1. Minimum cost walk

For these problems we consider *weighted* graphs (reminder: a weighted graph is a graph for which we have a number associated to each edge). ***Cost*** is often used when the goal is to find a walk or path between 2 vertices, to represent the cost of passing over an edge (e.g. for a graph representing a map and the edges the roads, the cost might mean the time it takes to go over that road, or fuel consumption, or distance etc.).

A common problem solved by graph theory is finding ***minimum cost walks*** (or paths or trails etc).

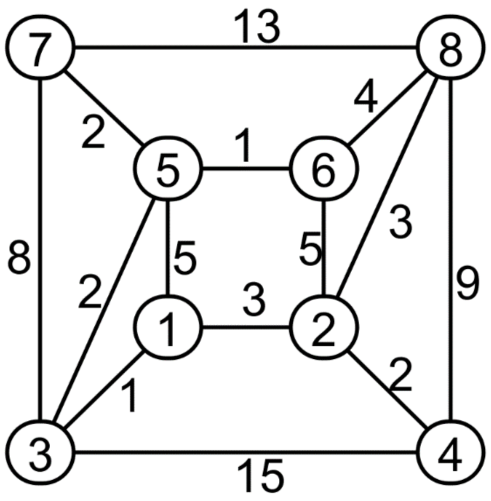

Figure 1

Let's consider the graph from figure 1. It is a weighted graph, with the number near each edge representing the weight – the cost.

It is clear that the shortest path (given by BFS ignoring the weights) is not the same as the minimum cost walk, for many pairs of vertices.

For example, the shortest path (as given by BFS ignoring the weights) between 7 and 4 is 7,3,4. But the minimum cost walk is 7,5,3,1,2,4.

The minimum cost walk is also called the shortest walk or shortest path (as the minimum cost walk is also a path, having duplicate edges or vertices would just increase the cost), so, if we have weights, shortest path refers usually to the minimum cost walk. Similarly, the total cost of the minimum cost walk is also called the **distance** between the 2 vertices, and the number of edges in the walk is called the **length**. While in a graph with no weights the distance and the length are the same, in a weighted graph the distance usually refers to the sum of costs and not the length. The first vertex is called the **starting** (denoted as s) vertex while the last one is called the **terminal** (denoted as t) vertex.

In our example, we have the minimum cost walk between 7 and 4: 7,5,3,1,2,4. The *distance* between 7 and 4 is 10 – the cost of the walk – while the length of the walk is 5 (we pass 5 edges). In this case s is 7 and t is 4.

In the following we will review some algorithms that find the minimum cost walk between 2 vertices. These algorithms are designed to find the minimum cost walks between the starting vertex and *all* other vertices. Sometimes an algorithm can be stopped early if we already found the walk between s and t, and we do not care about the other vertices.

2. <u>Dijkstra’s algorithm</u>

The idea in Dijkstra’s algorithm is to check all vertices, in the order of their distance to s and update the neighbors.

The result of this algorithm is 2 maps (or lists): first having the distance between each vertex and the starting vertex and the other having the previous vertex in the minimum cost walk between s and that vertex. Using the previous map, one can create the minimum cost walk between s and any vertex by always adding the previous until we find s.

```
Algorithm name: Dijkstra (classic Dijkstra)
Input:
      g: a weighted graph (with positive weights)
      s: a starting vertex in g
Output:
      distance: a map containing the distance from s to any vertex
      previous: a map containing the previous vertex in the minimum cost walk between
s and any vertex
Algorithm:
      Map distance
      Map previous
      Set vertices
      Set visited
      for v in graph.vertices():
            distance[v] = infinity #we initialize all distances with infinity
            previous[v] = -1
            vertices.add(v) #we create a copy of the vertices set
      end_for
      distance[s] = 0 #the distance from s to itself is 0
      while vertices is not empty:
            #returns the vertex with minimum distance (from the unvisited vertices)
            current = get_min_vertex(vertices, d)
            vertices.remove(current)
            visited.add(current)
            for n in g.get_neighbors(current):
                  if n not in visited:
                        dist_with_current = distance[current] + g.cost(current, n)
                        if distance[n] < dist_with_current:
                              distance[n] = dist_with_current
                              previous[n] = current
                        end_if
                  end_if
            end_for
      end_while

      return distance, previous
```

The complexity of this algorithm depends on how its different parts are implemented. We consider that all methods of `Map` and `Set` are implemented in Theta(1). Unless otherwise stated, from now on we will consider the graph implemented on a list of neighbors => `g.get_neighbor()` is in Theta(m/n). Finding the cost of an edge should be Theta(1).

The initialization is in Theta(n). What the algorithm does for every neighbor is constant time, so the entire for runs in the number of neighbors. At every step of the while we select a vertex then remove it from vertices, therefore the wile runs for every vertex. Since we check all the neighbours for all vertices, this is in total Theta(n+m) – that is the `for n in g.get_neighbors` runs in Theta(n+m) in total. The only function left is `get_min_vertex(vertices, d)`.

a) If this is implemented naively, then the run time is O(n) (not theta because vertices gets smaller at every iteration). So if we call this for n times, then, in total this will run n^2 times. So, overall the algorithm will have $O(n^2 + m)$.
b) This can be implemented with a priority queue so that it runs in O(log(n)). In this case it will run in total n*log(n) times. So, overall it will be O(n*log(n) + m).

