MSTs: Boruvka's Algorithm

1 Minimum Spanning Trees

Many weighted graphs in the real world have edges that are redundant. For example, your neighborhood's water system forms a weighted graph, where the weight of every edge is the length of the corresponding pipe. In such a graph, there are many paths between two vertices. However, it can also be helpful to reason about a restricted subset of such a graph. This is the motivation for the **minimum spanning tree**, which is a subgraph with certain special properties. We'll find out more about what the properties of MSTs are, as well as go over some algorithms for finding MSTs.

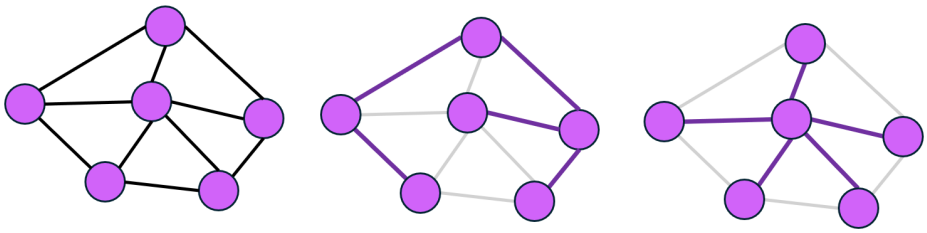
1.1 Definitions

Definition: Spanning Tree

A **spanning tree** of an undirected, connected graph $G = (V, E)$ is a tree $G' = (V, E')$, such that E' is a subset of E.

Essentially, a spanning tree keeps all of the vertices from the graph, but only keeps some of the edges so that the resulting subgraph is a tree—which is an undirected, connected graph with no cycles, which has exactly $n-1$ edges. In other words, it is a minimal subset of edges that keeps the vertices connected.

For example, in the picture below, a graph is shown on the left, and two possible spanning trees are shown to its right.



Now, given a *weighted* graph, we can define a *minimum* spanning tree of a graph as follows:

Definition: Minimum Spanning Tree

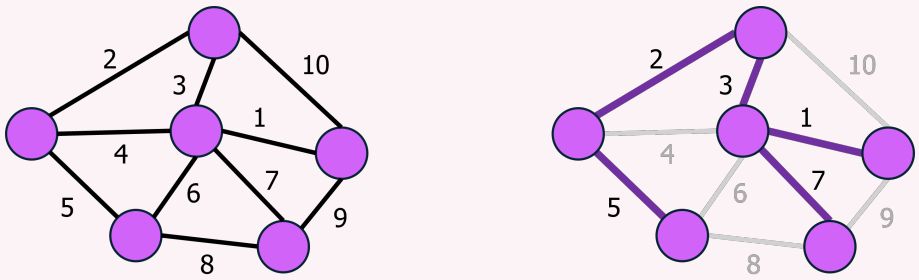
The **minimum spanning tree** (MST) of an undirected, connected graph $G = (V, E)$ is the spanning tree of G that minimizes $\sum_{e \in E'} w(e)$.

That is, we are minimizing the *sum* of the edge weights in the spanning tree.

Minimum spanning trees are immediately useful in some applications, such as laying out the internet cables in a building. Since every room in the building must be connected, we need our resulting subgraph to be connected. If we want to minimize the cost of the wire used, we can think of this problem as a graph where the edges are wires that we could lay, with weights corresponding to the costs in dollars. The MST of this graph corresponds to the set of wires needed to connect the building with minimum cost.

Example: Minimum Spanning Trees

Consider the following graph and its minimum spanning tree on the right:



The total weight of this spanning tree is $1+2+3+5+7 = 18$, which is the smallest possible.

Since for a spanning tree to even exist, the graph must have at least $n-1$ edges, we will simplify our analysis by assuming that $m \geq n-1$ for all inputs to our algorithms. In general, when this is not true, one can still find a "minimum spanning forest", which is an MST of each connected component, and indeed pretty much every MST algorithm still works for this scenario, with either minor or no modifications at all.

