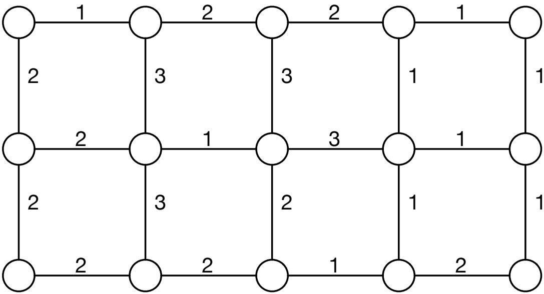
Minimum Spanning Trees & The Union-Find Data Structure

CMPUT 275 - Winter 2018

University of Alberta

Minimum Spanning Trees

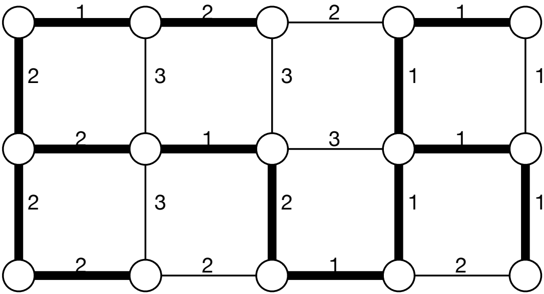
Given: A connected, undirected graph $G = (V; E)$ with edge costs $c(u, v) \geq 0$ for $uv \in E$.
Find: The cheapest $F \subseteq E$ such that graph $(V; F)$ is connected.



What is the cheapest set of edges required to maintain connectivity?

Minimum Spanning Trees

Given: A connected, undirected graph $G = (V; E)$ with edge costs $c(u, v) \geq 0$ for $uv \in E$.
Find: The cheapest $F \subseteq E$ such that graph $(V; F)$ is connected.



What is the cheapest set of edges required to maintain connectivity?

Applications

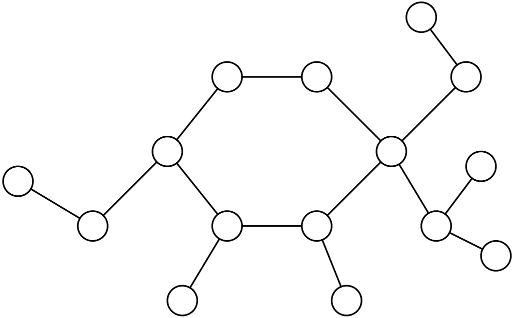
Network Design: Find the cheapest way to set up a network. Usually redundant edges are added to handle failures, but many heuristic algorithms start with spanning trees.

Approximation Algorithms for Hard Problems: Suppose you have to visit a set of clients in a road network. What is the fastest way to visit **all** and return home? Many problems like this do not seem to have efficient algorithms: heuristics build off of spanning trees.

Image Compression: Imagine you have a library of similar images (e.g. x-rays of the same body part from different people). Some compression algorithms exploit similarities between pictures to compress volumes of images, a minimum spanning tree (edge cost $\equiv$ differences between pictures) helps identify optimal compression rates for these compression schemes.

Basic Properties

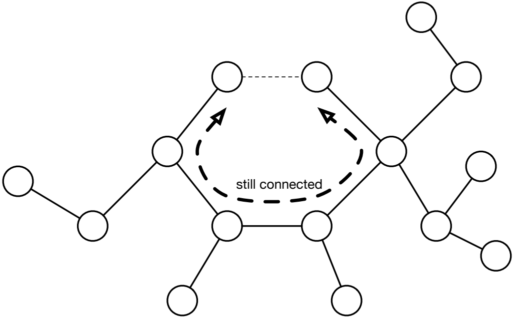
The cheapest set forms a **spanning tree**: a tree including all vertices.



Why? If there was a cycle, we could delete any edge on the cycle. This does not disconnect the graph: any path using this edge can instead go around the cycle.

Basic Properties

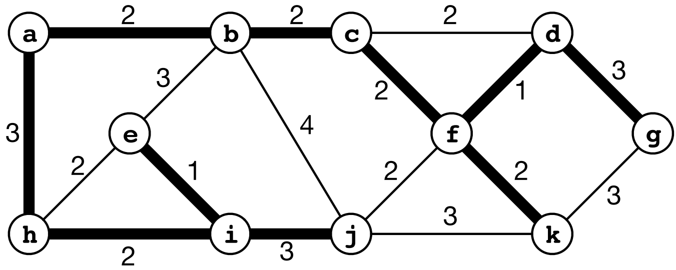
The cheapest set forms a **spanning tree**: a tree including all vertices.



