Minimum spanning trees

Punchline: a MST of a graph connects all the vertices together while minimizing the number of edges used (and their weights).

Minimum spanning trees

Given a connected, undirected graph \( G = (V, E) \), a minimum spanning tree is a subgraph \( G' = (V', E') \) such that...

- \( V = V' \) (\( G' \) is spanning)
- There exists a path from any vertex to any other one
- The sum of the edge weights in \( E' \) is minimized.

In order for a graph to have a MST, the graph must...

- ...be connected – there is a path from a vertex to any other vertex. (Note: this means \( |V| \leq |E| \)).
- ...be undirected.

Minimum spanning trees: example

An example of an minimum spanning tree (MST):

Minimum spanning trees: Applications

Example questions:

- We want to connect phone lines to houses, but laying down cable is expensive. How can we minimize the amount of wire we must install?
- We have items on a circuit we want to be “electrically equivalent”. How can we connect them together using a minimum amount of wire?

Other applications:

- Implement efficient multiple constant multiplication
- Minimizing number of packets transmitted across a network
- Machine learning (e.g. real-time face verification)
- Graphics (e.g. image segmentation)

Minimum spanning trees: properties

Important properties:

- A valid MST cannot contain a cycle
- If we add or remove an edge from an MST, it’s no longer a valid MST for that graph.
- Adding an edge introduces a cycle; removing an edge means vertices are no longer connected.
- If there are \(|V|\) vertices, the MST contains exactly \(|V| - 1\) edges.
- An MST is always a tree.
- If every edge has a unique weight, there exists a unique MST.

Minimum spanning trees: algorithm

Design question: how would you implement an algorithm to find the MST of some graph, assuming the edges all have the same weight?

Hints:

- Try modifying DFS or BFS.
- Try using an incremental approach: start with an empty graph, and steadily add nodes and edges.
**Intuition:** We start with an "empty" MST, and steadily grow it.

**Core algorithm:**
1. Start with an arbitrary node.
2. Run either DFS or BFS, storing edges in our stack or queue.
3. As we visit nodes, add each edge we remove to our MST.

---

**An example using a modified version of DFS:**

Stack: (a, b), (a, d),

---

**Stack:**

(a, b), (d, e), (d, f), (d, g),

---

**Stack:**

(a, b), (d, e), (d, f), (g, h),

---

**Stack:**

(a, b), (d, e), (d, f), (g, h),
Minimum spanning trees: approach 1, adding nodes

An example using a modified version of DFS:

Stack: (a, b), (d, e), (d, f).

What if the edges have different weights?

Observation:
We solved a similar problem earlier this quarter, when studying shortest path algorithms!
**Interlude: finding the shortest path**

**Review:** How do we find the shortest path between two vertices?

- If the graph is unweighted: run BFS
- If the graph is weighted: run Dijkstra’s

How does Dijkstra’s algorithm work?

1. Give each vertex \( v \) a “cost”: the cost of the shortest-known path so far between \( v \) and the start.
   (The cost of a path is the sum of the edge weights in that path)
2. Pick the node with the smallest cost, update adjacent node costs, repeat

**Minimum spanning trees: approach 1, adding nodes**

**Intuition:** We can use the same idea to find a MST!

**Core idea:** Use the exact same algorithm as Dijkstra’s algorithm, but redefine the cost:

- **Previously, for Dijkstra’s:**
  The cost of vertex \( v \) is the cost of the shortest-known path so far between \( v \) and the start
- **Now:**
  The cost of vertex \( v \) is the cost of the shortest-known path so far between \( v \) and any node we’ve visited so far

This algorithm is known as **Prim’s algorithm**.

**Compare and contrast: Dijkstra vs Prim**

**Pseudocode for Dijkstra’s algorithm:**

```python
def dijkstra(start):
    backpointers = new SomeDictionary<Vertex, Vertex()
    for (v : vertices):
        set cost(v) to infinity
        set cost(start) to 0
    while (we still have unvisited nodes):
        current = get next smallest node
        for (edge : current.getOutEdges()):
            newCost = min(cost(current) + edge.cost, cost(edge.dst))
            update cost(edge.dst) to newCost
            backpointers.put(edge.dst, edge.src)
    return backpointers
```

**Pseudocode for Prim’s algorithm:**

```python
def prim(start):
    backpointers = new SomeDictionary<Vertex, Vertex()
    for (v : vertices):
        set cost(v) to infinity
        set cost(start) to 0
    while (we still have unvisited nodes):
        current = get next smallest node
        for (edge : current.getOutEdges()):
            newCost = min(edge.cost, cost(edge.dst))
            update cost(edge.dst) to newCost
            backpointers.put(edge.dst, edge.src)
    return backpointers
```

**Prim’s algorithm: an example**

![Prim's algorithm: an example](image.png)

We initially set all costs to \( \infty \), just like with Dijkstra.
Prim’s algorithm: an example

We pick an arbitrary node to start.

We update the adjacent nodes.

We select the one with the smallest cost.

We potentially need to update h and c, but only c changes.

...and update the adjacent nodes. Note that we don’t add the cumulative cost: the cost represents the shortest path to any green node, not to the start.
We update both unvisited nodes, and modify the edge to \( h \) since we now have a better option.

\[ i \] has the smallest cost.

