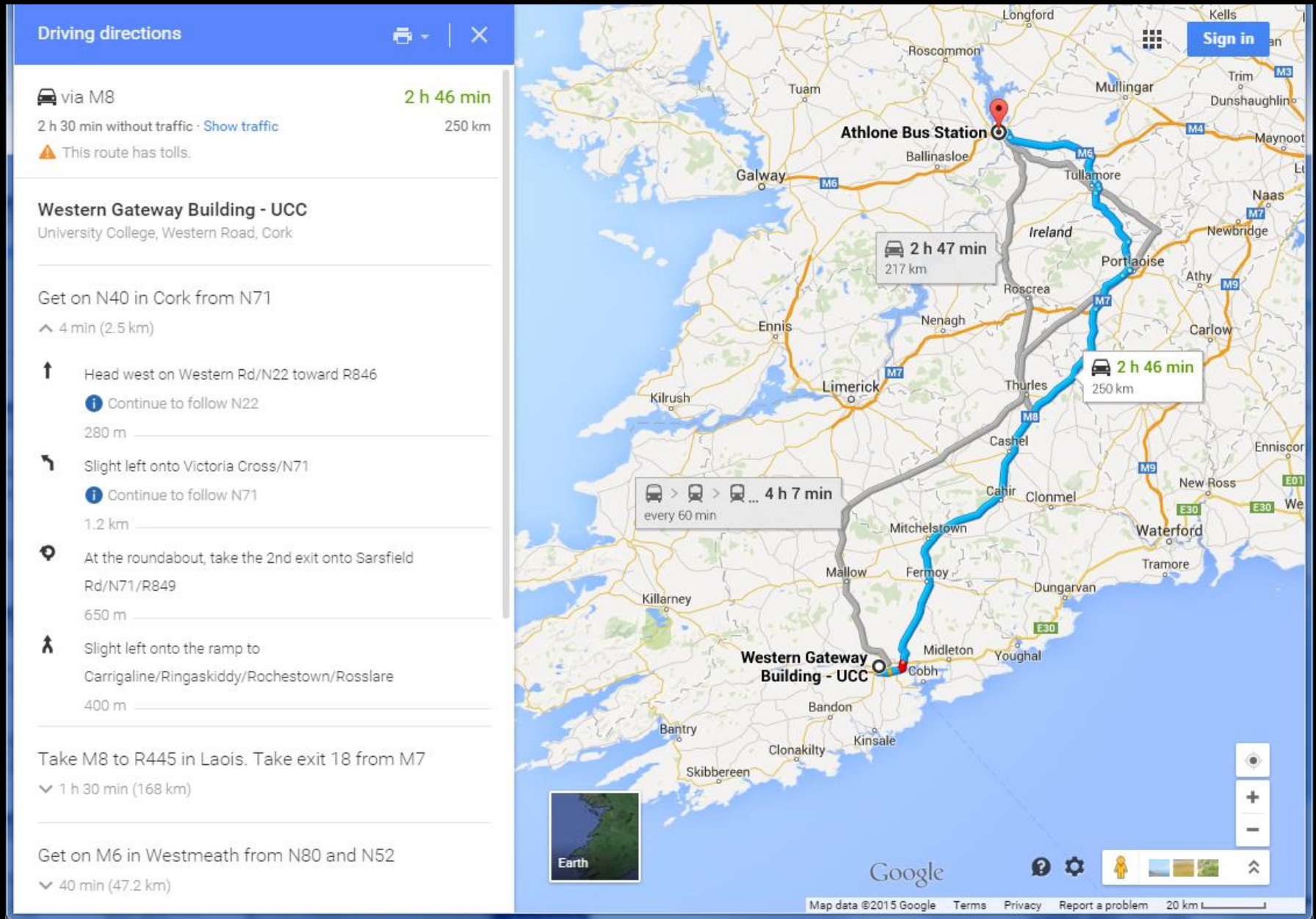
Shortest Paths in Weighted Graphs



A *path* from v_0 to v_k in a simple graph G is a sequence of vertices

$$<v_0, v_1, v_2, ..., v_k>$$

such that (v_i,v_{i+1}) is a oriented edge in G for each i from 0 to k-1.
If G is a *directed* graph, then the orientation of the edge must be the same as the direction.

The length of the path is the number of vertices - 1.

