

Definition 1.1. For a connected undirected graph $G = (V, E)$, a spanning tree is a tree consists of all the vertices and some of the edges in the graph.

Note that a spanning tree of a graph G is a subgraph of G that *spans* the graph (includes all its vertices).

Example 1.2. *The tree on the right shows a spanning tree of the graph on the left.*

Question 1.3. *How many spanning trees does a graph with n vertices and m edges have?*

A graph can have many spanning trees. They all have n vertices and $n-1$ edges (all trees with n vertices have $n-1$ edges).

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

†Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

Example 1.4. *The tree on the right shows another spanning tree of the graph on the left.*

Question 1.5. *Can you think of an algorithm for finding a spanning tree of a connected undirected graph?*

You can use a DFS or BFS. DFS and BFS trees are spanning trees. They are indeed work-efficient algorithms for computing spanning trees but they are not good parallel algorithms.

Question 1.6. *Can you think of an algorithm with polylogarithmic span for finding a spanning tree of a connected undirected graph?*

You can in fact use graph contraction to compute a spanning tree in parallel. The idea would be to use star contraction and add all the edges that are picked to define a star as part of a spanning tree.

Exercise 2. *Work out the details of the algorithm for spanning trees using graph contraction and prove that it produces a spanning tree.*

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. A minimum spanning tree of an undirected, connected, weighted graph is a spanning tree whose weight (the sum of the weights of all the edges in the tree) is no greater than that of any other spanning tree.

Definition 2.1. The minimum (weight) spanning tree (MST) problem is given an connected undirected graph $G = (V, E)$, where each edge e has weight $w_e \geq 0$, find a spanning tree of minimum weight (i.e., the sum of the weights of the edges). That is to say, we are interested in finding the spanning tree T that minimizes

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

Minimum spanning trees have many interesting applications.

Example 2.2. *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 Travelling Salesman 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 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 always complete it by inserting edges with large weights that make sure that the edge never appears in a solution.

Question 2.3. *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, minimum spanning trees yield a lower bound on TSP.

Approximating TSP with MST It turns out you can also use minimum spanning trees to approximate solutions to the TSP problem, effectively finding an (not a tight one) upper bound.

Question 2.4. *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 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.

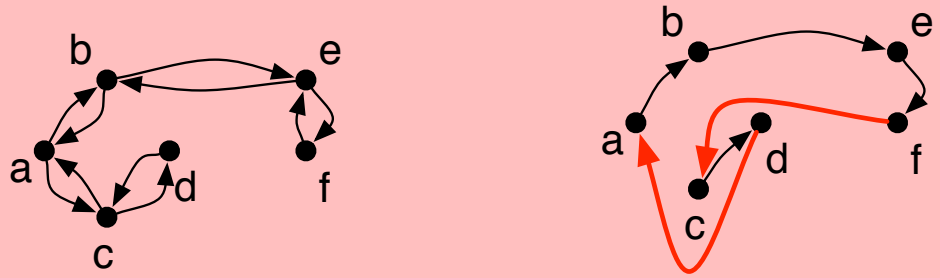
Example 2.5. *The figure on the right shows such a traversal using the spanning tree on the left. Starting at a, we can visit $a, b, e, f, e, b, a, c, d, c, a$.*

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

Question 2.6. *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.

Example 2.7. *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 2.8. *Assuming that edges are distances between cities, can we say anything about the lengths of the shortcut edges?*

By triangular equality such edges are shorter than the paths that they replace. Thus by taking shortcuts, we are not increasing the total distance.

Question 2.9. *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 soluction to the TSP problem and don't increase the total weight. Since TSP is no less that the minimum spanning tree (by our lower-bound argument), we have found an approximation algorithm for the Euclidian TSP problem, which is NP-hard.

Remark 2.10. It is possible to reduce the approximation factor to 1.5 using again minimum spanning trees.

3 Algorithms for Minimum Spanning Trees

There are several algorithms for computing minimum spanning trees. You have seen minimum weight spanning trees in 15-122 and possibly in 15-251. These classes possibly went over Kruskal's and Prim's algorithms.