Why? If there was a cycle, we could delete any edge on the cycle. This does not disconnect the graph: any path using this edge can instead go around the cycle.

Basic Properties

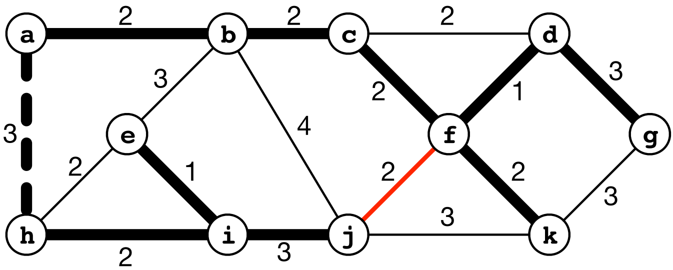
For this reason, the cheapest solution is called a **Minimum Spanning Tree**.



Is the set of bold edges in this picture a minimum spanning tree?

Basic Properties

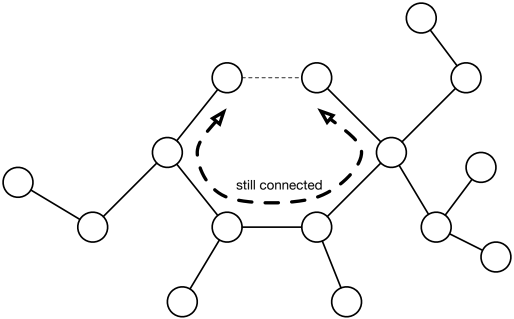
No: We can remove `ab` and add `jf` to get a cheaper spanning tree.



Idea to pick on: The edge `jf` was cheaper than some edge on the cycle it formed with the tree. So adding `jf` and removing this more expensive edge was a better solution.

Basic Properties

To be more explicit, for any tree $T = (V; F)$ and any edge $uv \notin F$, there is a unique path connecting u and v in T, so $F \cup \{u, v\}$ has exactly one cycle.



If uv is cheaper than some edge on this cycle, add uv and remove the more expensive edge!

Basic Properties

Property: For any minimum spanning tree $T = (V; F)$ of $G = (V; E)$, any $uv \in E - F$ (in E but not F) must be at least as expensive as any edge on the path connecting u to v in T.

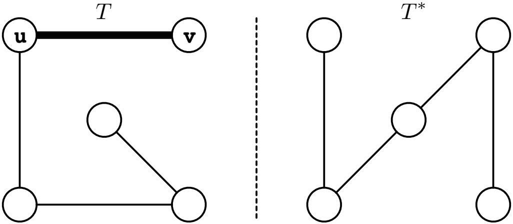
But is this enough? Does just knowing this property of some tree ensure it is a **minimum** spanning tree?

Yes
The algorithm will exploit this, so let's discuss why it is true.

Let $T = (V; F)$ be a spanning tree so each $uv \in E - F$ is at least as expensive as any edge on the $u - v$ path in T. Let $T^* = (V; F^*)$ be a minimum spanning tree.

If $F = F^*$ we are done, obviously. So suppose $F \neq F^*$.

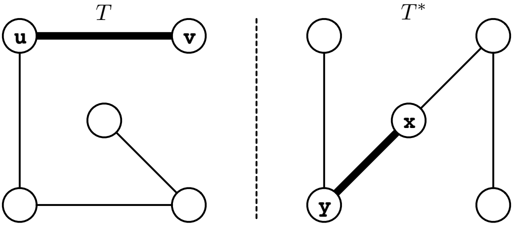
As $|F| = |F^*| = |V| - 1$ (all spanning trees have $|V| - 1$ edges), $F - F^* \neq \emptyset$. Let uv be the **cheapest** edge in f$F - F^*$.



Now, there is some edge on the $u - v$ path in T^* that is not in F.

Otherwise, the whole path along with uv, would form a cycle in the tree T!

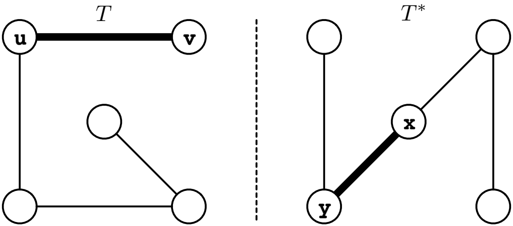
Call this edge xy (pictured in T^* below). Note xy and uv may share an endpoint even though they don't in the picture.



