Lab08-Graphs

VE281 - Data Structures and Algorithms, Xiaofeng Gao, TA: Li Ma, Autumn 2019

* Name: Jin Zhejian Student ID: 517370910167 Email: jinzhejian@outlook.com

1. **DAG.** Suppose that you are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinct nodes s and d. Describe an algorithm for finding a longest weighted simple path from s to d. For example, for the graph shown in Figure 1, the longest path from node A to node C should be $A \to B \to F \to C$. If there is no path exists between the two nodes, your algorithm just tells so. What is the efficiency of your algorithm? (Hint: consider topological sorting on the DAG.)

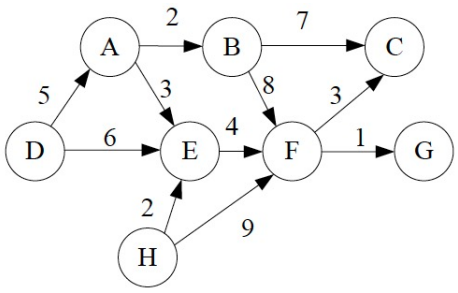

Figure 1: A weighted directed graph.

Solution.
My algorithm:

(a) Initialize distance array dist[] , where every entry is negative infinite except that: dist[s] = 0, which s is the source node.

(b) Topological sort the graph, save the result in a stack.

(c) for every vertex u in topological order{
    for every adjacent vertex v of u{
        if (dist[v] < dist[u] + weight(u, v)){
            dist[v] = dist[u] + weight(u, v)
        }
    }
}

Time complexity:

For the (b) part, the time complexity of topological sorting is O(V+E).

For the (c) part, we have v nodes and total adjacent vertices is E, so we need O(V+E).

Overall, the time complexity of this algorithm is O(V+E).

Implementation and outcome:

The DAG.cpp implementation of my algorithm is shown in the Appendix part.

The following is the output of my cpp code, in which I use number instead of alphabet for convenience (A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6, H = 7):

```
1  Following are longest distances from s 0 to d 2
2  Path is 0 1 5 2
3  distance is 13
```

□

2. **ShortestPath.** Suppose that you are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v. We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

Solution.
My solution is similar to the solution in the first problem.

My algorithm:

(a) Initialize distance array dist[] , where every entry is negative infinite except that: dist[s] = 1, which s is the source vertice.

(b) Topological sort the graph, save the result in a stack.

(c) for every vertex u in topological order{
    for every adjacent vertex v of u{
        if (dist[v] < dist[u] · r(u, v)){
            dist[v] = dist[u] · r(u, v)
            if (v is the end vertice) {
                add u to the path of source to end vertice
            }
        }
    }
}

Time complexity:

Still O(V+E). As explained in the first problem.

□

3. **GraphSearch.** Let $G = (V, E)$ be a connected, undirected graph. Give an $O(|V| + |E|)$-time algorithm to compute a path in G that traverses each edge in E **exactly once in each direction**. For example, for the graph shown in Figure 2, one path satisfying the requirement is

$$A \to B \to C \to D \to C \to A \to C \to B \to A$$

Note that in the above path, each edge is visited exactly once in each direction.

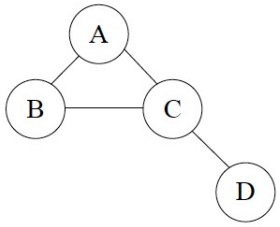

Figure 2: A undirected graph.