Question 3.1. *Do you remember these algorithms?*

At a glance, Kruskal's and Prim's seem to be two drastically different approaches to solving MST: whereas Kruskal's sorts edges by weight and considers the edges in order, using a union-find data structure to detect when two vertices are in the same component and join them if not, Prim's maintains a tree grown so far and a priority queue of edges incident on the current tree, pulling the minimum edge from it to add to the tree. The two algorithms, in fact, rely on the same underlying principle about "cuts" in a graph, which we'll discuss next.

Remark 3.2. It is easy to confuse Prim's and Kruskal's algorithms. One mnemonic that might help you is that Prim's is a priority search algorithm (Prim's "P" stands for priority).

Light Edge Rule. The main property that underlines many MST algorithms is a simple fact about cuts in a graph. Here, 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.

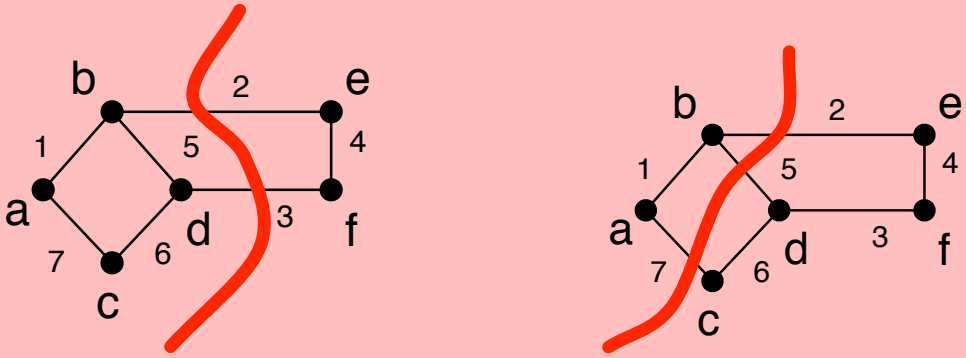
Exercise 3. *Prove that a graph with distinct edge weights has a unique minumum spanning tree.*

Definition 3.3. 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 $U \neq V$).

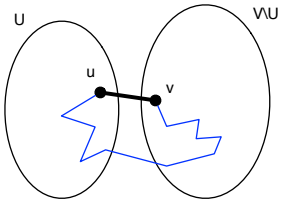
We sometimes say that a cut edge *crosses* the cut.

Example 3.4. *Some example cuts are illustrated. For each edge, we can find the lightest edge that crosses that cut.*



The following theorem states that the lightest edge across a cut is in the MST of G:

Theorem 3.5. *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 the 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 between u and v—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. □

Remark 3.6. Note that the last step in the proof uses the facts that (1) adding an edge to a spanning tree creates a cycle, and (2) removing any edge from this cycle creates a tree again.

The three algorithms for solving the MST problem are all based on Theorem 3.5: Kruskal's algorithm, Prim's algorithm and Borůvka's algorithm. Kruskal's and Prim's are based on selecting a single lightest weight edge on each step and are hence sequential, while Borůvka's selects multiple edges and 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 3.7. 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 The idea of Kruskal's algorithm is to sort the edges and then process them one at a time starting with the lightest edge. We maintain a set of components, where initially each vertex is its own component. When we process an edge we check if the two endpoints are in the same component. If so, we ignore the edge and move on. If not, we join the two components at the endpoints into a single component and add the edge to the MST. This joining is similar to the "edge" contraction we talked about in the last lecture, since it joins two components into one, except that we only do a single edge contraction on each step instead of multiple.

Algorithm 3.8 (Kruskal's Algorithm)**.**

1. *Sort edges by weight.*
2. *Start with every vertex in its own component.*
3. *For each edge (u, v) in sorted order:*
   If the u and v are in the same component skip.

   Otherwise, join the components and move on to the next edge.

Question 3.9. *Can you see why this algorithm always returns a minimum spanning tree?*

Question 3.10. *Can you see, why Kruskal's algorithm is correct.*

The intuitive reason for its correcteness is that it constructs a spanning tree by considering the edges from the smallest to the largest weight. In other words, it is almost a literal application of the light-edge rule. We can argue for correctness more carefully via proof by contradiction. Assume that the result is wrong. This means that some lightest edge e for a cut is not in the MST. Consider the time at which that e was considered by the algorithm. Since e is not chosen, its end points are in the same component and thus there is a path between them that crosses that path. Consider the edge e' that crosses the cut. Since the algorithm picks edges in sorted order, the weight of of e' is less than that of e. But then e is not the lighest edge crossing the cut; a contradiction. We therefore conclude that e is in the MST and thus the result is correct.