On one hand, $c(x, y) \geq c(u, v)$ which we can see as follows.

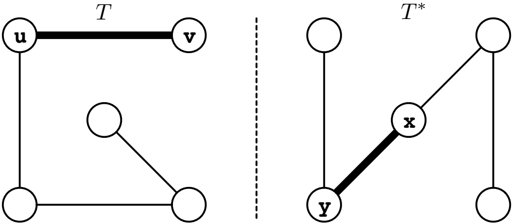
The $x - y$ path in T has an edge ab that is not in T^* (not pictured).

By our assumption on T, $c(x, y) \geq c(a, b)$. By our choice of uv as the cheapest edge not in T^*, $c(a, b) \geq c(u, v)$.

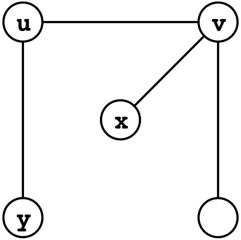


On the other hand, $c(x, y) \leq c(u, v)$ because T^* is a minimum spanning tree but we can get a spanning tree by swapping xy out and uv in to T^*.

So $c(x, y) = c(u, v)$.



Removing xy and adding uv gives another minimum spanning tree.



It is a spanning tree because xy was chosen to lie on the $u - v$ path in T^*. It is just as cheap because $c(u, v) = c(x, y)$.

We found another minimum spanning tree that shares even more edges in common with T.

Repeat the argument with this new tree until we get a minimum spanning tree having all edges in common with T (i.e. it is T).

The Algorithm

The algorithm is **greedy**. It builds the tree up using the cheapest edges first until it has a spanning tree.

It is called *Kruskal's Algorithm*.

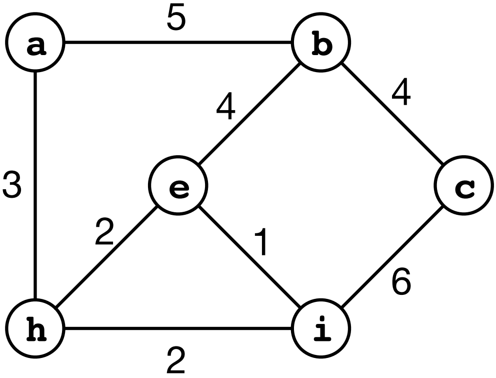
Algorithm 1 Kruskal's Minimum Spanning Tree Algorithm

```
F ← ∅
for each edge uv in nondecreasing order of cost do
   if u and v are not in the same connected component of (V; F) then
      F ← F ∪ {uv}
return F
```

Simple!

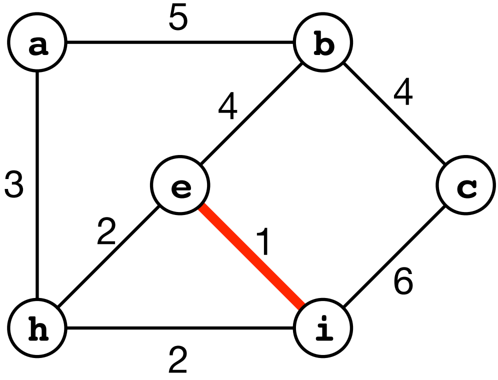
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



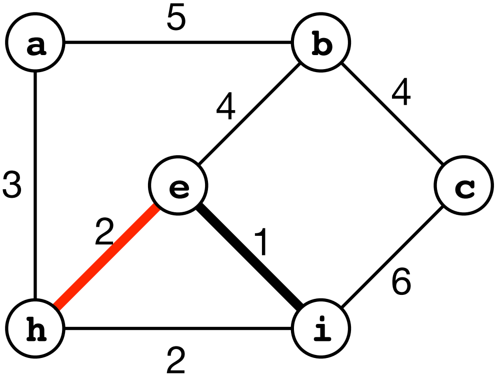
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



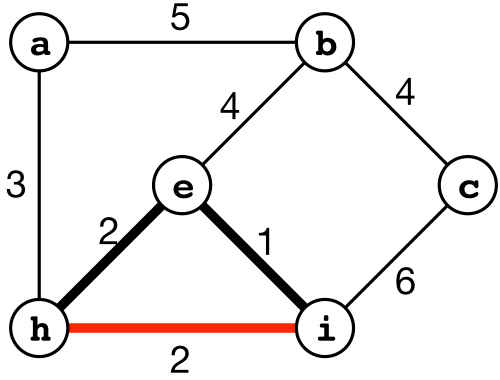
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



