Discrete Optimisation

Assignment 1 report

Introduction

Task: Find the shortest path between given nodes in five different graphs.

The solution for graph task1_test<i>.txt can be found in task1_test<i>_solution.txt. The format is a list of minimum lengths corresponding to node pairs requested.

Three algorithms have been used

- Dijkstra's algorithm: Used for tests 1, 2, and 3. This is the standard algorithm for the shortest path problem between two nodes. Complexity: O((E+V)log V)
- Breadth-first search: Used for test 5. As this graph in question is a constant-weighted graph we can used BFS, more efficient than Dijkstra's algorithm. Complexity: O(E+V)
- Bellman-Ford's algorithm: Used for test 4. This algorithm allows us to take into account negative weights on the graph, which Dijkstra's algorithm cannot. Complexity: O(EV)

Here are the running times for the 5 tests:

Test	1	2	3	4	5
Run time (s)	0.331	3.606	98.713	2042.488	147.535

Algorithm Explanations:

Dijkstra'algorithm:

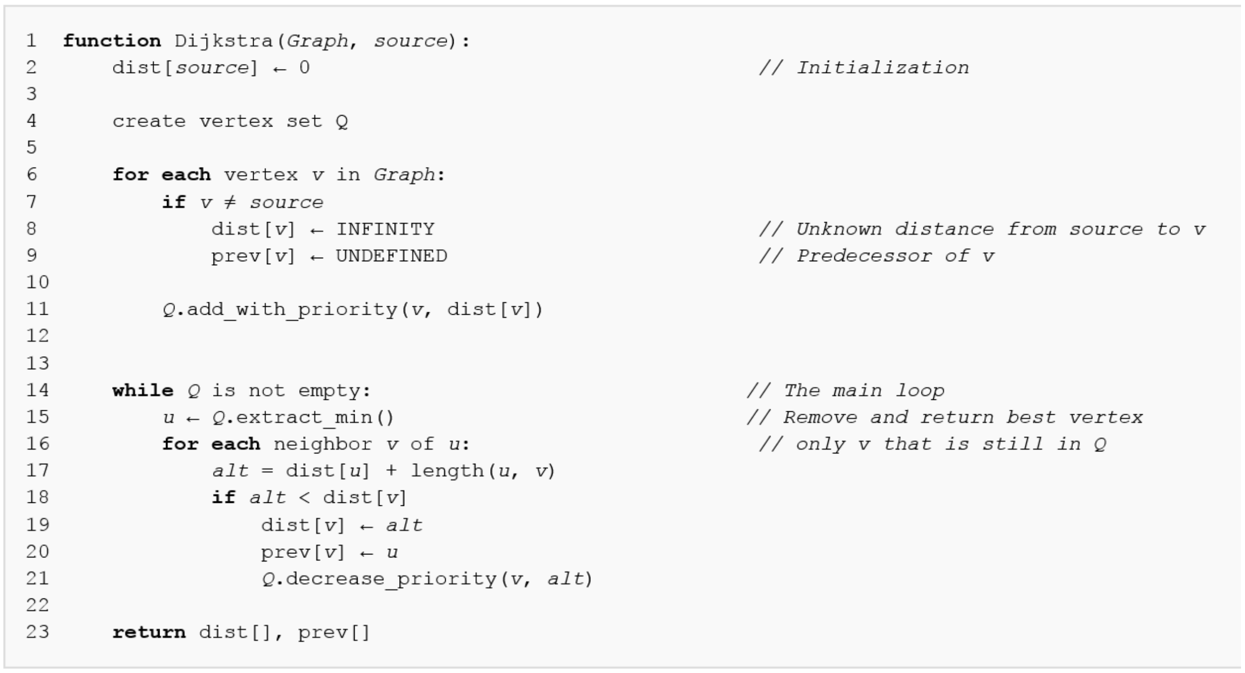
A min heap is used to optimise the run time. This allows us to select the node with the minimum distance from the source that is still in the heap in log time instead of linear time.