2 MST Properties

2.1 Properties of Trees

When we talk about trees, we typically are referring to undirected graphs which are connected and have no cycles. For a tree, the following three properties hold:

- A tree is connected.
- A tree has no cycles.
- A tree has $n-1$ edges.

As it turns out, knowing any two of these things on an undirected graph is sufficient to conclude that the third is true as well. We will occasionally also refer to **forests**, which are undirected graphs with no cycles that may not be fully connected—hence the name "forest", since the graph is made up of several trees.

What are the relevant properties of trees? The first is that adding any edge to a tree will create a

cycle. To argue that this is true, let's consider adding an edge (u, v) to a tree. However, vertices u and v were already connected in the tree, since a tree is a connected graph. Therefore, we now are able to go from u to v via the original path in the tree, and then go from v to u using the newly added edge – a cycle! Moreover, if we remove any edge from this cycle, we have a tree again. This is because removing an edge from the cycle we just created will lead to a connected graph with $n-1$ edges, which satisfy the properties on trees that we just defined.

2.2 Uniqueness of Edge Weights

Many algorithms and theorems regarding MSTs become simpler if we add the assumption that the weights of the graph are unique. Although this sounds like it comes with substantial loss of generality, it's actually easy to do in practice. Given a graph with duplicate weights, we can just add some kind of tie-breaking strategy to put a total order on the edges.

For example, one could simply assign each edge a unique index from 0 to $m-1$ before running the MST algorithm and use that as a tiebreaker for edges with equal weights. As long as the tie-breaking is deterministic and results in a total order, the edges will appear to be unique, and hence any algorithm that relies on unique weights will work correctly.

Theorem: Unique MST

In a graph with unique edge weights, there is a unique minimum spanning tree.

2.3 The Light-Edge Property

One important property of minimum spanning trees is the light-edge property, which specifies some edges that must be included in a minimum spanning tree. To understand the light-edge property, we must first define what a cut is on a graph:

Definition: Cut

A **cut** on a graph is defined as a partition of the vertices into two non-empty subsets, U and $V \setminus U$. Every vertex is in one of the two sets, and no vertex is in both.

An edge is defined as **crossing the cut** if it connects a vertex in U to a vertex not in U.

The following rule goes by several names, the "light-edge property", the "cut rule", or by Bob Tarjan as the "Blue Rule".

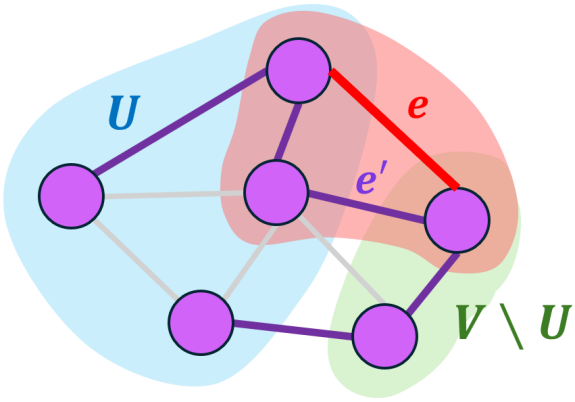
Theorem: Light-Edge/Cut/Blue Rule

Given a graph $G = (V, E)$, for any cut U and $V \setminus U$ on this graph, the lightest edge that crosses this cut must be in the minimum spanning tree of G.

If the edge weights are not unique, then the claim is that for *any* lightest edge crossing the cut, *there exists* an MST containing that edge (since the MST may no longer be unique either).

Proof. Assume for the sake of contradiction that e is not included in the MST. Then, there is

another edge e' across this cut that is in the MST T, since T must be connected. Consider the spanning tree created by removing e' and adding e to T.



This is, in fact, a tree because we added an edge to a tree, and then removed an edge from the resulting cycle. The total weight of this new tree must be less than the weight of T, since we removed e' and added an edge that has a smaller weight. Hence, T was not actually the minimum spanning tree, which is a contradiction. □

The light-edge property is very powerful on graphs where all of the edges have unique weights. In these graphs, every cut has a unique lightest edge across it, meaning that every cut tells us one edge that will be in the MST. Moreover, this property holds for *any* cut of the graph! This means that we can analyze a bunch of cuts in order to find all of the edges in the MST. This theorem is central to proving that many MST algorithms are correct.

