CS590 homework 7 – Graphs, and Shortest Paths

The due date for this assignment is Friday, Dec 7th, at 11.59pm. This assignment is worth 10% of your final grade.

Any sign of collaboration will result in a 0 and being reported to the Graduate Academic Integrity Board. Late submission policy described in the syllabus will be applied.

Develop a data structure for directed, weighted graphs $G = (V, E)$ using an adjacency matrix representation. The datatype `int` is used to store the weight of edges. `int` does not allow one to represent $\pm\infty$. Use the values `INT_MIN` and `INT_MAX` (defined in `limits.h`) instead.

```
#include <limits.h>

int d, e;

d = INT_MAX;
e = INT_MIN;

if (e == INT_MIN) { ...  }

if (d != INT_MAX) { ...  }
```

1. Develop a function `random_graph` that generates a random graph $G = (V, E)$ with n vertices and m edges taking the weights (integers values) at random out of the interval $[-w, w]$. In order to have a more natural graph we generate a series of random paths $v_0, v_1, \ldots, v_k$ through the graph. Each of the individual paths has to be non-cyclic. The combination of the random paths might contain a cycle. The edges used in the series of random paths has to add up to the desired m edges.

   Notes:

   - Determine the number of maximal allowed edges for a non-cyclic random path $v_0, v_1, \ldots, v_k$.
   - How do you ensure that the path is random? A random permutation of the vertices might be useful.
   - How do two random path that cross each other (share one or more edges) effect the overall edge count? Shared edges should be counted only once.

2. Describe (do not implement) how you would update the above implemented `random_graph` method to generate a graph $G = (V, E)$ that does not contain a negative-weight cycle. You are given a function that can determine whether or not an edge completes a negatice-weight cycle.

3. Implement the Bellman-Ford algorithm. What is the running time for Bellman-Ford using an adjacency matrix representation?

4. Implement the Floyd-Warshall algorithm. Your implementation should produce the shortest weight matrix $D^{(n)}$ and the predecessor matrix $\Pi^{(n)}$. Limit the number of newly allocated intermediate result matrices.

(1. 30pts, 2. 20pts, 3. 25pts, 4. 25pts)

Remarks:

- You are not allowed to use code from online resources. Your submission will be tested against that, and will receive a 0, and a report to the Graduate Academic Integrity Board if it is detected.
- A Makefile is provided to build the code in LinuxLab.
- The programming, and testing will take some time. Start early.
- Feel free to use the provided source code for your implementation. You have to document your code.

Stevens Institute of Technology

Department of Computer Science

CS590: ALGORITHMS

Homework Assignment 7

Submitted by:
Hadia Hameed
CWID: 10440803

Yijia Tan
CWID: 10427079

Submitted to:
Prof. Iraklis
Tsekourakis

December 7, 2018

1 Problem I

1. Develop a function `random_graph` that generates a random graph $G = (V, E)$ with n vertices and m edges taking the weights (integers values) at random out of the interval $[-w, w]$. In order to have a more natural graph we generate a series of random paths $v_0, v_1, \ldots, v_k$ through the graph. Each of the individual paths has to be non-cyclic. The combination of the random paths might contain a cycle. The edges used in the series of random paths has to add up to the desired m edges.

   Notes:

   - Determine the number of maximal allowed edges for a non-cyclic random path $v_0, v_1, \ldots, v_k$.
   - How do you ensure that the path is random? A random permutation of the vertices might be useful.
   - How do two random path that cross each other (share one or more edges) effect the overall edge count? Shared edges should be counted only once.

Solution:

We create an *edge_ complete(G)* function to check if the paths add up to the desired |G.E| = m edges.