Equivalently, a path from v_0 to v_k in a graph is a sequence of oriented edges

$$<(v_0, v_1), (v_1, v_2), ..., (v_{k-2}, v_{k-1}), (v_{k-1}, v_k)>$$

The length of the path is the number of edges.

In a weighted graph, we have a function *w* which maps from the set of edges to some other set, assigning a *weight* to each edge.

$$w : E \rightarrow S$$

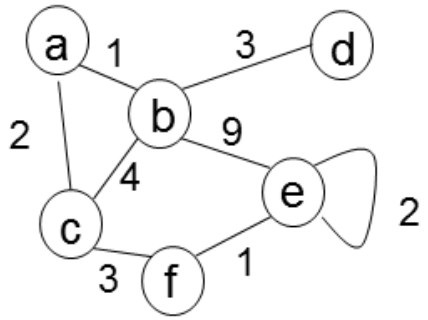
If the output set S is the integers, or other numerical set, then *w* can be viewed as the cost of travelling the edge, or distance across the edge, or time required to cross the edge, etc.

If we have a path made up of weighted edges where the weights are from some numerical set, then we can define the cost of the path.

If P is a path from v_0 to v_k, then the cost of the path is

$$\sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Path costs



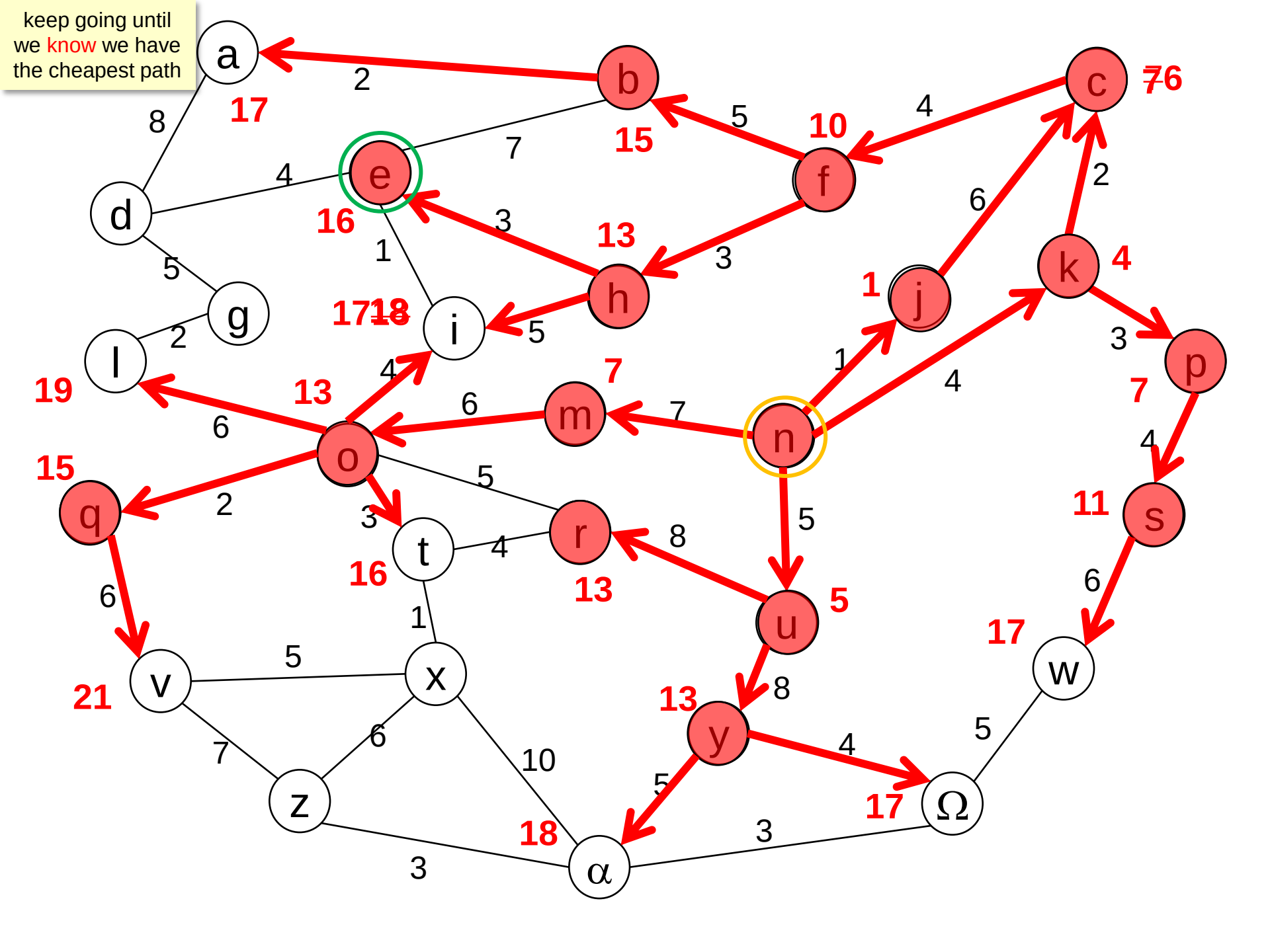
G = (V,E), where
V = {a,b,c,d,e,f}
E = {({a,b},1), ({a,c},2), ({b,c},4),
({b,d},3), ({b,e},9), ({c,f},3),
({e,e},2), ({e,f},1)}

