Algorithms and Computational Thinking
Autumn 2017

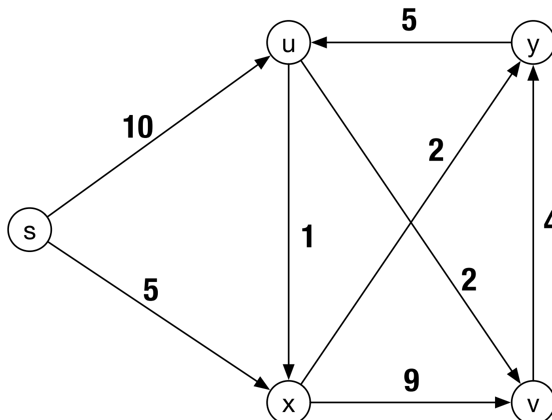
Tuesday, 7th November 2017

Exercise 1 - Graph Algorithms : Dijkstra's Shortest path Algorithm

In this exercise, you will write an algorithm and implement it in Python, Scala and Swift to find the shortest route in an unidirectional weighted graph. For this, you will implement Dijkstra's shortest path algorithm.

This algorithm is useful in several real world problems, from guiding cars via GPS to finding efficient routes for packets in networking. In the following, assume that you are solving a problem which involves finding shortest route between two cities in Europe.

Consider the following graph, where the nodes denote the cities, the edges connecting them denote the roads and the numbers on each edge represent the distance between the cities (weight or cost).



The Dijkstra Algorithm works as below :

1. Assign to every node a tentative distance value : set it to zero for some initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes as unvisited. Create a set of all the unvisited nodes called the unvisited set.

3. For the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be $6 + 2 = 8$. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.
5. If the destination node has been marked as visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm is completed.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

The Pseudocode using priority queue is given below : A min-priority queue is an abstract data type that provides 3 basic operations, add_with_priority(), decrease_priority() and extract_min(). Using such a data structure can lead to faster computing times than using a basic queue. We have provided you with the implementation of priority queue, you just need to import it in your code and use the functions.

```
1  function Dijkstra(Graph, source):
2      dist[source] ← 0                                  // Initialization
3
4      create vertex set Q
5
6      for each vertex v in Graph:
7          if v ≠ source
8              dist[v] ← INFINITY                        // Unknown distance from source to v
9              prev[v] ← UNDEFINED                       // Predecessor of v
10
11         Q.add_with_priority(v, dist[v])
12
13
14     while Q is not empty:                            // The main loop
15         u ← Q.extract_min()                           // Remove and return best vertex
16         for each neighbor v of u:                     // only v that is still in Q
17             alt = dist[u] + length(u, v)
18             if alt < dist[v]
19                 dist[v] ← alt
20                 prev[v] ← u
21                 Q.decrease_priority(v, alt)
22
23     return dist[], prev[]
```