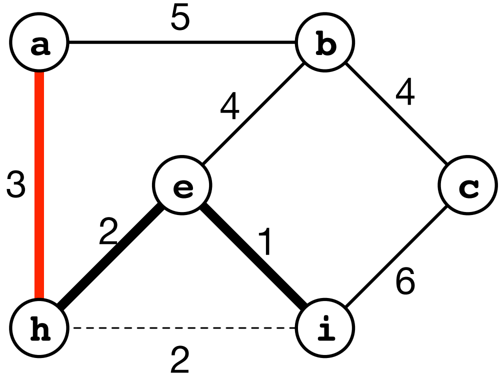
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



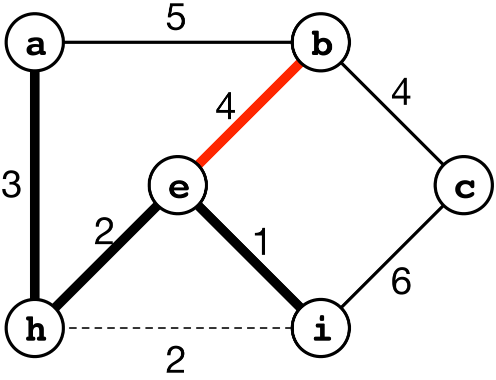
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



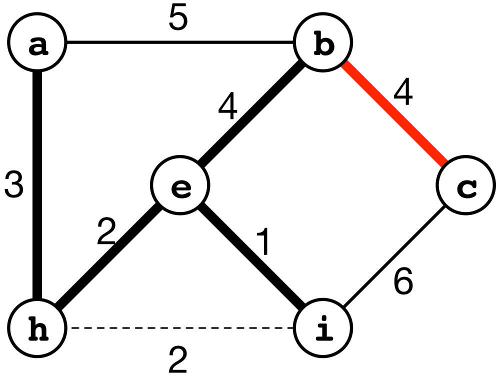
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



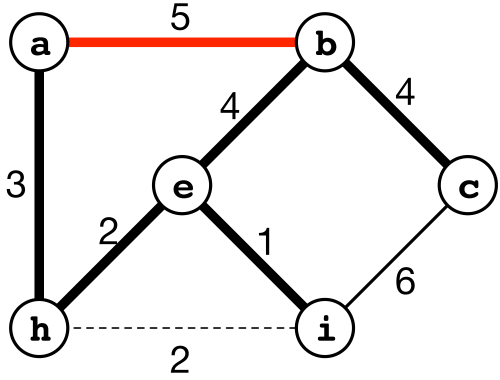
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



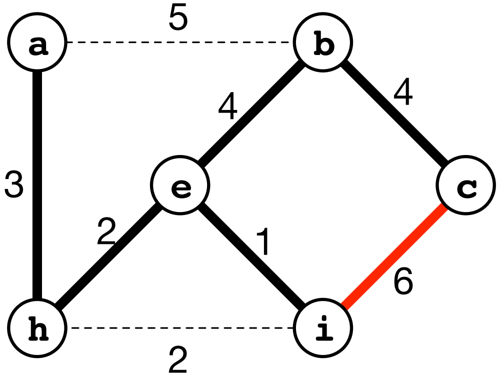
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



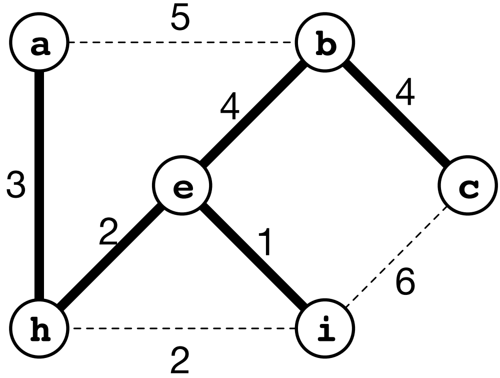
Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.



Tracing the Algorithm

Algorithm summary: consider edges in order of cost. If the endpoints aren't connected, keep the edge.

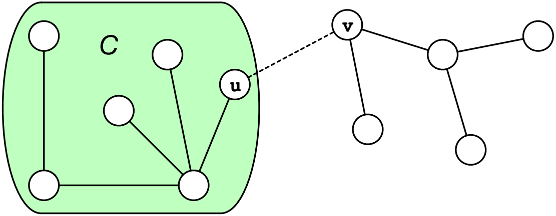


Does This Work?

F never contains a cycle. If it did, say uv was the last edge added. But then u and v were already connected so uv would not be added!