Path <a,b,c,f>:	<(a,b),	(b,c),	(c,f)>	Total cost
Edge costs:	1	4	3	8

What is the cost of <f,e,e,b>?

Can we write an algorithm that computes the cheapest path between any pair of vertices?

keep going until we know we have the cheapest path

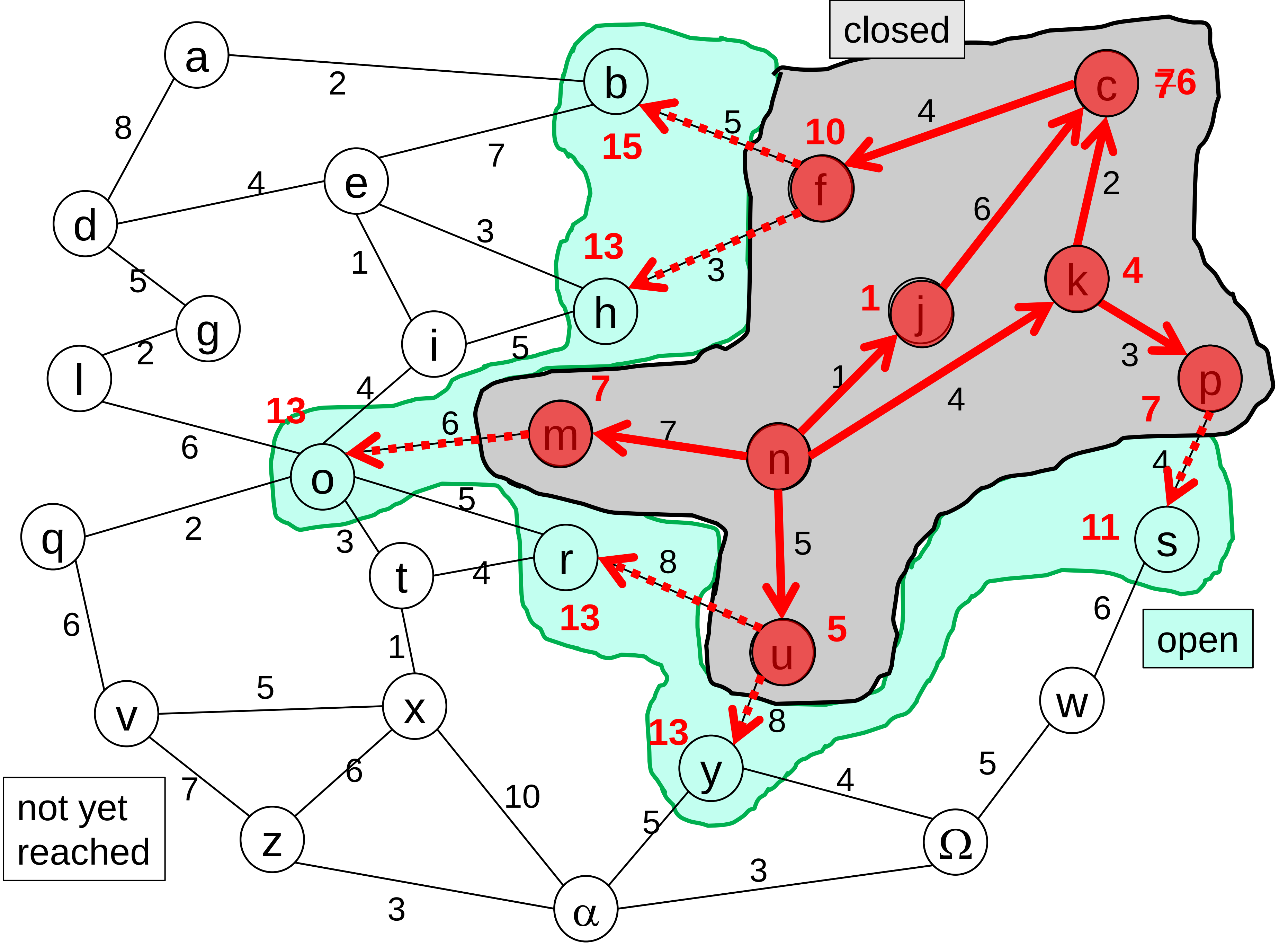


Adapt the algorithm from CS1113 so that we continue until we have found the shortest path from our start vertex to all other vertices that are reachable

```
while we still have open vertices
    pick the open vertex with lowest cost
    expand from that vertex to all neighbours not yet closed
    for each neighbour we reached
        add the cost of the edge to the neighbour onto the cost for the
                                        vertex to get a new path cost to the neighbour
        if the path cost is lower than the previous one, or neighbour is new
            update the neighbours path cost
    close the current vertex
```

How should we implement this in Python, using our ADT implementations?

What we need to do ...	Which data structures?
Maintain the original graph in an implementation that allows fast lookups of the neighbours of a given vertex	Adjacency map or adjacency list implementation
Final output should be a structure we can query which has, for each vertex, a cost and the previous vertex on its shortest path	Dictionary with vertices as keys, and values are (cost, predecessor) pairs