2.4 The Cycle Property

Another useful property for finding MSTs is the **cycle property**, which is also known as the heavy-edge property, or as Bob Tarjan's "Red Rule".

Theorem: Heavy-Edge Property

Consider a weighted graph G that contains a cycle C. The maximum-weight (heaviest) edge in C cannot be in the MST of G.

Similar to the light-edge property, in a graph without unique weights, then the claim becomes that for any heaviest edge on the cycle, there exists an MST not containing it.

Proof. Let e be the heaviest edge in the cycle C, and assume for the sake of contradiction that $e = (u, v)$ is in the MST T. Remove e from T, which leaves two disconnected trees. Now, there exists a path from u to v corresponding to the other edges of the cycle C. Some edge on this path must connect the two disconnected trees, call it e', and by assumption, $w(e') < w(e)$ since e was the heaviest edge on the cycle. Thus, we can construct a cheaper spanning tree by adding e' back into the tree, creating a contradiction as T is not the MST of G. □

3 Sequential MST Algorithms

We will start by reviewing two sequential MST algorithms which you have seen before—Prim's and Kruskal's. Then we will focus on a new algorithm, one that unlike both of these, is highly parallel and makes use of graph contraction.

3.1 Kruskal's Algorithm

In Kruskal's algorithm, we will build a tree by adding edges one by one, so long as they do not create a cycle. To ensure that we get the minimum spanning tree out, we will add edges in order from lightest to heaviest.

Algorithm: Kruskal's Algorithm

```
type edge_list = sequence<(int,int,real)>

fun kruskal(edges : edge_list) -> edge_list:
  T = /* empty tree */
  for u,v,w in sort(edges):  // sorted by weight, ascending
    if u and v are not connected in T:
      add (u,v,w) to T
  return T
```

This is described in a high-level sort of pseudocode here, but it can be implemented efficiently by using the *union-find* data structure to keep track of whether u and v are connected. This results in a cost of $O(m \log n)$ (the cost of sorting the edges).

How do we know Kruskal's algorithm is correct? We can use the light-edge property. Consider any edge e added by Kruskal's and the resulting spanning tree. If we remove e, we create a cut consisting of the vertices on either side of e. This edge must be the lightest edge crossing that cut because if it were not, Kruskal's would have considered the lighter edge first, and it would connect two disconnected vertices, hence Kruskal would have chosen it. Therefore this edge is indeed part of the MST.

3.2 Prim's Algorithm

Prim's algorithm is similar to Dijkstra's algorithm in that it uses a greedy search to traverse the graph, starting at a single vertex and working outward. However, it uses a slightly different priority function. Instead of storing the weight of the path from the source vertex to a given vertex, we will store just the weights of edges that connect a visited vertex to a non-visited vertex.

Algorithm: Prim's Algorithm

```
type adj_list = sequence<sequence<(int,real)>>

fun prim(adj : adj_list) -> edge_list:
  s = 0  // arbitrary start vertex
```

```
T = /* empty tree */
visited = [False for _ in 0...|G.vertices|-1]

pq = PriorityQueue()
pq.insert((0,s,s))

while pq not empty:
  w, u, v = pq.deleteMin()
  if visited[u]: continue
  visited[u] ← True
  if u != v: add (u,v,w) to T

  for neighbor, weight in adj[u]:
    if not visited[neighbor]:
      pq.insert((weight, neighbor, u))
return T
```

As with Kruskal's algorithm, we can show the correctness of Prim's algorithm using the light-edge property. Every edge we add is the lightest edge across the cut formed by the sets X and $V \setminus X$, since it is the cheapest edge in our priority queue that goes from X to $V \setminus X$. Thus, every edge that we do end up adding is an MST edge, which means that we will eventually return the MST as our graph search will reach every vertex at some point.