1.1 edges_complete(G):

Algorithm 1 Check if paths add up to the total number of edges.

```
1: procedure EDGES_COMPLETE(G)
2:     for for each vertex u ϵ G.V do
3:         for for each vertex v ϵ G.V do
4:             if G.Adj[u][v] != INT_MAX then
5:                 count++
6:     if count != |G.E| then
7:         return false
8:     else
9:         return true
```

© 2018 Stevens Institute of Technology - All Rights Reserved

```
bool graph::edges_complete()
{
   int count = 0;
   for (int i = 0; i < no_vert; i++)
     for (int j = 0; j < no_vert; j++)
       if (m_edge[i][j] != INT_MAX)
         count = count + 1;
   if (count == no_edges)
     return true;
   else
     return false;
}
```

1.2 random_graph(G)

Algorithm 2 Create a random graph with n vertices, m edges and weights in the interval [-w,w]

```
1: procedure RANDOM_GRAPH(N,M,W)
2:     perm = PERMUTATION(n)
3:     i = 0
4:     while i < n-2 and !edges_complete(G) do
5:         u = perm[i]
6:         v = perm[i + 1]
7:         if  G.Adj[u][v] == INT_MAX then
8:             weight = randomNumber(-w,w)
9:             G.Adj[u][v] = weight
           i++
```

```
void graph::random_graph()
{
  random_generator rg;
  int *perm = new int(no_vert);
  while (!edges_complete()) {
    permutation(perm, no_vert);
    int i = 0;
    while(i<(no_vert-2)&&!edges_complete()) {
      int v1 = perm[i];
      int v2 = perm[i + 1];
      if (m_edge[v1][v2] == INT_MAX) {
        random_generator rg;
        int randomWeight;
        rg >> randomWeight;
        int w = weights;
        randomWeight = randomWeight % (2*w + 1) + (-w);
```

© 2018 Stevens Institute of Technology - All Rights Reserved

```
        m_edge[v1][v2] = randomWeight;
        }
       i = i + 1;
    }
  }
  delete perm;
}
```

1.3 Output:

```
>> ./hw7 4 6 5
```

```
[[      Inf     -4      Inf     Inf     ]
[       Inf     Inf     4       Inf     ]
[       Inf     3       Inf     3       ]
[       -2      Inf     -4      Inf     ]
]
```

Figure 1: A random graph with 4 vertices and 6 edges with weights between [-5,5]

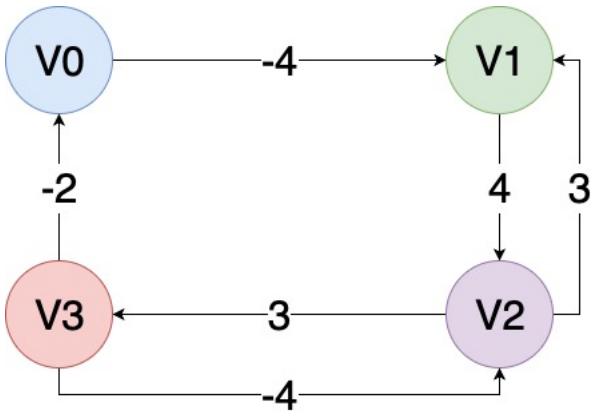

Figure 2: The graph associated with adjacency matrix generated in Fig. 1

1.4 Notes:

- The number of edges cannot be more than $n^2 - n$ to avoid self-directed cycles (entries on the diagonal of the adjacency matrix)

© 2018 Stevens Institute of Technology - All Rights Reserved

- A random permutation of numbers from $0, 1, 2, ..., n$ is taken where n is the total number of vertices. Then adjacent entries in the permuted array are taken to generate the edges. For example, if the resulting permuted array is [2,1,3,0] for $n = 4$ then the resulting edges will be (2,1), (1,3), (3,0). The corresponding index positions in the adjacency matrix are populated with random number from $[-w, w]$.
- The corresponding position in the matrix is only populated if that position is not already filled up with some non-infinity number (repeated edges are counted only once). Permuted combinations of the vertices are generated until the populated entries in the adjacency matrix add up to m edges.

2 Problem II

Solution:

Each time we add an edge to the set of edges, we will run the given function (GIVEN_FUNCTION(u,v)) to check if the edge *(u,v)* completes a negative cycle or not. If it does, then we will not populate the corresponding entry in the matrix. Otherwise, we will go ahead and insert a random weight from [-w,w] at the corresponding position. After the graph is generated, we will check if the random graph has a negative-weight cycle, using Bellman-Ford Algorithm. If it does, we will delete the allocated memory and call the random_graph function again.

