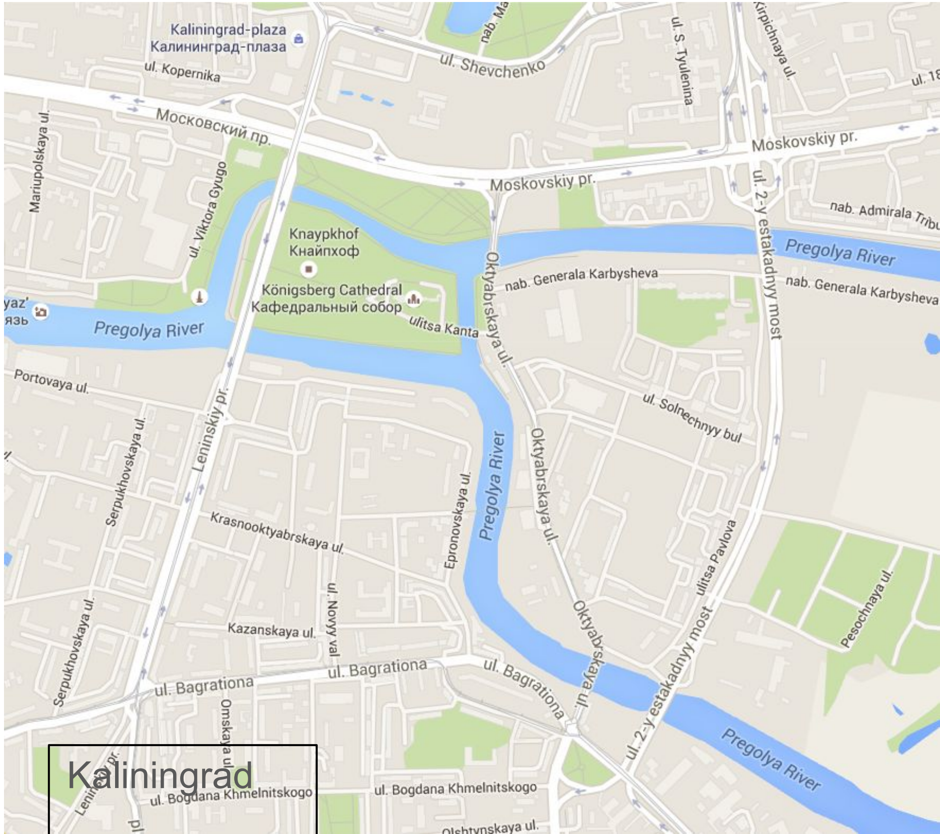
COMP20007 Design of Algorithms: Week 6

Week 6 tutorial – revision for MST (if you're in Friday pm tute feel free to attend an additional tute earlier in the week)