The complexity of Prim's is also $O(m \log n)$, similar to Kruskal's. This bound comes from the fact that our priority queue insertions and deletions will take $O(\log n)$ time, and we will have m insertions and m deletions throughout the algorithm.

4 A Parallel MST Algorithm - Borůvka's

Now that we've taken a look at some sequential MST algorithms, it's time to develop a parallel MST algorithm. We'll be specifically looking at **Borůvka's algorithm**, which is a **contraction**-based MST algorithm. It was actually developed earlier in history than our sequential MST algorithms, and was initially created to determine the optimal placements of power lines.

Our sequential algorithms both added only a single edge at a time, using the light-edge property to justify that the edge it was adding was a valid MST edge. To design a parallel algorithm, we need some way of identifying many MST edges all at once in parallel.

One elegant way to do this is to use the concept of "vertex bridges".

Definition: Vertex Bridge

Given a vertex v, the vertex bridge of v is the **lightest edge** adjacent to v.

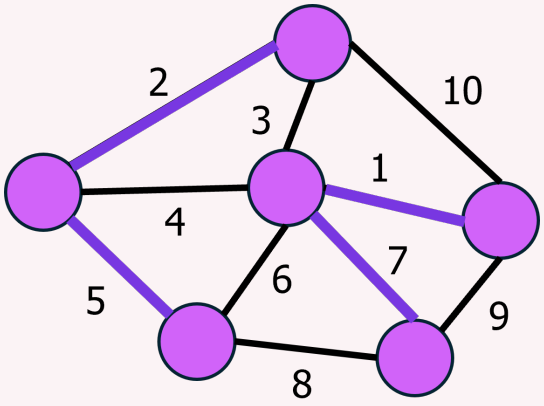
Why are these useful? The light-edge property once again tells us.

Theorem: Vertex Bridges are MST Edges

In a graph with unique edge weights, the vertex bridges of a graph are all in the MST.

Proof. For each v, consider the cut $U = \{v\}$. The lightest edge adjacent to v is by construction the lightest edge crossing this cut, and hence by the light-edge property is in the MST. □

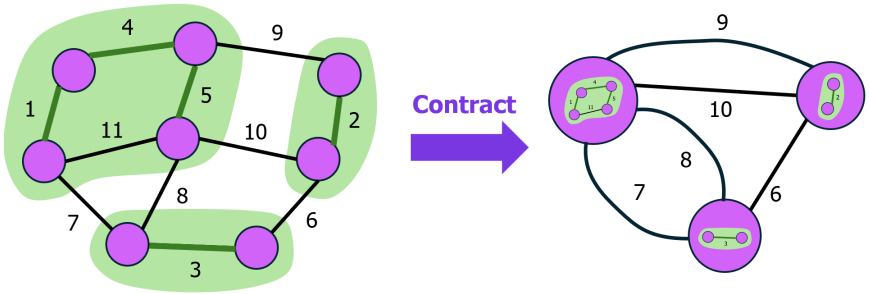
Example: Vertex Bridges



In this graph, the vertex bridges are highlighted in purple. Some edges are vertex bridges of multiple vertices, e.g., the 1-weight edge is the vertex bridge of both its endpoints, while other edges, like the 5-weight edge, is only the vertex bridge of one of its endpoints (the other endpoint has the 2-weight edge as its bridge).

Vertex bridges seem like a promising approach to designing a parallel algorithm since we can easily find them all in parallel and we know that they are all legal to add to the MST. The question that remains is what to do after we find all the vertex bridges? This will find many but not necessarily all of the MST edges, so how do we find the rest?

Well, once we add the vertex bridges, this induces a set of connected components in the original graph, of the vertices that can reach each other via their vertex bridges. It remains to connect these components to obtain a fully connected tree. Therefore, we can **contract** each connected component to a single vertex and remove edges that become self loops, and keep only the edges that connect different components.



To complete the MST, it suffices to take the bridge edges plus whatever edges form the MST of the contracted graph. This gives us all the ingredients to implement a recursive contraction algorithm! This algorithm, at a high level, is Borůvka's Algorithm.