The implementation used is heapq: https://docs.python.org/2/library/heapq.html

Three extra functions have been added to let us use the heap as a priority queue. To use this queue, we add nodes where the priorities are the distance from the source. When a distance needs to be updated, instead of deleting the node from the heap, we add another node and use the dictionary entry finder to map the node to this new distance. We then mark the previous distance as removed. This allows to avoid the complexity of restructuring the heap to delete nodes, but requires additional memory.

Note also that Dijkstra algorithm can only be run on graphs with positive weights. Therefore we initially check to make sure all weights are positive before running the algorithm.

The algorithm is implemented according the following pseudo code:



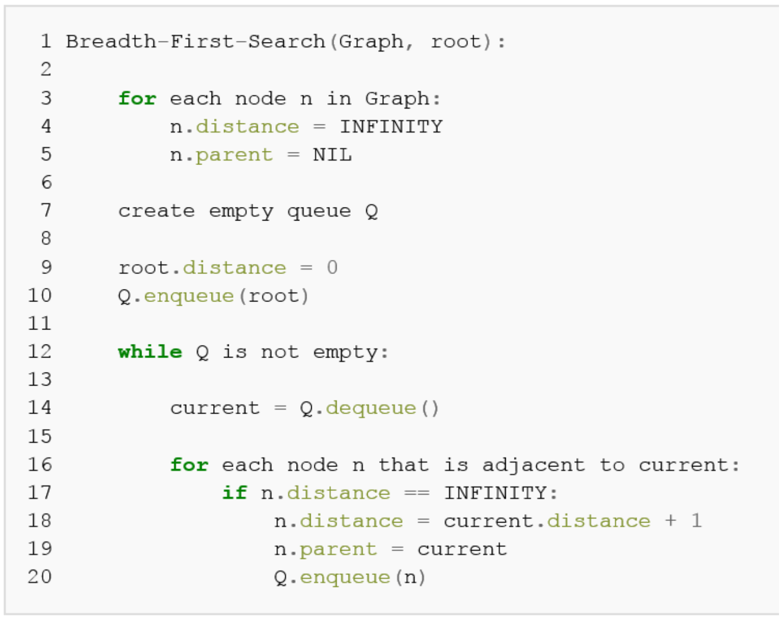
```
function Dijkstra(Graph, source):
    dist[source] ← 0                                    // Initialization

    create vertex set Q

    for each vertex v in Graph:
        if v ≠ source
            dist[v] ← INFINITY                          // Unknown distance from source to v
            prev[v] ← UNDEFINED                         // Predecessor of v

        Q.add_with_priority(v, dist[v])


    while Q is not empty:                              // The main loop
        u ← Q.extract_min()                            // Remove and return best vertex
        for each neighbor v of u:                       // only v that is still in Q
            alt = dist[u] + length(u, v)
            if alt < dist[v]
                dist[v] ← alt
                prev[v] ← u
                Q.decrease_priority(v, alt)

    return dist[], prev[]
```

Figure 1: Dijkstra's algorithm (source: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

Breadth first search

This search only works when all the weights are of the same value. We therefore check this before proceeding.

The algorithm is implemented according to the following code:



```
Breadth-First-Search(Graph, root):

    for each node n in Graph:
        n.distance = INFINITY
        n.parent = NIL

    create empty queue Q

    root.distance = 0
    Q.enqueue(root)

    while Q is not empty:

        current = Q.dequeue()

        for each node n that is adjacent to current:
            if n.distance == INFINITY:
                n.distance = current.distance + 1
                n.parent = current
                Q.enqueue(n)
```

Figure 2: Breadth first search (source: https://en.wikipedia.org/wiki/Breadth-first_search)