If G is connected, then $(V; F)$ will be connected at the end too.

Otherwise, let C be a connected component. Some $uv \in E$ has exactly one endpoint in C (otherwise G is not connected and C is one of its components). But then uv should have been added to F!

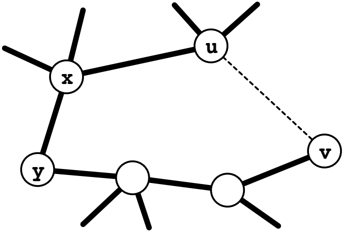


Does This Work?

OK, we get a spanning tree. Is it a minimum spanning tree?

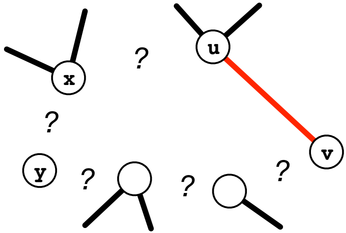
Use Our Characterization
Let $uv \in E - F$ (an edge we didn't buy).

Consider the $u - v$ path in F and say xy is the last edge on the path considered by the algorithm.



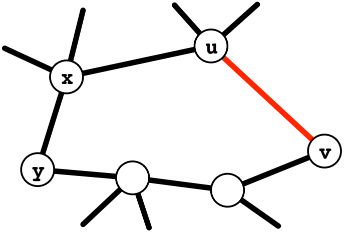
Does This Work?

Since *uv* is not purchased, *u* and *v* are already connected!



Does This Work?

But the final tree has a unique $u - v$ path, so when uv is being considered the whole path was already there!



Observation: uv is considered after xy, so $c(u, v) \geq c(x, y)$.

As this holds for any uv not on the tree, the tree must be a minimum spanning tree!

Musings

This is called a **greedy** algorithm. There is no precise definition of what this means, but the idea it invokes is that naive choices are made without consideration for the whole problem.

Here, we repeatedly greedily picked the cheapest edge that wouldn't form a cycle.

Often greedy algorithms fail for nontrivial problems. I chose to present the theory before the algorithm to emphasize that we need certain properties to be assured the greedy approach works!

We will cover more greedy algorithms later (assignment 2 is built around a different one). You will get more exposure to the idea.

Running Time

Algorithm 2 Kruskal's Minimum Spanning Tree Algorithm

$F \leftarrow \emptyset$
for each edge uv in nondecreasing order of cost **do**
  if u and v are not in the same connected component of $(V; F)$ **then**
    $F \leftarrow F \cup \{uv\}$
return F

We can use a BFS to check if u, v are connected for a total time of $O(|V| \cdot |E|)$ (there are $\leq |V|$ edges in the tree at any time, so the BFS runs in time $O(|V|)$.

Technically there is an $O(|E| \log |E|)$ sort at the start too.

A Better Algorithm

Once Again

Use of a fancy data structure can cut the running time down, this case to $O(|E|\log|E|)$ (and this is only due to the sorting part).

We use the **Union-Find** data structure to help us manage connected components.

Definition: A partition of a set S is a collection of subsets $S_1, S_2, \ldots, S_k$ so each $a \in S$ lies in exactly one subset.

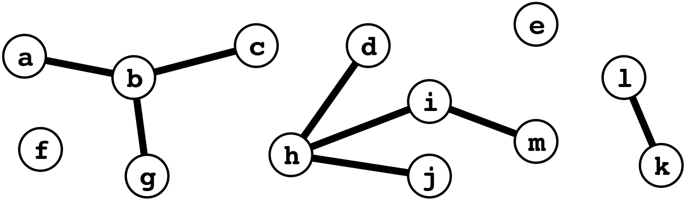
Example: $\{1,4\},\{5\},\{2,3,6\}$ is a partition of $\{1,2,3,4,5,6\}$.

Example: $\{1,2,3,4\}$ is a partition of $\{1,2,3,4\}$.

Example: $\{1\},\{2\},\{3\},\{4\}$ is a partition of $\{1,2,3,4\}$.

Union-Find

In Kruskal's algorithm, we need to track the connected components. The connected components form a partition of the set of all vertices!