Maintain the path costs for all open vertices in a structure we can query, obtain the minimum cost vertex efficiently, and update with new costs efficiently	APQ for open vertices, key is the cost, value is the vertex Dictionary of locations for accessing elements in the APQ



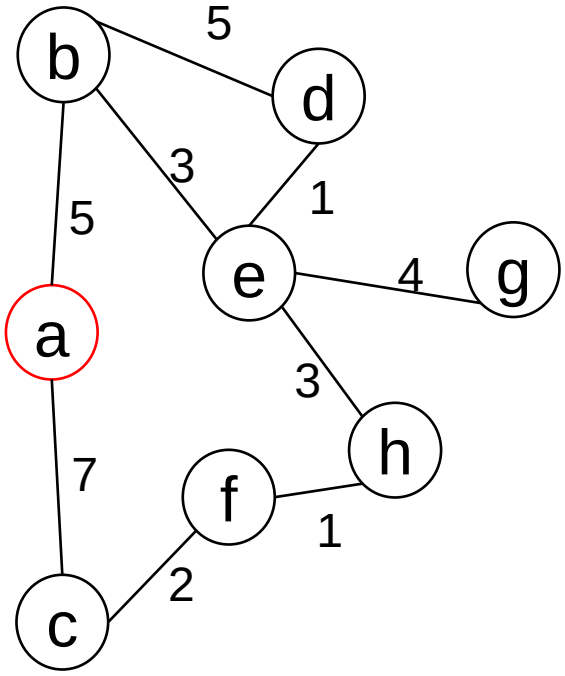
closed
open
not yet reached

dijkstra(*s*): find all shortest paths from *s* #pseudocode

open starts as an empty APQ
locs is an empty dictionary (keys are vertices, values are location in *open*)
closed starts as an empty dictionary
preds starts as a dictionary with value for *s* = None

add *s* with APQ key 0 to *open*, and add s: (returned elt) to *locs*
while *open* is not empty
    remove the min element *v* and its key from *open*
    remove the entry for *v* from *locs* and from *preds*
    add an entry for *v* with cost and predecessor into *closed*
    for each edge *e* from *v*
        w is the opposite vertex to *v* on *e*
        if *w* is not in *closed*
            newcost is *v*'s key plus *e*'s cost   #edit on 21/03/2017
            if *w* is not in *locs*
               add *w:v* to *preds*, add *newcost:w* to *open*,
                                                    add *w:(*returned elt from *open)* to *locs*
            else if *newcost* is better than *w*'s *oldcost*
                update *w:v* in *preds*, update w's cost in *open* to *newcost*
return *closed*

Run Dikstra to compute shortest paths from *a* to all other vertices



closed:

preds:

open:

locs:

Is Dijkstra's algorithm correct?