In order to recreate the walk based on the `previous` map we can use the following algorithm:

```
Algorithm name: GetWalk
Input:
        s, t: starting and terminal vertex for the walk that we want
        previous: the previous map as returned by Dijkstra’s algorithm
Output:
        walk: a list containing the vertices in the minimum cost walk from s to t.
Algorithm:
        List walk
        walk.append(t)
        last_prev = previous[walk[-1]] #the previous of the last element in the walk
        while last_prev ≠ -1:
                walk.append(last_prev)
        end_while
        walk.reverse() #reverse list so that the first element is s and the last is t

        return walk
```

This algorithm runs in O(n), more precisely in Theta(walk_length).

Observe that in Dijkstra’s algorithm each vertex is visited once. That means that once a vertex is visited, we found the minimum distance of the walk from s to that vertex. So, if we only want to find the walk from s to t, we could stop early once we visited t.

Please also note that the algorithm only works with positive weights – the edges should not have negative costs. But why would we have negative costs in the first place? Some examples:
- Fuel consumption: for electric cars going downhill regenerates fuel => negative fuel consumption
- In a game you have a state of the game. Some actions decrease your Score while some others increase your Score => cost of actions can be positive or negative (if we consider the graph as having the possible states as vertices and actions that change between states as edges).
- Neural networks can have negative wights to show a reverse relation
- etc.

Why doesn’t this algorithm work for negative weights? Because it only visits each vertex once. If we might find a better way afterwards because of a negative edge, the distance will not be updated.

3. Uniform Cost Search (UCS) - alternative Dijkstra

Uniform cost search is a variant of Best First Search (next lecture) on one hand but, on the other, is also an “alternative” Dijkstra’s algorithm. It is sometimes presented as Dijkstra’s algorithm itself. This is because, if there are no negative cost edges, the graph is visited exactly as in Dijkstra’s algorithm, making them effectively equivalent in this case. But if there *are* negative cost edges, UCS still manages to find the minimum cost walk (given that there are no negative cost cycles, we’ll se about that later).

The main element of UCS is a priority queue, which always gives the element with the smallest distance. It starts with s then for every popped element adds its neighbors that need to be updated. It runs until no neighbors need to be updated anymore.

As with the classic Dijkstra algorithm it computes all the minimum cost walks from s to all the other vertices. It generates the same distance and previous maps.

```
Algorithm name: Uniform Cost Search (UCS –alternative Dijkstra)
Input:
    g: a weighted graph (must be directed if there are negative cost edges)
    s: a starting vertex in g
Output:
    distance: a map containing the distance from s to any vertex
    previous: a map containing the previous vertex in the minimum cost walk between
s and any vertex
Algorithm:
    Map distance
    Map previous
    PriorityQueue pq #The priority is the minimum distance, where the distance of a
                      vertex should be the value associated to that vertex in the
                      ‘distance’ map
    for v in graph.vertices(): #same initialization
        distance[v] = infinity
        previous[v] = -1
    end_for
    distance[s] = 0
    pq.push(s)
    while pq is not empty:
        current = pq.pop()
        for n in g.get_neighbors(current):
            dist_with_current = distance[current] + g.cost(current, n)
            if distance[n] < dist_with_current:
                distance[n] = dist_with_current
                previous[n] = current
                pq.push(n)
            end_if
        end_for
    end_while

    return distance, previous
```

The complexity analysis is slightly different from Dijkstra’s algorithms, since the main while is not a fix number of iterations. Still, in the case of no negative cost cycles, it must be the same as Dijkstra’s.

In that case, we can see that each vertex will be added to the queue once: no matter if we still compare it afterwards with `dist_with_current` it will never be true as there can be no other better walk. So each vertex will be added once, but all vertices must be add at least once, since they all start at infinity, then they will be updated with the real distance and added to the priority queue. So each vertex is added to the queue exactly once, then it is exactly as Dijkstra's. If we consider that the queue is implemented efficiently with log(n) push and pop, then the complexity is the same O(n*log(n) + m).
If we have negative cost edges, things are a bit more complicated. In the very worst case, if the graph is dense enough, for every second new edge that we add, it becomes an edge that must be included in the paths towards all the other (already visited) vertices, thus having to add all other vertices again in the queue. That means add all vertices for every second edge, and when we add all vertices, we visit their neighbors, so we visit again all the edges that were visited until now for every second new edge visited. So, in the very worst case this is O(m^2*log(n) + n). But this is a very specific case, usually we would have to add at most a few nodes for every new visited edge, so the average complexity is O(m*log(n) +n).

If there are no negative cost edges, we can stop the algorithm early when we find the terminal vertex if needed, as in Dijkstra's case. However, if there are negative cost cycles we cannot stop early since the first time we visit the terminal vertex we might not have found the optimal walk yet.

Please note that this algorithm does **not** work if there are negative cost cycles (a cycle where the sum of the cost of the edges is a negative number). First of all, the minimum cost *walk* does not make sense if there are negative cost cycles, since we would need to infinitely go around the cycle to minimize the cost of the walk. Indeed, that is what happens with the UCS algorithm if there are negative cycles: it gets stuck in an infinite loop.
The minimum cost path, on the other hand, makes sense even if there are negative cost cycles, since in a path we cannot go over the same vertices twice. However, this is a much harder problem (see lectures 8 and 9) and UCS cannot guarantee to find a path, nor is there an easy way to modify UCS to not get stuck in a negative cost cycle.

Also note that UCS and other algorithms that find minimum cost walk in the presence of negative cost edges without negative cost cycles only work for directed graphs. Why? Well, a negative cost undirected edge is basically a negative cost cycle itself.