CMPT 280

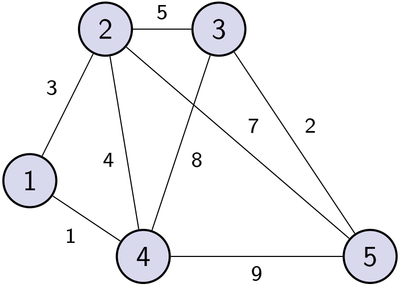
Tutorial: Kruskal's Algorithm

G. Scott Johnston

University of Saskatchewan

Kruskal's Algorithm: Example

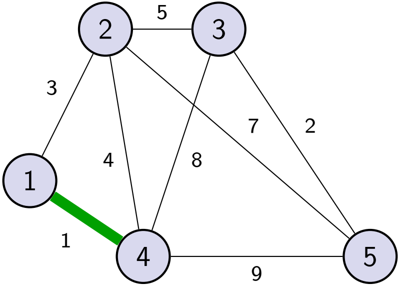
Suppose we are finding the minimum spanning tree of this graph:



Exercise: what **is** the minimum spanning tree of this graph?

Kruskal's Algorithm: Example

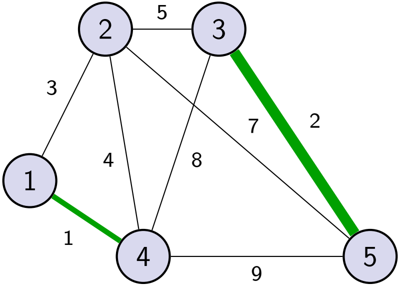
Incrementally add the shortest edge to the minimum spanning tree:



The edge with weight 1 is the shortest.

Kruskal's Algorithm: Example

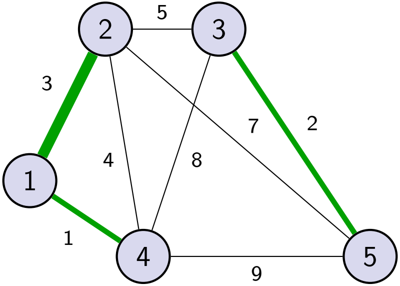
Incrementally add the shortest edge to the minimum spanning tree:



The edge with weight 2 is next.

Kruskal's Algorithm: Example

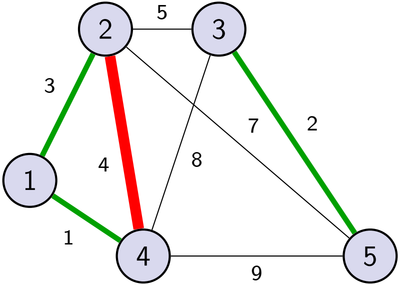
Incrementally add the shortest edge to the minimum spanning tree:



The edge with weight 3 is next.

Kruskal's Algorithm: Example

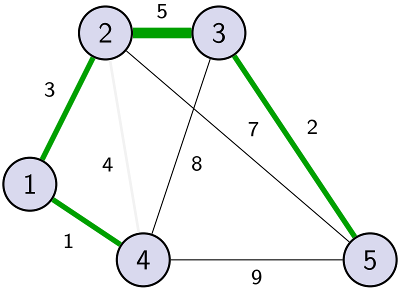
Incrementally add the shortest edge to the minimum spanning tree:



But nodes 2 and 4 are already connected in the tree!
So, don't even bother adding it.

Kruskal's Algorithm: Example

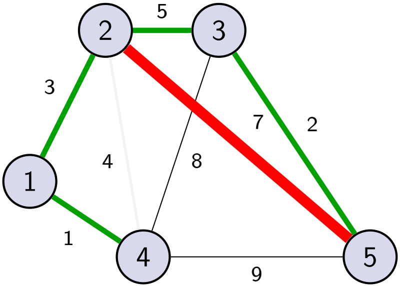
Incrementally add the shortest edge to the minimum spanning tree:



The edge with weight 5 is the next smallest.

Kruskal's Algorithm: Example

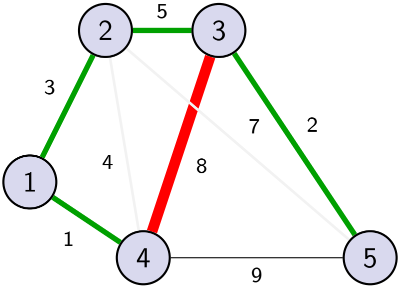
Incrementally add the shortest edge to the minimum spanning tree:



Because the tree is finished, the remaining edges should also be rejected when we see if their two nodes are already added.