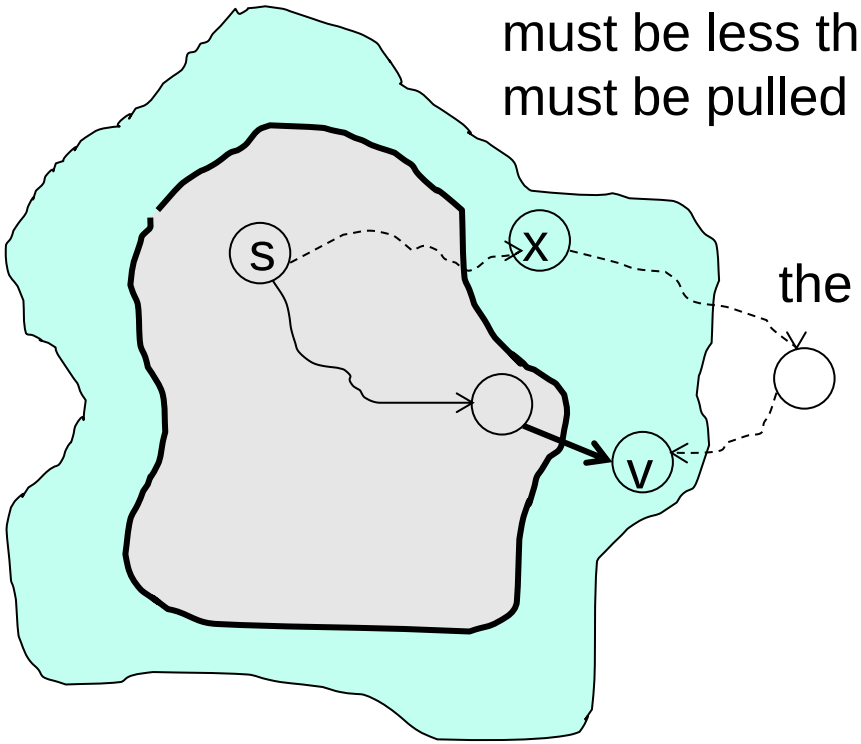
As long as the graph does not contain a negative weight, when a vertex v is added to 'closed', its path cost is the cost of the shortest path from s

Proof:

We can recreate the path by stepping backwards over the predecessors. Let S be this path, and all vertices in S must be in closed.
Suppose the path cost of S is not the lowest possible for v. Let P be a path with a lower cost. If all the vertices in P are in closed, then the last vertex before v in P would have led to v with P's cost when it was closed. Contradiction.
Therefore there must be a vertex in P not in closed. Let x be the first vertex in P not in closed. Since the cost of P is less than the cost of S, the cost of the path to x must be less than the cost of S. But then x should have been removed from 'open' before v. Contradiction.

Therefore the path to v must be the shortest path.

x is the first vertex on the 'shorter path' that is not already in *closed*. The path from s to x must be less than the path from s to v. So x must be pulled into closed before v. Contradiction.



the supposed shorter path

About to add v into closed.
Suppose there is another path to v that is shorter than this current one ...

Worst case running time of Dijkstra

Each vertex is added into *open* at most once, and is taken out of *open* at most once. So we do the outer loop *n* times.
Each time round the loop, we remove min item from the APQ.
We delete an entry from the *locs* dictionary (hash map)
Add an entry into the *closed* dictionary
We go round the inner loop degree(*v*) time, but amortised once per edge
Inside the inner loop: add to APQ or update
update the *predecessors* dictionary
So we have:

- n additions to the APQ
- n removals of min key item from APQ
- n deletions from the dictionary
- n additions to a dictionary
- m updates to a dictionary
- m updates (or additions) to the APQ

Dictionary updates are O(1) expected (and could be O(1) if we used a list of fixed length), so the complexity will depend on the APQ.

Dijkstra APQ: heap vs unsorted

Heap APQ		Unsorted APQ
O(log n)	n additions to the APQ	O(1)
O(log n)	n removals of min key item from APQ	O(n)
O(log n)	m updates (or additions) to the APQ	O(1)

so total is O(n log n + m log n)
which is O((n+m)log n).

If graph is dense: O(n^2 log n)
If graph is sparse: O(n log n)

so total is O(n + n^2 + m)
which is O(n^2 + m).

If graph is dense: O(n^2)
If graph is sparse: O(n^2)

If number of edges, m << n^2/log n, then prefer the heap APQ

If m >> n^2/log n, then prefer the unsorted list APQ

Next lecture

Further graph algorithms