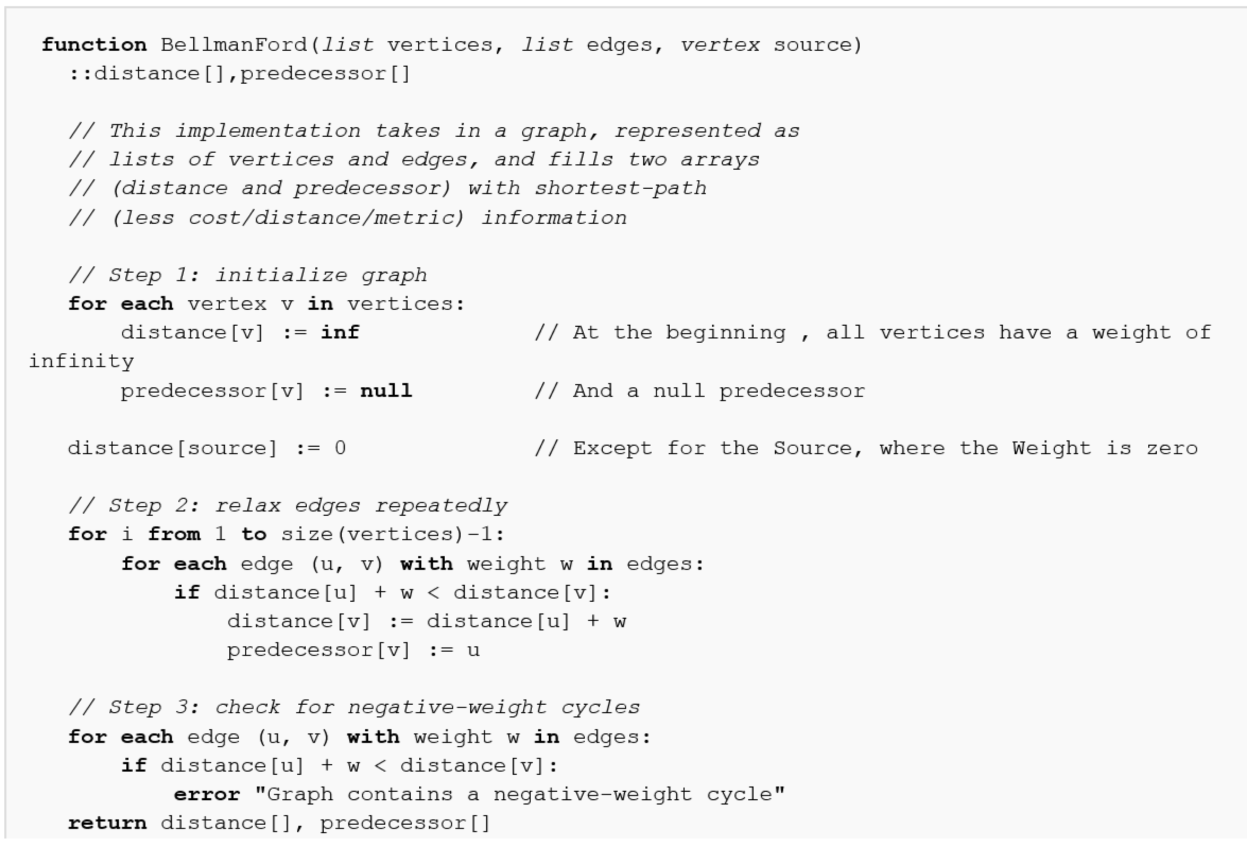
For the implementation of the queue, the deque library is used.

Bellman-Ford

This algorithm can accommodate negative weights. However it cannot deal with negative cycles (a shortest path cannot exist because loop around the negative cycle would reduce the distance infinitely).

The algorithm successively relaxes the edges, by finding on each iteration i the shortest path by using i nodes. If a shortest path can be found on the nth iteration (where n is the number of nodes), then a negative cycle exists.

The algorithm is based on the following pseudo code:



```
 function BellmanFord(list vertices, list edges, vertex source)
   ::distance[],predecessor[]

   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       distance[v] := inf             // At the beginning , all vertices have a weight of
infinity
       predecessor[v] := null         // And a null predecessor

   distance[source] := 0              // Except for the Source, where the Weight is zero

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"
   return distance[], predecessor[]
```

Figure 3: Bellman Ford algorithm (source: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)