Lab08-Graphs

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

∗ Please upload your assignment to website. Contact webmaster for any questions.

∗ Name:Jintian Ge Student ID:517021911142 Email: gejintian@sjtu.edu.cn

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.)

   Solution. Basic idea is that firstly we use topological sorting to realize linearization. Then, we cut off the interval between s and d. Here we get $[s, v_0, v_1, v_2, ..., d]$. Next, we go through these vertex and delete all vertexes which cannot be inversely reached by u. Then, we use a vector DQ to contain these nodes. For each node, it has: a string presenting its name, and an indicator. An indicator is a pair, in which one is a pointer to another node and one is the total cost that the corresponding node will bring. We will make sure that $indicate.cost$ is the largest. When we reach s, we will visit the indicator, and go through the path. Finally we will find the longest path. If s has no indicator, then there is no path from s to d.

   In this graph, for example, if we want path from A to C, then node B in the vector DQ has two members: a sting 'B', which is its name, and an indicator that containing a pair: {Pointer to F(&F), cost: 11}. In the tracing back part, when we visit B we will choose F as the next node to visit.

   For efficiency, there are two parts of this algorithm might affect the efficiency. In the topological sort and delete part, it needs $O(|E|)$ work at most. In find path part, in the worst case(all vertexes are included) we need $O(|V|)$ work. Therefore the overall time complexity should be $O(|V| + |E|)$.

   The following is the two structures I mentioned before and the pseudo code of the algorithm.

```
struct indicator{
    node *pointer;
    int cost;
};
struct node{
    string name;
    indicator indicate;
};
```

□

Alg. 1: LongestPathSelect

input : A directed acyclic graph G=(V,E), two distinct vertices s and d.
output: Longest path from s to d

```
1  L=TopologicalSorting(G);
2  Cut off L such that M starts from s and ends at d;
3  EXPLORE(d) with all path inverse;
4  Delete vertices in M which were not be reached in EXPLORE(d) with all path
     inverse;
5  vector¡node¿ DQ;
6  for n in L do
7      node n(None, 0);
8      DQ.PushBack(n);
9  while not DQ.empty() do
10     v ← DQ.PopBack();
11     if (ForwardEdge(u, v) ∈ V)&&(u ∈ L) then
12         if u.indicate.cost < v.indicate.cost + PathWeight(u, v) then
13             u.indicate.cost ← v.indicate.cost + PathWeight(u, v);
14             u.indicate.pointer ← &v;
15 vector < string > Path;
16 if s.vect.empty() then
17     return No path;
18 else
19     v ← s;
20     while not v.vect.empty() do
21         p ← ∗v.indicate.pointer;
22         Path.pushBack(p.name);
23         v ← p;
24     return Path;
```

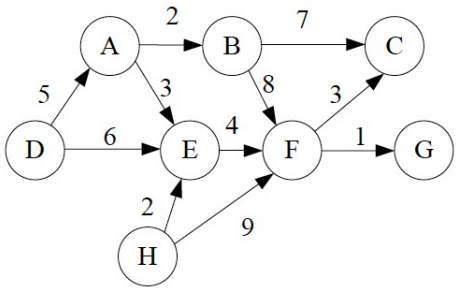

Figure 1: A weighted directed graph.

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. Basic idea is using Dijkstra's Algorithm. Meanwhile, each time we calculate reliability, we use multiply instead of add. And we update the value of each vertex if and only if $r(s,u) > r[u]$. Also, we will store the predecessor which causing the maximum reliability in that vertex.

Alg. 2: ReliablePathSelect

input : A directed graph G=(V,E), two distinct vertices s and d.
output: Most reliable path from s to d

```
1  s.r ← 0;
2  s.predecessor ← None;
3  INSERT(Q,s);
4  foreach u ∈ V/s do
5      u.r ← 0 INSERT(Q,u);
6  while Q ≠ ∅ do
7      u ← EXTRACT − MAX(Q);
8      S ← S ∪ {u};
9      foreach v ∈ Adj[u] do
10         if v.r < u.r × r(u,v) then
11             v.r ← u.r × r(u,v);
12             v.predecessor ← u;
13             DECREASE-KEY(Q,v);
14 return TraceBack(s,d);
```

□

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.

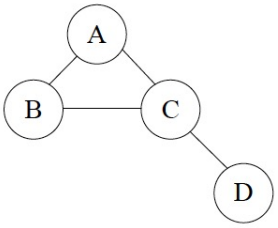

Figure 2: A undirected graph.

Solution. Since the graph is connected, we can use an improved DFS. First, I will redefine the EXPLORE recurrent function, and then use it in the path search. Note that I use $AddTrace(edge(*))$ to add one component to the path. For example, $AddTrace(edge(v,u))$ stands for add a trace '$v \to u$' to the path.

For the EXPLORE function:

Alg. 3: EXPLORE

input : A directed graph G=(V,E); a vertex $v \in V$
output: Find the path which traverses each edge exactly once in each direction

```
1  if v.predecessor then
2      AddTrace(edge(v.predecessor, v));
3  v.visit ← True;
4  foreach edge(v, u) ∈ E do
5      if not u.visit then
6          u.predecessor ← v;
7          EXPLORE(u);
8      else
9          if edge(v, u) ∉ Trace then
10             /*The edge between v and u had not been went through*/
                AddTrace(edge(v, u));
11             AddTrace(edge(u, v));
12     if v.predecessor then
13         AddTrace(edge(v, v.predecessor));
```

This is the whole pseudo code:

□

Alg. 4: GraphSearch

input : A directed graph G=(V,E)
output: Find the path which traverses each edge exactly once in each direction

```
1  foreach u ∈ V do
2      u.predecessor ← None;
3      u.visit ← None;
4  Randomly choose a vertex v;
5  u.visit ← True;
6  EXPLORE(G, v);
```