Cosc 201
Algorithms and Data Structures
Lecture 14 (9/4/2025)
Minimum Spanning Trees

Brendan McCane
brendan.mccane@otago.ac.nz

UNIVERSITY OF OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

Dijkstra's algorithm again - directed graphs

q: an initally empty priority queue.
distances: shortest-path distance from *start* to *v*, initially ∞.
parent: the parent of *v* along the shortest path from *start* to *v*, initially all `null`.

```
1:  distances[start] ← 0
2:  q.add(start)
3:  while q is not empty do
4:      v ← q.remove()
5:      for e in v.edges do
6:          d ← distances[v] + e.weight
7:          if d < distances[e.v2] then
8:              distances[e.v2] ← d
9:              parent[e.v2] ← v
10:             q.add(e.v2, d)
11:         end if
12:     end for
13: end while
```

Prove that Dijkstra's algorithm computes the shortest distances

We only update distances when we add a vertex to the queue. No updates are made when we remove a vertex. We need to show that when we remove a vertex from the queue, the current distance must be the shortest.

By induction:

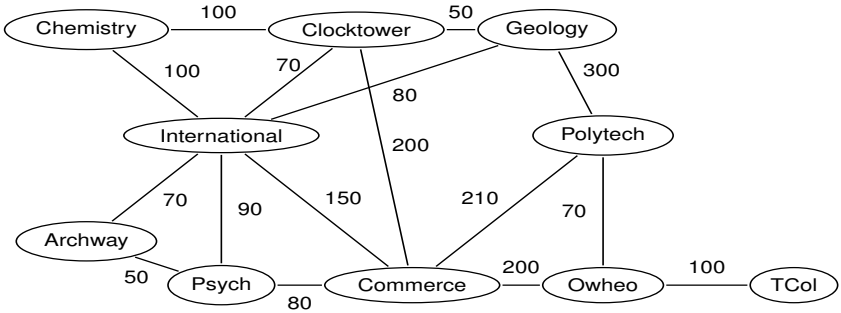
- Base case: the first vertex taken from the queue is *start* and $distances[start] = 0$. This is the smallest possible distance and therefore the base case is true.

Prove that Dijkstra's algorithm computes the shortest distances

- Inductive step:
  - Assume that the distances for all vertices removed from the queue are correct and are the shortest distances.
  - We remove vertex v from the queue.
  - There are three possible cases for v: we have never seen v before, we have seen v and it is in the queue, or we have seen v and already removed it from the queue.
  - In the first case, we add v to the queue with a distance that must be less than ∞ and therefore is the shortest distance so far.
  - In the second case, we have already seen v and it is in the queue. The distance we add to the queue must be less than the distance already in the queue (because of the if statement). When we add v to the queue again, it will be placed higher in the queue than the current position and will therefore be removed first and have a shorter distance.
  - In the third case, we have already seen v and removed it from the queue. By assumption, this is shorter than the current distance considered, and therefore v is not added to the queue again.
  - Therefore, by induction, when first removing a vertex from the queue, it must have the shortest distance recorded.

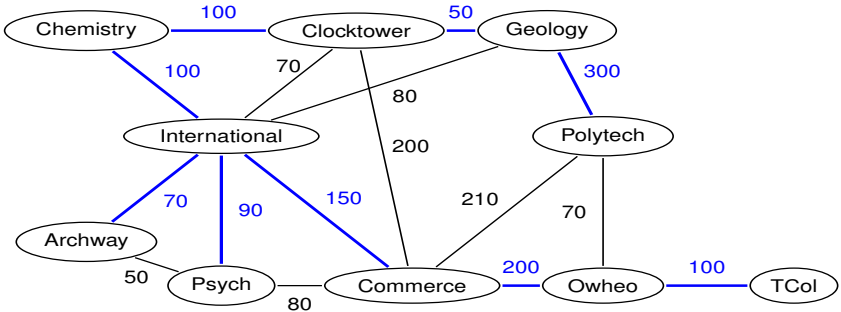
A problem

Imagine you are tasked with connecting the Dunedin campus buildings with a wired network. The wires are expensive, and you want to connect all the buildings with the minimum total length of wire. Here is a graph of the buildings and the distances between them. What is the connectivity pattern you choose?



A problem

Here is the shortest path tree starting from Chemistry (in blue).



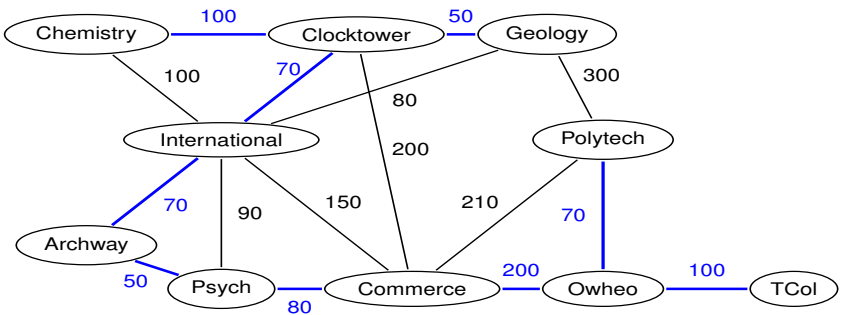
Total distance = 1160. Can we do better?

Prim's algorithm

q: an initally empty priority queue.
distances: minimum wt of any edge from *v* to the tree, initially ∞.
parent: the parent of *v* initially all `null`.

```
1:  q.add(start)
2:  while q is not empty do
3:      v ← q.remove()
4:      for e in v.edges do
5:          d ← e.weight
6:          if d < distances[e.v2] then
7:              distances[e.v2] ← d
8:              parent[e.v2] ← v
9:              q.add(e.v2, d)
10:         end if
11:     end for
12: end while
```

The minimum spanning tree



Total distance = 790.

Matters of efficiency

- How large a graph do you think we could find all the single-source shortest paths in in a reasonable time?
- **Scenario 1**: For any two vertices v and w there's an edge from v to w of random weight (and the weight from v to w versus w to v might differ). How large a graph can we analyse? There are $n \times (n-1)$ edges to consider.
- **Scenario 2**: The graph is an $n \times n$ grid in which every vertex is connected to its orthogonal neighbours by edges of random weight. There are approximately $2n^2$ edges. How large a graph can we analyse?