Kruskal's Algorithm: Example

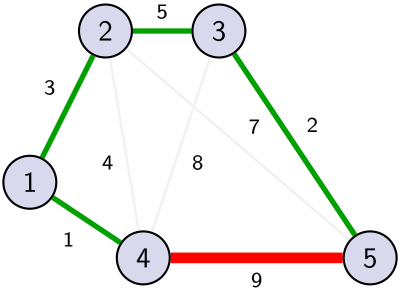
Incrementally add the shortest edge to the minimum spanning tree:



Because the tree is finished, the remaining edges should also be rejected when we see if their two nodes are already added.

Kruskal's Algorithm: Example

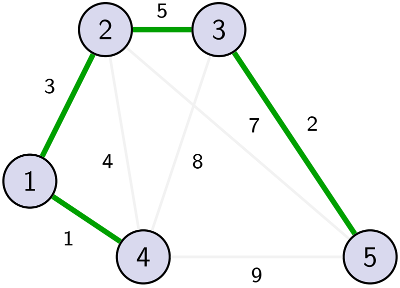
Incrementally add the shortest edge to the minimum spanning tree:



Because the tree is finished, the remaining edges should also be rejected when we see if their two nodes are already added.

Kruskal's Algorithm: Example

The set of edges that were added are then the minimum spanning tree.



Summary

Kruskal's algorithm is an example of a *greedy algorithm*.

- The minimum spanning tree problem is an *optimization problem*.
  - We're not just finding a spanning tree, we're finding the minimum one.
- A greedy algorithm solves an optimization problem by repeatedly picking the best possible addition to the solution.
  - Each graph edge is a possible addition to the spanning tree.
  - Pick the best one! Then... pick the next best one!
  - Don't pick any edges that can't be part of the solution.

Implementation Problem

- In a spanning tree (minimum or otherwise), there can't be more than one path between any two nodes (because it's a tree!)
- We can't add an edge to the minimum spanning tree that would cause the tree to not be a tree.
- Thus, we need to be able to easily tell when a node is already connected to another node.
- The slow way would be to traverse the current tree starting at one node, and see if we can find the other node using the edges already in the tree.
- We can do better than that.

Disjoint Sets

In general, *disjoint sets* are subsets of a collection of items where no item can be in more than one subset.
Examples:

- Cities can't be in more than one country.
- Sports teams can't be in more than one league.
- Nodes can't be in more than one connected component of a graph.

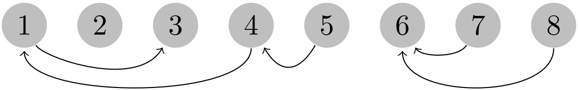
Disjoint Sets

Explanation:

- Each country/league/connected component represents a set of cities/teams/nodes.
- Each country/league/connected component is a subset of the full set of cities/teams/nodes.
- Each country/league/connected component subset is completely disjoint from the other subsets, because no city/team/node can be in more than one subset.

Implementing Disjoint Sets

We can use a graph to represent disjoint sets.



What are the disjoint sets in this collection? How are they represented?

To do:

- How do you find out which set an item is in?
- How do you join two sets together?

The `find` Operation

To find which set a node belongs in, walk the directed edges in the graph until you've run out of edges (i.e. arrive at the root of a tree).

```
Algorithm find(v):
// v: a vertex in a graph
// returns: the set that vertex belongs to

save our graph cursor position

set the vertex cursor to v
set the edge cursor to the first edge of v
while there is an edge item:
    move the vertex cursor to the terminal of the edge cursor
    move the edge cursor to the first edge of the vertex cursor

result <- the vertex cursor
reset the graph cursor position
return result
```

The set that the node is in is represented by returning one node from the set.
As long as the `find()` of two items is the same, then they are in the same set.

The Union Operation

To join two sets together, you have to add an edge from one set to the other.

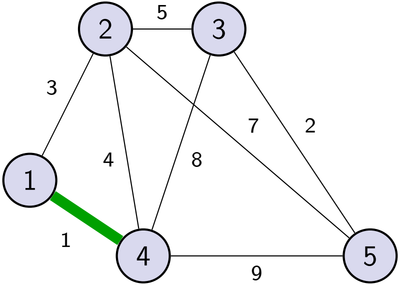
```
Algorithm union(v1, v2):
// v1: any vertex
// v2: any vertex
// postcondition: v1 and v2 are in the same set.

root_v1 <- find(v1)
root_v2 <- find(v2)

if (root_v1 != root_v2):
    add a directed edge from root_v2 to root_v1
```

Now, `find()` will find that any item that was in the set of either `v1` or `v2`, will be in the same set.

By using `find()` on the two items, we can make sure that we only ever draw an edge from a node that doesn't already have a node going out of it.