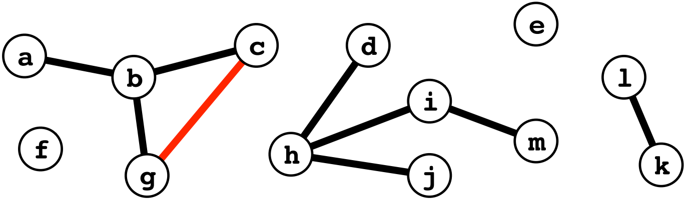


The connected components pictured partition the vertices as

$$\{a, b, c, g\}, \{d, h, i, j, m\}, \{e\}, \{f\}, \{l, k\}$$

Union-Find

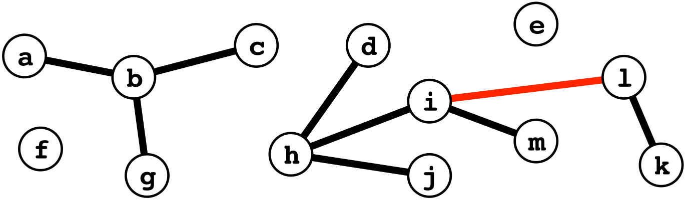
When considering an edge in Kruskal's algorithm, we just ask if the endpoints are in the same part of the partition.



Pictured: The endpoints of cg are in the same part $\{a, b, c, g\}$, so don't keep the edge (they are connected already).

Union-Find

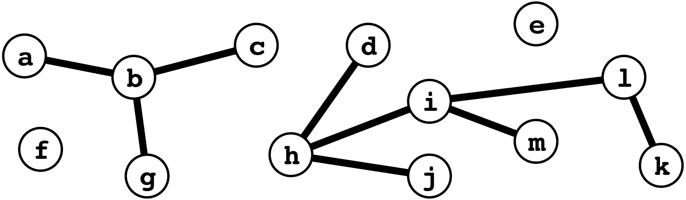
When considering an edge in Kruskal's algorithm, we just ask if the endpoints are in the same part of the partition.



Pictured: The endpoints of il are in the different parts: $\{h, d, i, j, m\}$ and $\{l, k\}$. Keep the edge!

Union-Find

When considering an edge in Kruskal's algorithm, we just ask if the endpoints are in the same part of the partition.



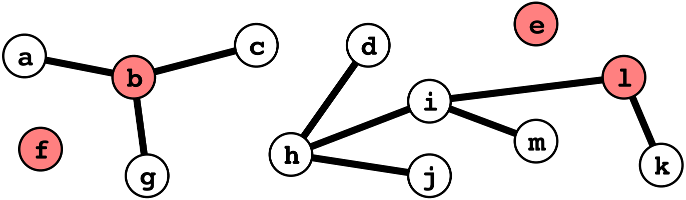
Pictured: Keeping the edge merges the two components. The partition is now

$$\{a, b, c, g\}, \{d, h, i, j, k, l, m\}, \{e\}, \{f\},$$

A `Python3` set quickly supports membership checking, but merging two sets of total size n runs in $O(n)$ time. This is not good enough.

The **Union-Find** data structure we will discuss maintains a partition of a set S.

It also maintains a distinct representative of each part of the partition.



To check if two items are in the same set, just check they have the same representative!

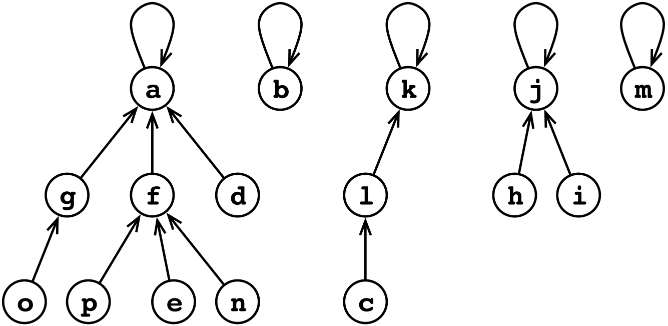
The supported operations are:

- `init(S)` - initialize to the partition of S where each item is in a part by itself
- `find(x)` - return the representative of the part containing x
- `union(x,y)` - merge the parts containing x and y (if different)

Example: If $S = \{1, 2, 3, 4\}$ then `init(S)` would create an instance of the Union-Find data structure that initially has the partition

$$\{1\}, \{2\}, \{3\}, \{4\}$$

Internally, each part of the partition is a rooted tree where items point toward the root. The root is the representative for that part.



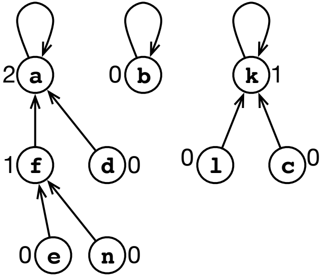
One slight difference: the root also has an edge pointing to itself.

Python Representation

A dictionary storing where each item points.