Question 3.11. *What is the work and the span of Kruskal's algorithm?*

We can use a union-find data structure to join components and to check if two vertices are in the same component. With an efficient implementation of union-find, the work for the algorithm is dominated by the need to sort the edges. The algorithm therefore runs in $O(m \log n)$ work. It is nearly fully sequential, i.e., its span is asymptotically is Ωm.

Prim's Algorithm

Prim's algorithm perfroms a priority search to construct the minimum spanning tree.

Question 3.12. *Do you recall how priority search worked?*

The idea of priority search is to keep a visited set X and a frontier F of vertices connected to vertices in X by an edge, while assigning a priority to each vertex in the frontier. When choosing the next vertex to visit from the frontier, priority search chooses the lowest (or highest) priority vertex.

Algorithm 3.13 (Prim's Algorithm)**.**

1. *Set visited set $X = \emptyset$.*
2. *Let $F = \{(v, -\infty)\}$, where f is any vertex.*
3. *Repeat until F is empty:*
   (a) *Pick from F the vertex u with the lowest priority.*
   (b) *Add v to X.*
   (c) *Update the frontier F by inserting each neighbor v of u prioritized by the weight of the edge (u, v).*
   (d) *Delete the visited vertices from F.*

In particular we start at an arbitrary vertex s, maintain a visited set X, and maintain priorities on the frontier vertices $v \in F$ based on the minimum weight edge from X to v. On each step we visit a vertex v with minimum priority via an edge (u, v) and add (u, v) to the MST. Since for MST we assume the graph is connected, this algorithm will visit all vertices.

Question 3.14. *Can you see why this algorithm is correct?*

At any point, the algorithm has found a minimum spanning tree of the algorithm. It then extends the minimum spanning tree by considering the visited set X and the rest of the graph and the ligtest edge into the minimum spanning tree.

Exercise 4. *Carefully prove correctness by induction.*

Remark 3.15. This algorithm was invented in 1930 by Czech mathematician Vojtech Jarnik and later independently in 1957 by competer scientist Robert Prim. As a person never into "shortest paths", Edsger Dijkstra's rediscovered it in 1959.

Question 3.16. *What is the work and span of Prim's algorithm?*

The algorithm runs with exactly the same work as Dijkstra's algorithm: $O(m \log n)$. As with

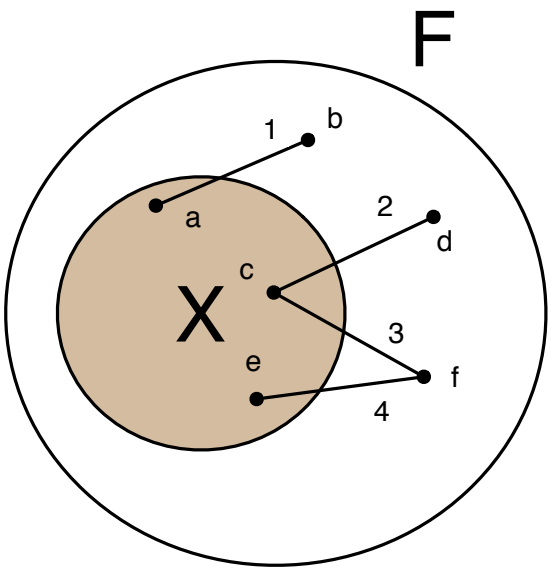


Figure 1: Prim's algorithm illustrated.

Dijkstra's algorithm it is fully sequential.

Here is the code for Prim's algorithm.

