Lecture 20: Weighted Graphs

COSC242: Algorithms and Data Structures

Brendan McCane

Department of Computer Science, University of Otago

Weighted Graphs

Graphs really become useful when we can give weights or costs to the edges. Weighted graphs can be used to model:

- maps with weights representing distances
- water networks with weights representing water capacity of pipes
- electrical circuits with weights representing resistance or maximum voltage or maximum current
- computer or phone networks with weights representing length of wires between nodes

One of the canonical applications for weighted graphs is finding the shortest path between two nodes. These algorithms are used in Google Maps for example.

We will focus on single-source shortest paths. These problems have a particular source vertex, s, and construct shortest paths to all other vertices in the graph (if they exist).

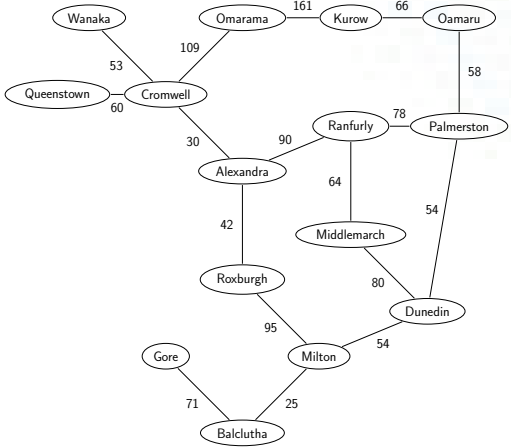
Dijkstra's Algorithm

Edsger Dijkstra is a famous foundational computer scientist who is best know for his shortest path algorithm (from 1959).

The basic idea is to use a breadth-first search and a priority queue keeping track of the shortest distances so far.

This is best illustrated by an example.

Dijkstra's Algorithm example



Dijkstra's Algorithm - Variables

First a description of the variables :

$G = (V, E)$	is the graph of interest.
$s \in V$	the source vertex
$w[e]$	is the weight of edge e
$adj[v]$	nodes adjacent to node v
$d[v]$	contains the best distance to the source computed so far for vertex v.
$\pi[v]$	contains a pointer back towards the source for the best path computed so far for vertex v.
Q	a priority queue of vertices ordered by d. If $d[v]$ changes for vertex v, then Q is automatically updated.

Dijkstra's Algorithm - Pseudo-code

```
1:  procedure DIJKSTRA(G,s,w)
2:      for each v ∈ V do                                   ▷ initialise
3:          d[v] ← ∞, π[v] ← Nil
4:      end for
5:      d[s] ← 0
6:      insert all v ∈ V into Q
7:      while Q is not empty do
8:          u ← dequeue(Q)
9:          for v ∈ adj[u] do
10:             if d[v] > d[u] + w(u, v) then               ▷ relax step
11:                 d[v] ← d[u] + w(u, v)
12:                 π[v] ← u
13:             end if
14:         end for
15:     end while
16: end procedure
```


Minimum Spanning Tree

Let's say you want to design a (wired) computer network for the university campus. One of the nodes in the network is a special node (the gateway) that connects the campus to the rest of the world. We don't really care how many internal nodes there are between a given node and the gateway, but the cable is really expensive, and we would like to minimise the total length of cable used while requiring that all nodes are connected to the gateway somehow. How can we minimise the total length of cable used?

For this problem, the shortest path algorithms might not give us the overall minimum length of cable. Why?

We need a different algorithm - a minimum spanning tree algorithm.

MST definition

A *spanning tree* of an undirected graph, G, is a connected acyclic subgraph of G that contains all the vertices in G. Note: all undirected acyclic graphs are trees. A graph may have more than one spanning tree.

A *minimum spanning tree* of a weighted undirected graph, G, is a spanning tree of G with minimum total weight. That is, the sum of the edge weights is a minimum.

MST algorithm

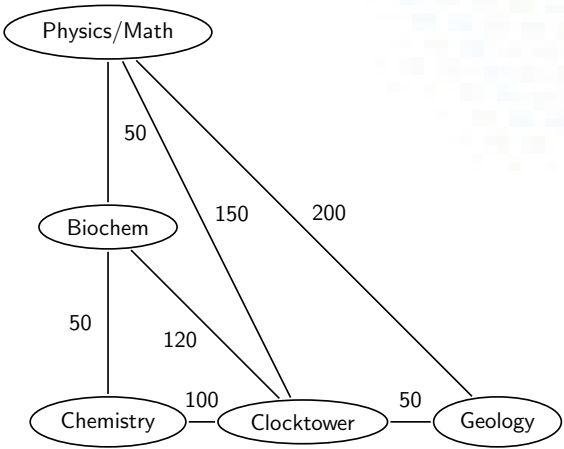
Prim's algorithm is a classic MST algorithm that works in a similar way to Dijkstra's algorithm.

Again we use a priority queue and add edges in a greedy fashion.

That is, the shortest edges are added to the current tree.

Again, let's look at an example:

MST example


Physics/Math
50
150
200
Biochem
50
120
100
50
Chemistry
Clocktower
Geology

Prim's Algorithm - Pseudo-code

```
1:  procedure MST-PRIM(G,r,w)
2:      for each v ∈ V do                                   ▷ initialise
3:          d[v] ← ∞, π[v] ← Nil
4:      end for
5:      d[r] ← 0
6:      insert all v ∈ V into Q
7:      while Q is not empty do
8:          u ← dequeue(Q)
9:          for v ∈ adj[u] do
10:             if v ∈ Q and w(u, v) < d[v] then
11:                 d[v] ← w(u, v)
12:                 π[v] ← u
13:             end if
14:         end for
15:     end while
16: end procedure
```


Relevant parts of the textbook

Dijkstra's algorithm is discussed in Section 24.3.

Prim's algorithm is discussed in Section 23.2.