Example: {`g:a, a:a, c:l, m:m, p:f, ...`}

To ensure the operations are efficient, a little bit of extra data will be recorded to make sure the trees aren't too tall.



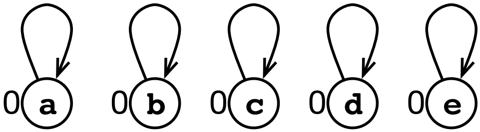
For each item x, a value $rank(x)$ will be kept with the following properties.

- For each representative x of a part S_i, $|S_i| \geq 2^{rank(x)}$.
- For any item x that is **not** a representative, $rank(x) < rank(parent(x))$.

Initializing For Set S

Have each item S point to itself.

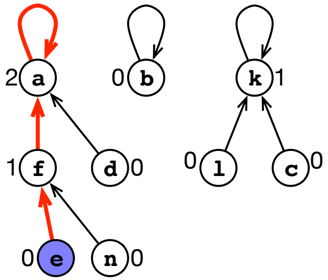
Set the rank of everything to 0.



Done!

Finding the representative of x

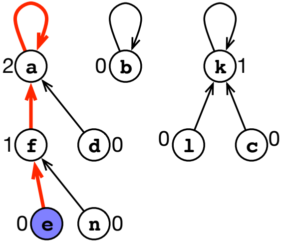
Crawl up the tree until you hit the representative.



Algorithm 3 find(x)

while $x \neq parent(x)$ **do**
    $x \leftarrow parent(x)$
return x

Running Time of Find



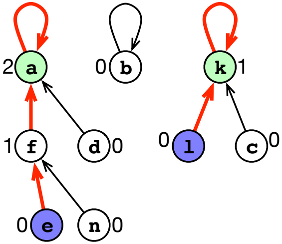
Obviously $O(\#$ height of the tree). But what is this?

Claim: A rooted tree with representative x has *height* $\leq$ *rank*(x).
Obvious because the heights strictly decrease down the tree.

But $\#$ items in this tree is $\geq 2^{rank(x)}$. Taking logarithms:
$rank(x) \leq \log_2 n$. So `find` takes $O(\log n)$ time.

Union Operation

Here the structure changes, so we have to make sure the properties of ranks is maintained.



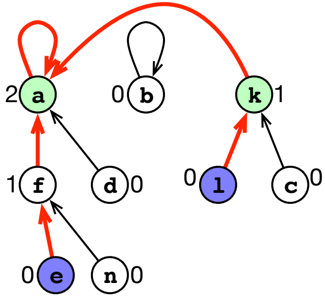
First, find the representatives of the items.

Pictured

find(e) and *find*(l) return a and k (respectively)

Union Operation

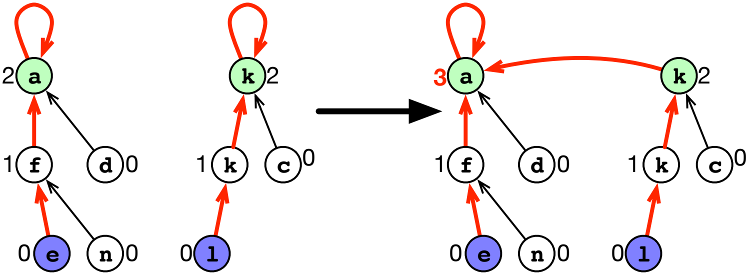
If the ranks are different, set the parent of the lower-rank representative to the higher rank representative.



Do not change any ranks.

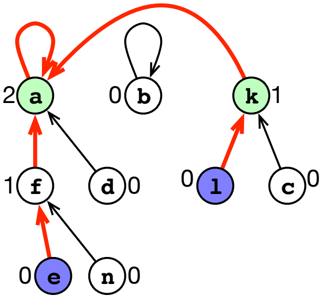
Union Operation

If the ranks are the same, pick one to be the new representative and increase its rank by 1.