Algorithm: Borůvka's Algorithm (High Level)

1. For each vertex in parallel, find its lightest adjacent edge (vertex bridge)
2. Contract these edges to form a smaller graph
3. Recursively find the MST of the contracted graph
4. Return the vertex bridges and the edges of the contracted MST

It remains to fill in the details of how we implement these steps and analyze their cost.

Finding the vertex bridges If the graph is represented as an adjacency list, finding the vertex bridges is easy. In parallel for each vertex, just reduce over the neighbouring edges and pick the lightest one. So, this can be implemented in $O(m)$ work and $O(\log n)$ span.

Contraction To contract the graph, we need to find the **connected components** of the graph induced by the vertex bridges. We can find the connected components by using **star contraction** with the vertex bridges as the edge set. Once we have found the connected components, the algorithm just filters out edges whose endpoints are in the same connected component (these are self-loops in the contracted graph) and then relabels the endpoints of the remaining edges with the connected component of the endpoint.

The cost of this process is dominated by the cost of star contraction to find the connected components. We earlier showed that star contraction can be implemented in $O((n+m)\log^2(m+n))$ expected work and $O(\log^3(n))$ span w.h.p. Filtering and relabeling edges takes just $O(m)$ work and $O(\log n)$ span, so this step costs $O(m\log^2(n))$ expected work and $O(\log^3 n)$ span w.h.p.

Theorem: Cost of Borůvka's Algorithm

Borůvka's Algorithm can be implemented with cost $O(m\log^3(n))$ expected work and $O(\log^4 n)$ span w.h.p.

Proof. One round of finding the vertex bridges and contracting the graph can be implemented with $O(m\log^2(n))$ expected work and $O(\log^3 n)$ span w.h.p. as described above.

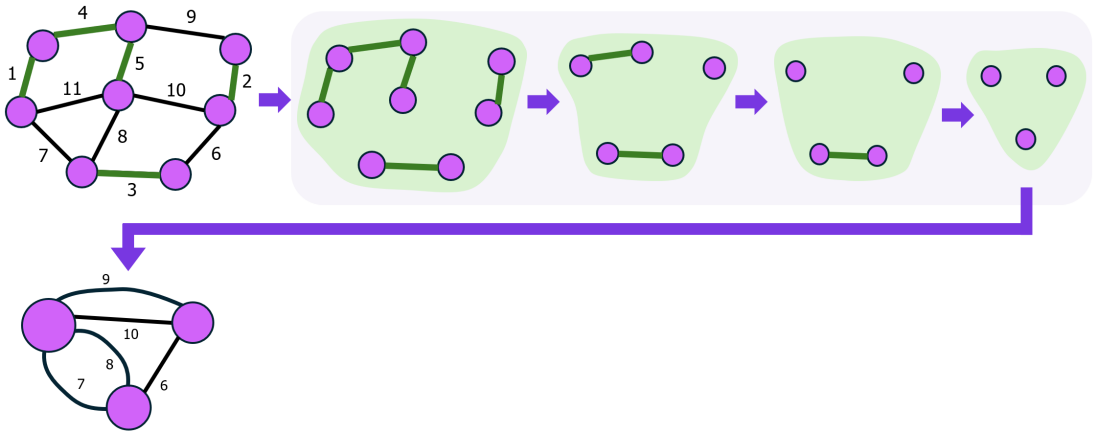
Given n vertices, there are up to n vertex bridges, but there will be fewer since two vertices may share a vertex bridge. Since each edge can be the bridge of at most two vertices, this means that there are *at least* $n/2$ distinct vertex bridges selected. Each edge contracted reduces the number of vertices by one, and hence and contracting $n/2$ vertex bridges removes $n/2$ vertices.