Kruskal's Algorithm: Example

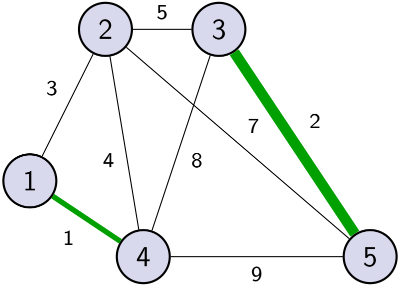
Let's repeat Kruskal's, and watch the disjoint set graph:



Are 1 and 4 connected?

What will `union(1,4)` look like?

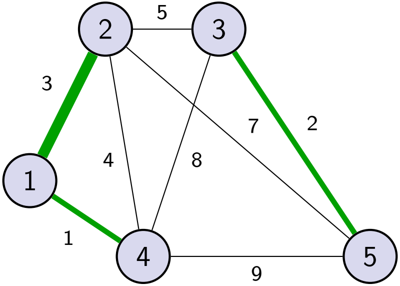
Kruskal's Algorithm: Example



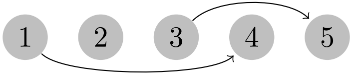
Are 3 and 5 connected?

What will `union(3,5)` look like?

Kruskal's Algorithm: Example

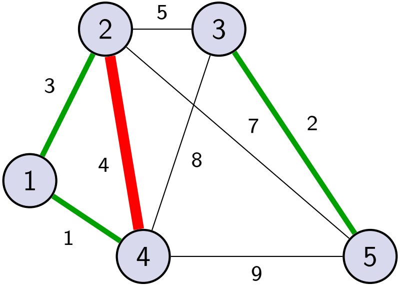


Are 1 and 2 connected?

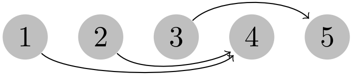


What will `union(1,2)` look like?

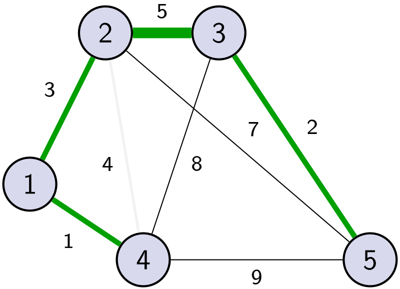
Kruskal's Algorithm: Example



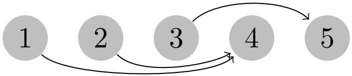
Are 2 and 4 connected?



Kruskal's Algorithm: Example

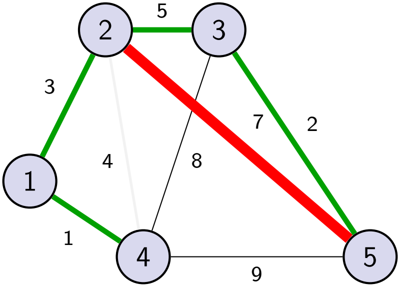


Are 2 and 3 connected?

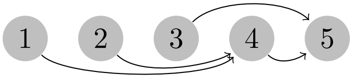


What will `union(2,3)` look like?

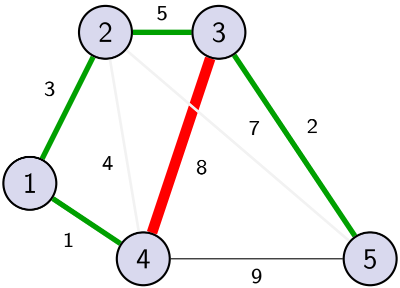
Kruskal's Algorithm: Example



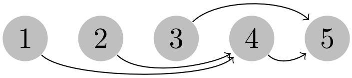
Are 2 and 5 connected?



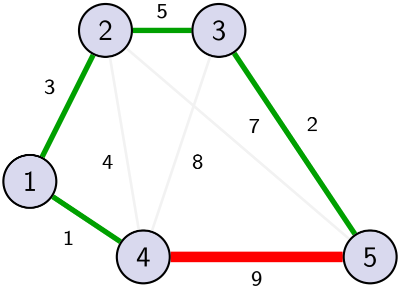
Kruskal's Algorithm: Example



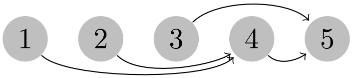
Are 3 and 4 connected?



Kruskal's Algorithm: Example



Are 4 and 5 connected?



Summary

- We used a graph representing a forest of trees (representing disjoint subsets) to solve another graph problem (minimum spanning tree).
- What is the time complexity of walking the unfinished minimum spanning tree to find if two nodes are connected?
- What is the time complexity of `find()`?
- What is the time complexity of `union()`?