Recall we want:

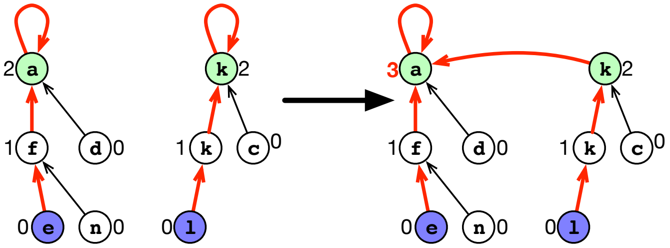
1. For each part S_i with representative x, $|S_i| \geq 2^{rank(x)}$
2. For each non-representative x, $rank(x) < rank(parent(x))$



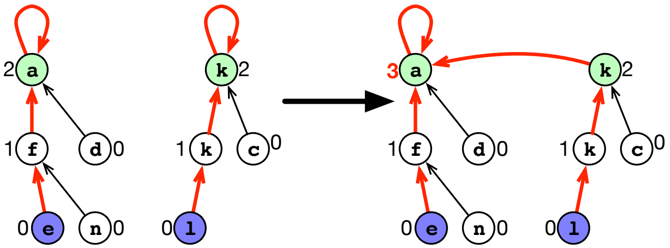
Pretty easy to see this still holds if the representative ranks were different.

Recall we want:

1. For each part S_i with representative x, $|S_i| \geq 2^{rank(x)}$
2. For each non-representative x, $rank(x) < rank(parent(x))$



If the ranks were the same, we increased the rank of the new representative (so point #2 still holds).



Say we merged parts S and T. Let *rank*() denote the old ranks and *rank*′() denote the new ranks.

Let ry be the representative of the new part $S \cup T$ (**pictured**: a) and rx the representative of the old part T (**pictured**: k).

$$|S| + |T| \geq 2^{rank(rx)} + 2^{rank(ry)} = 2 \cdot 2^{rank(ry)} = 2^{rank(ry)+1} = 2^{rank'(ry)}$$

So #1 continues to hold.

Pseudocode

Algorithm 4 union(x, y)

```
rx, ry ← find(x), find(y)              # find their representatives
if rx == ry then
    return False                       # already lie in the same set, no merge
if rank(rx) > rank(ry) then
    swap(rx, ry)
parent(rx) ← ry                        # ry is the new representative
if rank(rx) == rank(ry) then
    rank(ry) ← rank(ry) + 1
return True                            # there was a merge
```

Running Time: Everything except the two find operations runs in $O(1)$ time. So $O(\log n)$.

Back to Kruskal

Algorithm 5 Kruskal's Minimum Spanning Tree Algorithm

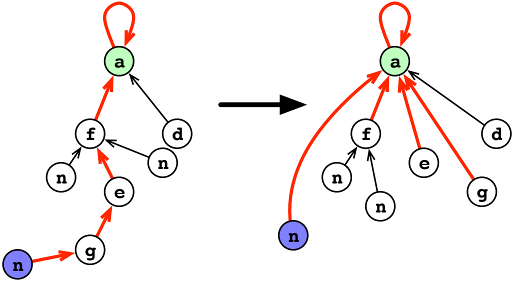
```
F ← ∅
uf ← UnionFind(V)
for each edge uv in nondecreasing order of cost do
    if uf.find(u) ≠ uf.find(v) then
        F ← F ∪ {uv}
        uf.union(u, v)
return F
```

The running time is dominated by an $O(|E| \log |E|)$ sort and the $O(|E|)$ union and find operations, each taking $O(\log |V|)$ time.

In Total: $O(|E| \log |E|)$
Done!

Even Faster Union-Find: Path Compression

What if we collapsed the entire "find chain" to the root?
Future calls would run faster.



Algorithm 6 find(x) with path compression

if $x \neq parent(x)$ **then**
$\quad parent(x) \leftarrow find(parent(x))$
return $parent(x)$

Even Faster Union-Find: Path Compression

Even though the structure changes, it is easy to verify the rank properties still hold. No need to adjust them!

In the worst case, a `find` operation can still take $O(\log n)$ time.

But the **total** running time over k successive calls to `union` and `find` is: $O(\alpha(k) \cdot k)$ where $\alpha()$ is a crazy-slow growing function (**much** slower than $\log k$).

It is called the **inverse Ackermann** function (a keyword to look up on Google if you want).

Technically $\alpha(k) \to \infty$ as $k \to \infty$, but $\alpha(2^{192837128417}) \leq 4$ so, practically speaking, this is a constant!

Done: I Mean It This Time!

But we still lose $O(|E| \log |E|)$ from sorting so it doesn't improve the asymptotic running time of Kruskal's.

If the edges were, somehow, already in sorted order this would take $O(\alpha(|E|) \cdot |E|)$ time in total: **linear time** for all practical purposes (even though it technically isn't).

Amazingly: Someone described a sequence of `union` and `find` operations that would cause the total running time to be at least $c \cdot \alpha(|E|) \cdot |E|$ for some constant c!

So the worst-case analysis with this funny function $\alpha()$ is tight!