Week 6 workshop – catch up – finish any previous labs



Minimum Spanning Tree revisited: Kruskal's algorithm

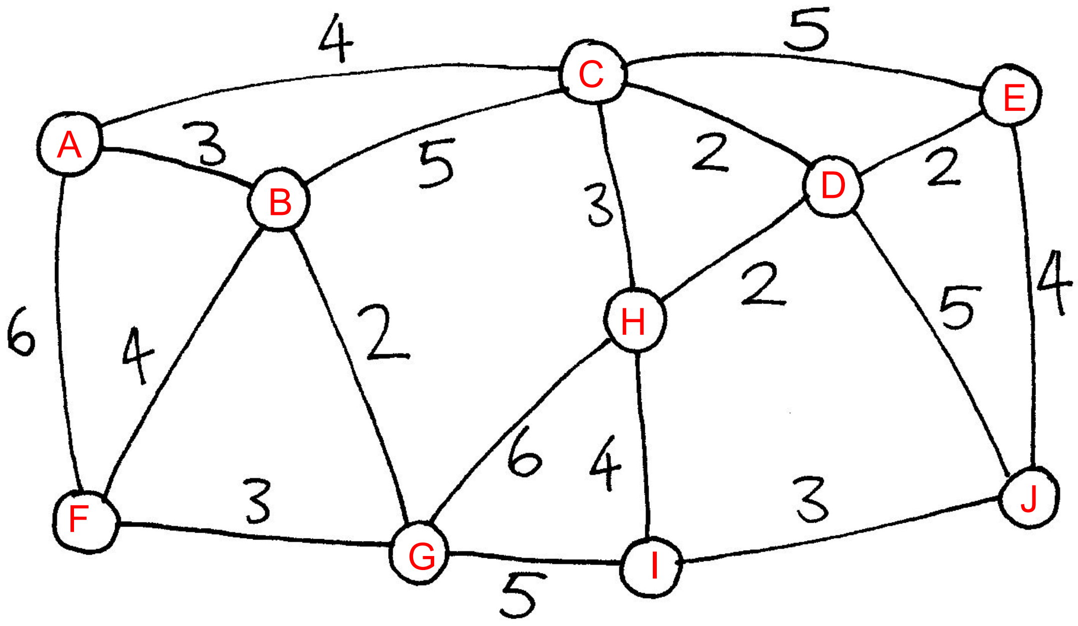
```
procedure kruskal(G, w)
Input:    A connected undirected graph G = (V, E) with edge weights w_e
Output:   A minimum spanning tree defined by the edges X

for all u ∈ V:
   makeset(u)

X = {}
Sort the edges E by weight
for all edges {u, v} ∈ E, in increasing order of weight:
   if find(u) ≠ find(v):
      add edge {u, v} to X
      union(u, v)
```

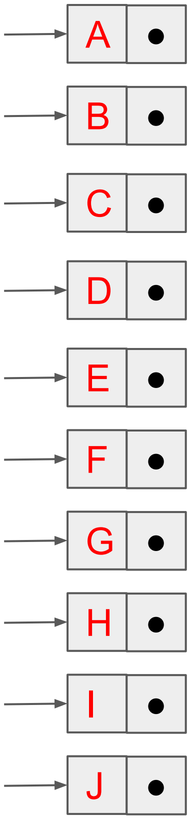
What was the property we checked last time, and why aren't we checking it this time? Why is this better?

Disjoint Set data structure (Union Find)

- Operations:
  makeset(x)
  find(x)
  union(x,y)
- Exploring possible implementations

Disjoint Set data structure (Union Find)





Source: csunplugged.org

Sample question

3.13. *Undirected vs. directed connectivity.*

(a) Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ whose removal leaves G connected. (*Hint:* Consider the DFS search tree for G.)

(b) Give an example of a strongly connected directed graph $G = (V, E)$ such that, for every $v \in V$, removing v from G leaves a directed graph that is not strongly connected.

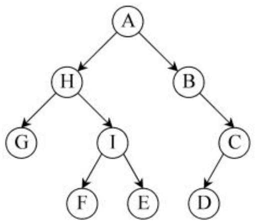
(c) In an undirected graph with 2 connected components it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.

(a) Consider the DFS tree of G starting at any vertex. If we remove a leaf (say v) from this tree, we still get a tree which is a connected subgraph of the graph obtained by removing v. Hence, the graph remains connected on removing v.

(b) A directed cycle. Removing any vertex from a cycle leaves a path which is not strongly connected.

(c) A graph consisting of two disjoint cycles. Each cycle is individually a strongly connected component. However, adding just one edge is not enough as it (at most) allows us to go from one component to another but not back.

Sample question



3.18. You are given a binary tree $T = (V, E)$ (in adjacency list format), along with a designated root node $r \in V$. Recall that u is said to be an *ancestor* of v in the rooted tree, if the path from r to v in T passes through u.

You wish to preprocess the tree so that queries of the form "is u an ancestor of v?" can be answered in constant time. The preprocessing itself should take linear time. How can this be done?

Do a DFS on the tree starting from r and store the previsit and postvisit times for each node. Since the given graph is a tree, and we started at the root, the DFS tree is the same as the given tree. Thus, u is an ancestor of v if and only if $\texttt{pre}(u) < \texttt{pre}(v) < \texttt{post}(v) < \texttt{post}(u)$.

Approaches to determining complexity

- Inspect the nested loop structure
- Write down the recurrence and solve it directly
- Write down the recurrence and apply the Master Theorem
- Consider how many times the nodes or edges are accessed