Pseudo Code 3.17 (Prim's Algorithm)**.**

```
 1  function prim(G) =
 2  let
 3    function enqueue v (Q,(u,w)) = PQ.insert (w,(v,u)) Q

 4    function prim'(X, Q, T) =
 5      case PQ.deleteMin(Q) of
 6          (NONE, _) ⇒ T                                  % Done
 7        | (SOME(d,(u,v)),Q') ⇒
 8          if (v ∈? X) then prim'(X, Q', T)               % Already visited
 9          else let
10                 X' = X ∪ {v}                            % Visit
11                 T' = T ∪ {(u,v)}                        % Add edge to MST
12                 Q'' = iter (enqueue v) Q' N_G(v)        % Enqueue v's neighbors
13          in prim'(X', Q'', T') end

14     s = an arbitrary vertex from G
15     Q = iter (enqueue s) {} N_G(s)                      % Enqueue s's neighbors
16  in
17    prim'({s}, Q, {})
18  end
```

This algorithm returns the set of edges in the MST. The only significant differences from Dijkstra's

algorithm is that the weight we use for the priority queue is w instead of $w + d$. There are a few other minor changes such as maintaining the MST T and storing an edge in the priority queue instead of just the target vertex.

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 looking at a parallel algorithm based on an approach by Borůvka, which we may even be able to invent on the fly. The 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 must be in the MST.

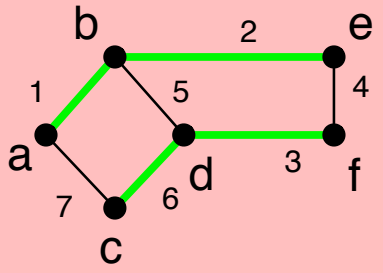
Question 4.1. *Consider a trivial cut, what edges cross the cut?*

A trivial cut is defined by a vertex v and the rest of the vertices in the graph. The edges that cross the cut are the edges incident to v. The edge with the minimum edge weight thus must be in an MST of the graph by the light-edge rule. We will refer to these edges as the *minimum-weight edges* of the graph.

Question 4.2. *Can you think of a way to pick many such light edges in one step?*

We can simply pick the minimum-weight edge incident on every vertex of a weighted graph G, because they all belong to the MST.

Example 4.3. *The minimum-weight edges of the graph are highlighted.*



Now we have found some edges in the MST.

Question 4.4. *Have we found them all? Can we stop?*

Sometimes we can stop, because it is indeed possible that we have selected $n-1$ distinct edges (we will see an example of this soon). But in most cases, we cannot because the same edge can be selected as the minumum weight edge by multiple vertices. Indeed, in our example, the edge (a, b) is selected by both a and b.

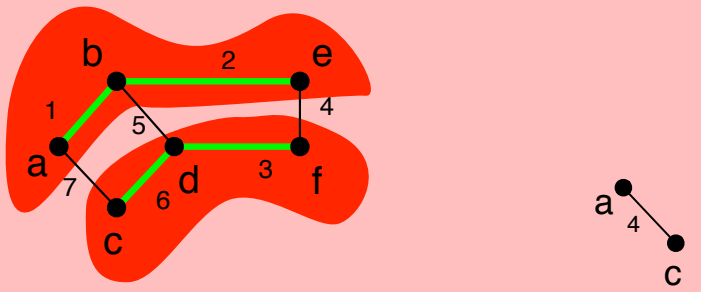
Question 4.5. *Given that we have found some of the edges, how can we proceed?*

The edges that we have selected cover some of the cuts in the graph. 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 4.6. *How can we collapse the graph along the edges?*

Borůvka's algorithm. This is exactly what graph contraction is for. Recall that in graph contraction, all we need is a partition of the graph into disjoint subgraphs. Gives such a partition, we can then select the subgraphs induced by each partiton, and contract the graph. So we can do the same here.

Example 4.7. *Contraction along the minimum edges. Note that we have to select the minimum edge to connect the contracted subgraphs.*



This 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.)

The Borůvka's algorithm can be described as follows.

Algorithm 4.8 (Borůvka)**.** *While edges remain do*

1. *Find minimum-weight edges (as part of MST).*
2. *Contract.*

Let's think about the efficiency of this algorithm. Let's focus on the number of rounds of contraction that we have to perform.

Question 4.9. *Suppose that we picked k minimum-weight edges, how many vertices will we remove?*

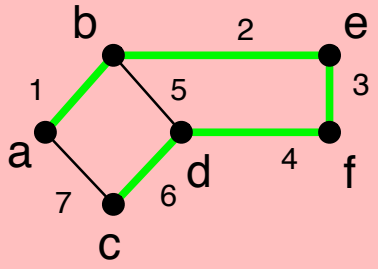
Since contracting edge removes exactly one vertex, we will remove k vertices. Now, we can see that we will therefore remove at least half of the vertices.