Solution. We can make a little change to DFS:

Alg. 1: DFS_change(u)

```
1  VISITED(u) = TRUE;
2  foreach v ∈ adj(u) do
3      if !VISITED(v) then
4          set depth[v] = depth[u] + 1
5          traverse edge u → v;
6          DFS_change(v);
7          traverse edge u → v;
8      else if depth[v] > depth[u] then
9          traverse edge u → v;
10         traverse edge u → v;
```

□

Appendix

(1) DAG.cpp

```
#include <iostream>
#include <limits.h>
#include <list>
#include <stack>
#define NINF INT_MIN
using namespace std;

class AdjListNode {
int v;
int weight;

public:
AdjListNode(int _v, int _w){
v = _v;
weight = _w;
}
int getV() { return v; }
int getWeight() { return weight; }
};

class Graph {
// No. of vertices
int V;
// Pointer to an array containing adjacency lists
list<AdjListNode>* adj;
void topologicalSortUtil(int v, bool visited[], stack<int>& Stack);
public:
Graph(int V); // Constructor
~Graph(); // Destructor
void addEdge(int u, int v, int weight);
// Finds longest distances from given source vertex
void longestPath(int s, int d);
};

Graph::Graph(int V) // Constructor
{
this->V = V;
adj = new list<AdjListNode>[V];
}

Graph::~Graph() // Destructor
{
delete [] adj;
}

void Graph::addEdge(int u, int v, int weight)
{
AdjListNode node(v, weight);
```

```
49 adj[u].push_back(node);
50 }
51
52 void Graph::topologicalSortUtil(int v, bool visited[], stack<int>&
      Stack)
53 {
54 // Mark the current node as visited
55 visited[v] = true;
56
57 // Recur for all the vertices adjacent to this vertex
58 list<AdjListNode>::iterator i;
59 for (i = adj[v].begin(); i != adj[v].end(); ++i) {
60 AdjListNode node = *i;
61 if (!visited[node.getV()])
62 topologicalSortUtil(node.getV(), visited, Stack);
63 }
64
65 // Push current vertex to stack which stores topological sort
66 Stack.push(v);
67 }
68
69 void Graph::longestPath(int s, int d)
70 {
71 stack<int> Stack;
72 int dist[V];
73
74 bool* visited = new bool[V];
75 for (int i = 0; i < V; i++) {
76 visited[i] = false;
77 }
78
79 // Call the recursive helper function to store Topological
80 // Sort starting from all vertices one by one
81 for (int i = 0; i < V; i++) {
82 if (!visited[i]) {
83 topologicalSortUtil(i, visited, Stack);
84 }
85 }
86
87 // Initialize distances to all vertices as infinite and
88 // distance to source as 0
89 for (int i = 0; i < V; i++) {
90 dist[i] = NINF;
91 }
92
93 dist[s] = 0;
94
95 // Process vertices in topological order
96 cout << "Path␣is␣" << s << "␣";
97 while (!Stack.empty()) {
```

```
// Get the next vertex from topological order
int u = Stack.top();
Stack.pop();

// Update distances of all adjacent vertices
list<AdjListNode>::iterator i;
if (dist[u] != NINF) {
for (i = adj[u].begin(); i != adj[u].end(); ++i) {
if (dist[i->getV()] < dist[u] + i->getWeight()) {
dist[i->getV()] = dist[u] + i->getWeight();
}
if (i->getV() == d) {cout <<  u <<  " " ;}
}
}
}

// Print the calculated longest distances
//    for (int i = 0; i < V; i++) {
//        (dist[i] == NINF) ? cout << "INF " : cout << dist[i] << " ";
//    }
cout << d <<  endl;
(dist[d] == NINF) ? cout << "distance is INF " : cout << "distance is
   " << dist[d]  << " ";

delete [] visited;
}

// Driver program to test above functions
int main()
{
int A = 0, B = 1, C = 2, D = 3, E = 4, F = 5, G = 6, H = 7;
Graph g(8);
g.addEdge(A,B,2);
g.addEdge(A,E,3);
g.addEdge(B,C,7);
g.addEdge(B,F,8);
g.addEdge(D,A,5);
g.addEdge(D,E,6);
g.addEdge(E,F,4);
g.addEdge(F,C,3);
g.addEdge(F,G,1);
g.addEdge(H,E,2);
g.addEdge(H,G,9);

int s = 0;
int d= 2;
cout << "Following are longest distances from s "
<< s  << " to d " << d << " \n";
g.longestPath(s,d);
return 0;
```

```
147 }
```