Algorithm 3 Modified random_graph(n,m,w) to avoid negative-weight cycles.

```
1:  procedure MODIFIED_RANDOM_GRAPH(N,M,W)
2:      perm = PERMUTATION(n)
3:      i = 0
4:      while i < n-2 and !edges_complete(G) do
5:          u = perm[i]
6:          v = perm[i + 1]
7:          if G.Adj[u][v] == INT_MAX and !GIVEN_FUNCTION(u,v) then
8:              weight = randomNumber(-w,w)
9:              G.Adj[u][v] = weight
            i++
10:     if BELLMAN_FORD(G) then
11:         delete allocated memory
12:         modified_random_graph(n,m,w)
```

© 2018 Stevens Institute of Technology - All Rights Reserved

3 Problem III

Solution:

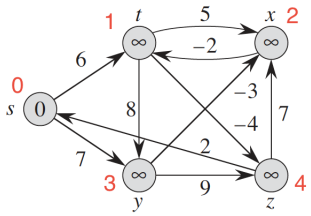

Figure 3: Input graph for Bellman-Ford Algorithm. Source: Page 652, *Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2009.*

```
----------------------------
BELLMAN-FORD ALGORITHM
----------------------------

Pass : 0
-------------------------
Relaxing Edge (0,1):
Changing distance of Vertex[1] to : 6
-------------------------
Relaxing Edge (0,3):
Changing distance of Vertex[3] to : 7

Pass : 1
-------------------------
Relaxing Edge (1,2):
Changing distance of Vertex[2] to : 11
-------------------------
Relaxing Edge (1,4):
Changing distance of Vertex[4] to : 2
-------------------------
Relaxing Edge (3,2):
Changing distance of Vertex[2] to : 4

Pass : 2
-------------------------
Relaxing Edge (2,1):
Changing distance of Vertex[1] to : 2

Pass : 3
-------------------------
Relaxing Edge (1,4):
Changing distance of Vertex[4] to : -2
-----------------------------
Negative Cycle Doesn't Exist.
-----------------------------
```

Figure 4: Output for Bellman-Ford Algorithm for input graph shown in Fig. 3.The vertices are number 0,1,2,3,4 instead of 1,2,3,4,5.

© 2018 Stevens Institute of Technology - All Rights Reserved

The running time for Bellman-Ford algorithm is O(VE). Running times for individual blocks in the algorithm are given in the code.

4 Problem IV

Solution:

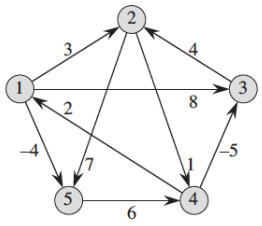

Figure 5: Input graph for Floyd-Warshall Algorithm. Source: Page 690, *Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2009.*

```
----------------------------
FLOYD-WARSHALL ALGORITHM
----------------------------
Input Graph

[[      Inf     3       8       Inf     -4      ]
[       Inf     Inf     Inf     1       7       ]
[       Inf     4       Inf     Inf     Inf     ]
[       2       Inf     -5      Inf     Inf     ]
[       Inf     Inf     Inf     6       Inf     ]
]

Floyd-Warshall: Shortest Distance Matrix:

[[      0       1       -3      2       -4      ]
[       3       0       -4      1       -1      ]
[       7       4       0       5       3       ]
[       2       -1      -5      0       -2      ]
[       8       5       1       6       0       ]
]

Floyd-Warshall: Predecessor Matrix:

[[      Inf     2       3       4       0       ]
[       3       Inf     3       1       0       ]
[       3       2       Inf     1       0       ]
[       3       2       3       Inf     0       ]
[       3       2       3       4       Inf     ]
]
```

Figure 6: Distance matrix and predecessor matrix output for Floyd-Warshall Algorithm for input graph shown in Fig. 5. The vertices are number 0,1,2,3,4 instead of 1,2,3,4,5. For the predecessor matrix, instead of NIL, INT_MAX was used to show the vertex did not have a predecessor.

© 2018 Stevens Institute of Technology - All Rights Reserved