Exercise 5. *Prove that each Borůvka step must remove at least 1/2 the vertices.*

Question 4.10. *What kind of contraction can we use, edge contraction, star contraction?*

We cannot use edge contraction because the selected edges may not be independent, implying non-disjoint partitions. We cannot use star contraction either because the selected edges can be arbitrary trees, not necessarily just stars consisting of a center and satellites.

Example 4.11. *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 minumum-weight edges will form a forest.

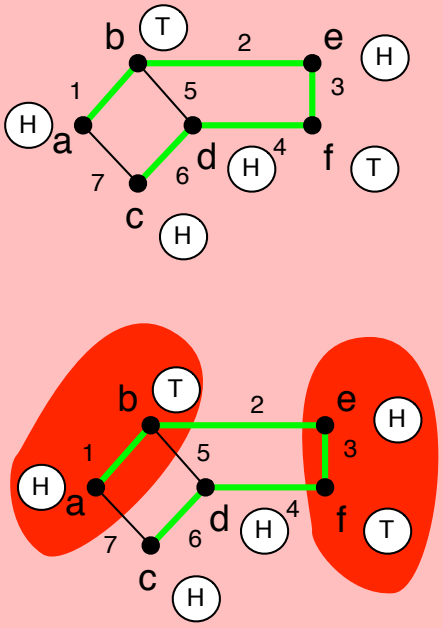
Exercise 6. *Prove that the minumum-weight edges will indeed form a forest.*

Question 4.12. *What is work and the span of the algorithm?*

Thus we will need to use tree contraction as a substep. This requires $O(m)$ work and $O(\log^2 n)$ span (if using array sequences). Furthermore every step must remove at least half the vertices. Therefore the algorithm will run in at most $\lg n$ rounds and hence will take $O(m \log n)$ work and $O(\log^3 n)$ span.

Borůvka's algorithm, improved with star contraction. We will now improve the span of Borůvka by a logaritmic factor by fusing tree contraction into the algorithm directly instead of layering it. The idea is to apply randomized star contraction on the minumum-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 4.13. *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 4.14 (Star Contraction along Minimum-Weight Edges)**.**

```
1  function minStarContract(G = (V, E), i) =
2  let
3      minE = minEdges(G)
4      P = {u ↦ (v, w) ∈ minE | ¬heads(u, i) ∧ heads(v, i)}
5      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 4.15. *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 $v \in \texttt{domain}(P)$ (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, $\mathbf{Pr}\left[R_v\right] \geq \mathbf{Pr}\left[T_v\right]\mathbf{Pr}\left[H_u\right] = 1/4$. By the linearity of expectation, we have that the number of removed vertices is

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

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 graph contracts but the weight and label stays fixed. This leads to the following slightly-updated version of `minStarContract`:

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

The function $\texttt{minEdges}(G)$ in Line 3 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 3.5, since all these edges are minimum out of the vertex, they are safe to add to the MST. Line 4 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 5 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.

```
1  function MST((V,E), T, i) =
2  if |E| = 0 then T
3  else let
4      (V', PT) = minStarContract((V,E), i)
5      P = {u ↦ v : u ↦ (v,w,ℓ) ∈ PT} ∪ {v ↦ v : v ∈ V'}
6      T' = {ℓ : u ↦ (v,w,ℓ) ∈ PT}
7      E' = {(P[u], P[v], w, l) : (u,v,w,l) ∈ E | P[u] ≠ P[v]}
8  in
9    MST((V', E'), T ∪ T', i + 1)
10 end
```

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 have to 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. Here is code that merges edges by taking the one with lighter edge weight.

```
function joinEdges((v1, w1, l1), (v2, w2, l1)) =
  if (w1 ≤ w2) then (v1, w1, l1) else (v2, w2, l1)

function minEdges(E) =
let
   ET = {(u,v,w,l) ↦ {u ↦ (v,w,l)} : (u,v,w,l) ∈ E}
in
  reduce (merge joinEdges) {} ET
end
```

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.