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:_____ Student ID:_____ Email: _____

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.)

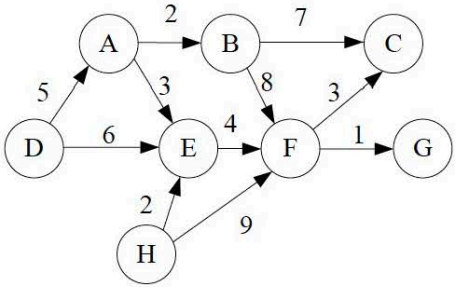

Figure 1: A weighted directed graph.

Solution. Basic idea: For a weighted directed acyclic graph G, when we specify the source node s, and we want to find the longest path from s to the other vertices, which is equivalent to copying G to G', then changing the weight of all the edges in G' to a negative value. In this way, the shortest path in G' is the longest path in the original G. □

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. In order to get the most reliable path, we aim to find a path p such that the product of the probabilities on that path is maximized. Let s be the source and t be the terminal, and let $p =< v_0, v_1, ..., v_k >$, where $v_0 = s$ and $v_k = t$, then

$$p = \arg\max \prod_{i=1}^{k} r(v_{i-1}, v_i) \tag{1}$$

This problem can be easily converted to a single-source shortest path problem by letting the weight on edge (u, v) be $w(u, v) = -\log r(u, v)$. Since logarithm will not change the monotonicity, and a minus logarithm will convert a minimization problem into a maximization problem, a shortest path p on the converted graph satisfying

$$p = \arg\min \prod_{i=1}^{k} w(v_{i-1}, v_i)$$

will satisfy (1) as well.

We use Dijkstra's algorithm to solve shortest path problem on the converted graph. The initialization of weights takes $O(E)$ time, and the rest are the same as Dijkstra's algorithm. We assume that the graph is sparse, i.e. $E = o(V^2/\log V)$, and all vertices are reachable from source, hence the algorithm runs in $O((V+E)\log V + E) + O(E) = O(E \log V)$. □

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.

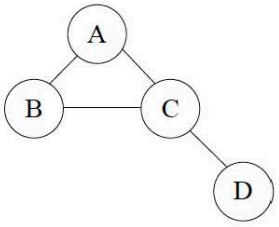

Figure 2: A undirected graph.

Solution. We can use a variant of depth-first search. Every edge is marked the first and second time it is traversed with unique marks for each traversal. Edges that have been traversed twice may not be taken again.

Our depth-first search algorithm must ensure that all edges all explored, and that each edge is taken in both directions. To ensure that all edges are explored, the algorithm must ensure that unexplored edges are always taken before edges that are explored once. To ensure that edges are taken in both directions, we simply backtracking until a new unexplored edge is found. This way, edges are only explored in the reverse direction during the backtracking.

Time complexity: $O(V + E)$

- Perform a depth-first search of G starting at an arbitrary vertex.(The path required by the problem can be obtained from the order in which DFS explores the edges in the graph.)
- When exploring an edge (u, v) that goes to an unvisited node the edge (u, v) is included for the first time in the path.
- When DFS backtracks to u again after v, the edge (u, v) is included for the 2nd time in the path, this time in the opposite direction(from v to u).
- When DFS explores an edge (u, v) that goes to a visited node we add $(u, v)(v, u)$ to the path. In this way each edge is added to the path exactly twice.

□