CMSC 123: Data Structures

1st Semester AY 2019-2020

Prepared by: CE Poserio

[INLAB] Exercise 09: Single Source Shortest Path

Shortest Path for Unweighted Graphs

The **length of a path** is the number of edges in the path. The **shortest path** from a start (aka *source*) vertex, `s`, to an end vertex, `t`, is a path with the minimum length. Note that there can be multiple shortest paths from `s` to `t` in a graph. The length of the shortest path from `s` to `t` is considered as the **distance** from `s` to `t`. The shortest path problem involves finding the shortest path from a vertex to another vertex. To do this, the easier problem must be solved first — finding the distance from vertex `s` to `t`.

Reduction to Finding Path Distance

Finding the distance from a source vertex v_s to another vertex v_t can be solved using BFS. In BFS, paths from v_s to other vertices $v_i, i \in [0, |V|)$ in the graph are created. During the first iteration, paths of length one are created, which are the paths with only edge $P = \{[(v_s, v_j)] \mid j \in [0, |V|), v_j \in \text{adjacent}(v_s), \text{ and } s \neq j\}$. The second iteration expands these paths to a path of length two; for each path in P, a new edge from v_j to a new vertex v_k is added, where $v_k \in \text{adjacent}(v_j)$ and $j \neq k$. This process of expanding the paths is done until all vertices are visited.

An iterative version of the BFS algorithm is shown below:

```
let n = number of vertices
let visited = array of size n, initialized to FALSE
let frontier = an empty queue
let s = source vertex

frontier.enqueue(s)
visited[s] = TRUE

while(!frontier.isEmpty()):
    u = frontier.dequeue()
    // print u
    for v in adjacent(u):
        if !visited[v]:
            frontier.enqueue(v)
            visited[v] = TRUE
```

The queue `frontier` contains the last vertex of each path, which are then expanded in the next iterations. The algorithm is modified below to compute distances from v_s to other vertices v_j.

```
...
let s = source vertex
let distance = array of size n, initialized as INFINITY
distance[s] = 0
frontier.enqueue(s)
...
```

First, an integer array `distance` is created. This will contain the distance of each vertex from the source vertex. The values of this array are initialized to ∞ (or any large number), representing vertices are far away from the source; distances are unknown. v_s is 0 steps away from itself, thus, `distance[s] = 0`. Next we update the distances.

```
for v in adjacent(u):
    if !visited[v]:
        frontier.enqueue(v)
        distance[v] = distance[u] + 1
        visited[v] = TRUE
```

During the expansion of vertex u, the distances of its neighbors are updated as `distance[u]+1`. After above modifications, the algorithm now computes the distances from v_s to all other vertices in the graph. However, we are interested in the *path* and not just the distance.

Finding the Actual Paths

To find the actual path, we need to maintain a `parent` array that tracks which vertex u was expanded when vertex v was visited.

```
...
let distance = array of size n, initialized as INFINITY
let parent = array of size n, initialized as -1

distance[s] = 0
...
```

During the expansion, `parent` array is updated as follows:

```
for v in adjacent(u):
    if !visited[v]:
        frontier.enqueue(v)
        distance[v] = distance[u] + 1
        parent[v] = u
        visited[v] = TRUE
```

The algorithm now tracks (1) the distance of each vertex from v_s and (2) the `parent` of each vertex. The parent of v_s will be left as -1; the source has no parent. The path can now be generated by following the *ancestors* of the target vertex, v_t. Since, we are only interested in the path to v_t, the algorithm can be terminated early once expanded to vertex v_t; no need to discover all vertices. However, at worst case, the path to v_t need to pass through $O(|V|)$ vertices, thus, terminating early is rarely helpful.

Shortest Path for Weighted Graphs

The length of a path in weighted[1] graphs is the sum of the weights of the edges in the path. Finding the shortest path in a weighted graph uses the similar steps done in unweighted graphs. Distance is still the path with the minimum length. Since finding the actual path is very easy by using the `parent` array, the main focus is still in finding the distances from vertex v_s to other vertices.

Finding the distances

Generally, BFS is still used to update the distances from the source by expanding the *frontiers*. However, edges are weighted, thus, expansion of vertex must be *selective*; expand one at a time and expand only unexpanded vertices with the smallest distance from the source. Moreover, a vertex u is only expanded to vertex v if `distance[v] <= distance[u] + weight(edge(u,v))`. Also, the first time a vertex is visited does not ensure that the vertex expanded from a shortest path; thus, the distance must be allowed to be updated multiple times. The algorithm begins by initializing needed arrays and other values:

[1] We consider here non-negative edge weights.

```
let n = number of vertices
let visited = array of size n, initialized to FALSE
let distance = array of size n, initialized to INFINITY
let s = source vertex
let frontier = an empty queue

distance[s] = 0
frontier.enqueue(s)
```

Then, other vertices are visited using the following algorithm:

```
while !frontier.isEmpty():

    find vertex u with minimum distance from frontier

    if !visited[u]: // expand only unvisited vertices
        visited[u] = TRUE

        for v in adjacent(u):
            // allow re-updating distances
            alternate_distance = distance[u] + weight(edge(u,v))
            if alternate_distance <= distance[v]:
                // go to v from u
                // distance is incremented by the weight of the edge
                distance[v] = alternate_distance
                frontier.enqueue(v)
```

Finding the actual paths can be done the same way in unweighted graphs – by incorporating a `parent` array. As a whole, the algorithm described for finding shortest paths in non-negatively weighted graphs is called **Dijkstra's Algorithm**.

Tasks

Given a weighted *digraph*, some source vertex s, and target vertex t, find and display the shortest path from s to t and also, display the length of the path.

Input

The input consists of a single weighted digraph, single source vertex s, and multiple target vertex t. The input begins with a line containing four non-negative integers, $1 \leq n \leq 10000$, $0 \leq m \leq 30000$, $1 \leq q \leq 100$, and $1 \leq s \leq n$, separated by whitespace, where n is the number of vertices, m is the number of edges, q is the number of queries (target vertices), and s is the source vertex. This line is followed by m lines; each of these lines contains three values $1 \leq u, v \leq n$, and $1 \leq w \leq 100$, which indicates that there is a directed edge from u to v with weight w. Then follow q lines of queries, each consisting of a single integer t, which asks to find the shortest path from s to t and its length.

Note: Vertices are named from 1 to n.

Output

For each query, display two lines of output. The first line consists of space separated integers, the path from s to t. The second line consist of a single integer, the path length of the shortest path. If t is unreachable from s, ouput "Impossible" (without quotes) for the first line and -1 for the second line.

Sample Run

Here is a sample input (also `test5.in`):

Sample Input
12 13 4 5
1 2 1
1 3 3
1 4 3
2 5 2
2 6 5
4 7 1
4 8 2
5 9 6
5 10 1
7 11 10
7 12 1
3 8 4
10 11 2
12
1
9
7

… and the expected output (`test5.out`):

Sample Output
Impossible
-1
Impossible
-1
5 9
6
Impossible
-1

See `sssp_io.zip` for more test cases.

Submission

Submit a `.zip` file named **SurnameEx09.zip** to our Google Classroom. Inside this archive file, there must be a file named `main.c` which contains the `main()` function and other source files needed to run your program.

Questions

If you have any questions, approach your lab instructor.

References

Cormen, Thomas H., Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.

Weiss, Mark Allen. *Data Structures and Algorithms*. Benjamin/Cummings.