Therefore, it takes $O(\log n)$ recursive calls before the graph has one vertex and the algorithm is done. So, we multiply the cost of one round of the algorithm by $O(\log n)$ rounds to find that the total cost is $O(m\log^3(n))$ expected work and $O(\log^4 n)$ span w.h.p. □

How good is this? Compared to our sequential algorithms which both cost $O(m \log n)$, this implementation of Borůvka's costs a factor of $O(\log^2(n))$ more work. So, this is pretty close but not quite efficient compared to those. Can we do better?

4.1 A More Efficient Borůvka's

Borůvka's Algorithm is fundamentally a graph contraction algorithm—it shrinks the graph to a smaller one by finding the connected components induced by the vertex bridges and then recursively finds the MST of that smaller graph. Our first implementation of Borůvka's Algorithm performs the contraction/shrinking step using star contraction to find the connected components. This means that the contraction step of our algorithm is itself performing an entire graph contraction algorithm! So, we are doing contraction for $\log n$ rounds as part of one round of contraction for the overall algorithm. This sounds like it might be redundant!



The example above elucidates what is going on. We perform $\log n$ rounds of contraction using star partitioning to ultimately shrink the graph to half as many vertices. This process would repeat at the next level, another $\log n$ rounds of contraction to shrink the graph to half as many vertices again, and so on.

The goal of the contraction step is to produce a smaller graph to recursively find the MST of. It isn't necessarily required that we produce the *smallest possible* graph, just that the graph is a constant fraction smaller. So, shrinking the graph from n vertices to $n/2$, or half, is great, but it's overkill. If we could just eliminate say, one quarter of the vertices, leaving us with $3n/4$ vertices, that would be perfectly sufficient.

Looking at the picture above, an idea perhaps jumps right out!

Idea: Do Less Contraction!

Each round of star partitioning performed by star contraction is removing a constant fraction of the vertices until it is left with one vertex per connected component. We don't actually need to go all the way, we can just **do the first round of star partitioning and stop there!**

That is, we essentially just replace the use of star contraction on the vertex bridges to compute

the connected components, with a single round of star partitioning, instead of all $\log n$ rounds. From the graph contraction lecture, we remember that star partitioning removes one quarter of the vertices in expectation (a vertex is removed if it flips tails and at least one of its neighbours flips heads), so this is still enough to ensure that Borůvka's finishes in $O(\log n)$ rounds w.h.p.

So, the only change to the algorithm is the contraction step. Instead of fully computing the connected components, run one round of star partitioning. The edges that contract are included in the final MST. For the other edges, filter those that are in the same star, and relabel the endpoints of the remaining ones with their star center that they belong to.

Theorem: Cost of Borůvka's Algorithm (Improved)

Borůvka's Algorithm can be implemented with cost $O(m \log^2(n))$ expected work and $O(\log^3 n)$ span w.h.p.

Proof. The bounds are the same as the previous algorithm, except that the contraction step now only costs one round of star partitioning, which from earlier lectures costs $O(m \log n)$ expected work and $O(\log^2 n)$ span w.h.p.

Since star partitioning removes at least one quarter of the vertices in expectation, Borůvka's Algorithm will take $O(\log n)$ rounds w.h.p. to complete, and hence the total cost is $O(m \log^2(n))$ expected work and $O(\log^3 n)$ span w.h.p. □

4.2 A Final Version

Our improved implementation of Borůvka's costs $O(m \log^2(n))$ expected work and $O(\log^3 n)$ span w.h.p. which is still one log factor more expensive than our sequential algorithms. As a concluding note, we will mention that it *is* possible to improve this to $O(m \log(n))$ expected work and $O(\log^2 n)$ span w.h.p.

It actually does not require any more changes to Borůvka's, but rather, we just need a more efficient implementation of star partitioning. Star partitioning can in fact be implemented in $O(m+n)$ work and $O(\log n)$ span, which gives us an efficient implementation of Borůvka's.

Theorem: Cost of Borůvka's Algorithm (Finally)

Borůvka's Algorithm can be implemented with cost $O(m \log(n))$ expected work and $O(\log^2 n)$ span w.h.p.

Proof. Use an $O(m+n)$ work and $O(\log n)$ span implementation of star partitioning, which is possible, but we will not have time to cover it. □