Again, we update the adjacent unvisited nodes.

\[ f \] has the smallest cost.

We update \( h \) again.

\[ g \] has the smallest cost.
Prim’s algorithm: an example

$h$ has the smallest cost. Note that there nothing to update here.

Prim’s algorithm: another example

Now you try. Start on node $a$.

There are no more nodes left, so we’re done.
Analyzing Prim's algorithm

**Question:** What is the worst-case asymptotic runtime of Prim's algorithm?

**Answer:** The same as Dijkstra's: \( O(|V| t_s + |E| t_u) \) where...

- \( t_s \) = time needed to get next smallest node
- \( t_u \) = time needed to update vertex costs

So, \( O(|V| \log(|V|) + |E| \log(|V|)) \) if we stick to data structures we know how to implement; \( O(|V| \log(|V|) + |E|) \) if we use Fibonacci heaps.

---

Minimum spanning trees, approach 2

**Recap:** Prim's algorithm works similarly to Dijkstra's – we start with a single node, and “grow” our MST.

A second approach: instead of “growing” our MST, we...

- Initially place each node into its own MST of size 1 – so, we start with \(|V|\) MSTs in total.
- Steadily combine together different MSTs until we have just one left
- How? Loop through every single edge, see if we can use it to join two different MSTs together.

This algorithm is called **Kruskal’s algorithm**.
Kruskal’s algorithm

An example, for unweighted graphs. Note: each MST has a different color.
Kruskal’s algorithm

An example, for unweighted graphs. Note: each MST has a different color.

...
An example, for unweighted graphs. Note: each MST has a different color.
Kruskal’s algorithm

An example, for unweighted graphs. Note: each MST has a different color.
**Kruskal’s algorithm: weighted graphs**

**Question:** How do we handle edge weights?

**Answer:** Consider edges sorted in ascending order by weight. So, we look at the edge with the smallest weight first, the edge with the second smallest weight next, etc.

**Pseudocode for Kruskal’s algorithm:**

```python
def kruskal():
    mst = new SomeSet<Edge>()
    for (v : vertices):
        makeMST(v)
    sort edges in ascending order by their weight
    for (edge : edges):
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)
    return mst
```

- `makeMST(v)`: stores v as a MST containing just one node
- `findMST(v)`: finds the MST that vertex is a part of
- `union(u, v)`: combines the two MSTs of the two given vertices, using the edge (u, v)

**Kruskal’s algorithm: example with a weighted graph**

Now you try:

```
Now you try:
```

```
Now you try:
```

```
Now you try:
```
Now you try:

\[ a \quad 8 \quad c \quad 7 \quad d \]

\[ b \quad 4 \quad i \quad 2 \quad e \]

\[ h \quad 1 \quad g \quad 10 \quad f \]

\[ 4 \quad 8 \quad 11 \quad 7 \quad 4 \]

\[ 2 \quad 9 \quad 14 \quad 10 \quad 6 \]

\[ 1 \quad 2 \quad 1 \quad 6 \quad 7 \]

\[ 21 \]
Kruskal’s algorithm: example with a weighted graph

Now you try:

Now you try:

Kruskal’s algorithm: example with a weighted graph

Now you try:

Now you try:
Kruskal’s algorithm: example with a weighted graph

Now you try:

Now you try:

Now you try:

Now you try:

Now you try:

Now you try:
Kruskal’s algorithm: example with a weighted graph

Now you try:

```
a 11
h 1
b 8
c 7
d 10
```

Kruskal’s algorithm: example with a weighted graph

Now you try:

```
a 11
h 1
b 8
c 7
d 10
```

Kruskal’s algorithm: example with a weighted graph

Now you try:

```
a 11
h 1
b 8
c 7
d 10
```

Kruskal’s algorithm: example with a weighted graph

Now you try:

```
a 11
h 1
b 8
c 7
d 10
```

Kruskal’s algorithm: example with a weighted graph

Now you try:

```
a 11
h 1
b 8
c 7
d 10
```
Kruskal's algorithm: example with a weighted graph

Now you try:

```
Now you try:
```

Kruskal's algorithm: analysis

What is the worst-case runtime?

```
def kruskal():
    mst = new SomeSet<Edge>()
    for v : vertices:
        makeMST(v)
        sort edges in ascending order by their weight
    for edge : edges:
        if findMST(edge.src) != findMST(edge.dst):
            union(edge.src, edge.dst)
            mst.add(edge)
    return mst
```

Note: assume that...

- makeMST(v) takes $O(t_m)$ time
- findMST(v) takes $O(t_f)$ time
- union(u, v) takes $O(t_u)$ time

Kruskal's algorithm: analysis

```
Making the $|V|$ MSTs takes $O(|V| \cdot t_m)$ time

Sorting the edges takes $O(|E| \cdot \log(|E|))$ time, assuming we use a general-purpose comparison sort

The final loop takes $O(|E| \cdot t_f + |V| \cdot t_u)$ time
```

Putting it all together:

$O(|V| \cdot t_m + |E| \cdot \log(|E|) + |E| \cdot t_f + |V| \cdot t_u)$

The DisjointSet ADT

But wait, what exactly is $t_m$, $t_f$, and $t_u$? How exactly do we implement makeMST(v), findMST(v), and union(u, v)?

We can do so using a new ADT called the DisjointSet ADT!