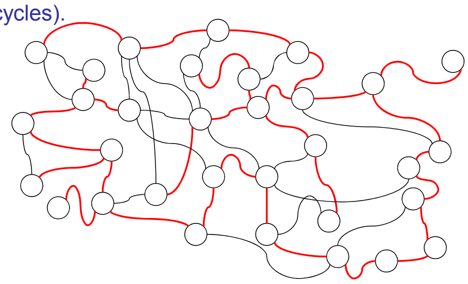
COMP 261 Lecture 11

Minimum Spanning Trees

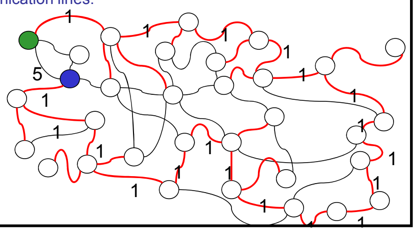
Spanning tree

- A spanning tree of a connected, undirected graph, is a subgraph that contains all the nodes but is a tree (no cycles).



Minimum spanning tree

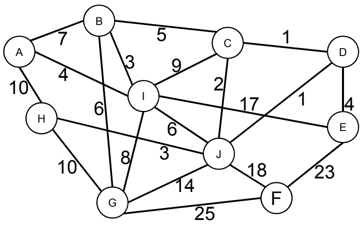
- Minimum spanning tree of a **weighted** connected, undirected graph is a spanning tree with total weight no greater than the total weight of any other spanning tree.
- Example use:
  - Find cheapest way to connect a set of towns/substations/cell towers with power/communication lines.
- May not be the shortest paths:



Prims Algorithm

Minimum Spanning Tree:

Idea: Grow the spanning tree from a seed node, always choosing the lowest weight edge to an unconnected node.



Prim's algorithm

- Given: a graph with weighted edges.
  Initialise *fringe* to have a dummy edge to the start node
  Repeat until all nodes visited:
    Choose from *fringe* a minimum weight edge to an unvisited node
    Add the edge to the spanning tree
    Visit the node at the other end of the edge,
    Add the unvisited neighbours of the node to the *fringe*

- Questions:
  - What data structures do you use to represent the graph?
  - How do you find the smallest edge from *fringe* efficiently?
  - How do you tell if an edge is out of or within *visited* ?
  - How do you represent the output tree?

Refining Prim's alg, explicit graphs

- Given: a graph with *N* nodes and *E* weighted edges
  - represented as an adjacency list, where
  - nodes have a *visited* flag and each edge has a *intree* flag

  Initialise: for each node, *node.visited* and *node.intree* ←false,
      count ← *0*
      fringe ← priority queue of <length, edge, node>

  priority
  small length ⇒ high priority

  Pick a node *n*, *fringe*.enque(〈 0, -, *n* 〉)
  Repeat until *count* = N or *fringe* is empty:
    <*length, edge* , *node*> ← *fringe*.dequeue
    If not *node.visited* then
      node.visited ← true,
      edge.intree ← true
      count ++
      for each *edge* out of *node* to *neighbour*
        if not *neighbour.visited* then
          add <*edge.length, edge, neighbour*> to *fringe*

Analysing Prim’s algorithm

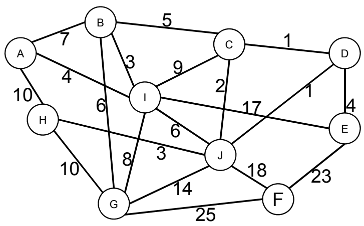
- Prim’s algorithm is a best first search (rather than depth first)
- What’s the cost of the algorithm?
  Assume: N nodes, E edges
- What kind of graph makes it fail?

Kruskal's algorithm

- Alternative algorithm for minimum spanning trees:
- Idea: Connect small trees together, always choosing the lowest weight edge.
- Given: a graph with weighted edges.
  Initialise *forest* to be a set of trees, each containing one node
  Repeat until *forest* contains only one tree:
    Choose a minimum weight edge that connects two trees in *forest*
    Add the edge to the spanning tree and combine the two trees

Graph Algorithms

Minimum Spanning Tree: Kruskal's Algorithm



- Questions:
  - How do you find the smallest edge efficiently?
  - How do you efficiently determine if an edge connects two trees or not?

Refining Kruskal's algorithm

- Given: a graph with *N* nodes and *E* weighted edges

  forest ← a set of *N* sets of nodes, each containing one node of the graph

  edges ← a priority queue of all the edges: ⟨n_1, n_2, length⟩

  priority: short edges first

  spanningTree ← an empty set of edges

  Repeat until *forest* contains only one tree or edges is empty:

  ⟨n_1, n_2, length⟩ ← dequeue(*edges*)

  If n_1 and n_2 are in different sets in *forest* **then**

  merge the two sets in *forest*

  ~~Add *edge*~~ to the *spanningTree*

  return *spanningTree*

  What's the cost?

- Implementing *forest* :
  - set of sets with two operations:
    - findSet(n_1) =?= findSet(n_2) = "find" the set that *n* is in
    - merge(s_1, s_2) = replace s_1, s_2 by their "union"

Greedy algorithms

- Both Prim's and Kruskal's algorithms are "greedy":
- Every time they make a decision (to add an edge to the spanning tree), they commit to it.
  - no backtracking
  - no sidetracks that get abandoned.
- Greedy algorithms work when
  - There is enough information at each point to make a correct decision
  - optimal solutions to sub-problems are always sub-solutions of the full problem.
- Greedy algorithms are generally fast.