Algorithm Design and Analysis (Fall 2023)
Assignment 2
Deadline: Nov 27, 2023

1. (25 points) Design a polynomial time algorithm that, given a directed unweighted graph $G = (V, E)$ and $s \in V$, outputs "yes" if there is a vertex $u \in V$ such that there are at least two different simple paths from s to u and outputs "no" otherwise. A simple path is a path that does not visit a vertex more than once. Two paths are different if they differ in at least one edge. Prove the correctness of your algorithm and analyze its time complexity.

Solution:

The algorithm is below, which is a variant of DFS.

Algorithm 1 Find Two Paths

```
function DFS($s$): {Boolean vectors $visit[\,]$ and $finish[\,]$ are initialized to $False$.}
1: $visit[s] = True$
2: for each $v \in N(s)$ do
3:     if $visit[v] = False$ then
4:         DFS($v$)
5:     else if $finish[v] = True$ then
6:         return $True$
7:     end if
8: end for
9: $finish[s] = True$
```

Each edge will be visited at most once, and each vertex will be visited at most twice, so the time complexity is $O(|V| + |E|)$.

Now I'll prove the correctness of this algorithm. There are four kinds of edges in DFS search tree introduced in the lecture: tree edge, forward edge, back edge, cross edge.

Tree edge cannot independently form two different paths from s to v, and back edge will violent the assumption of simple path.

Cross edge points from a vertex u in one branch of the search tree to a vertex v in another branch. Let m denotes their nearest common ancestor, then two different path from m to v can be identified: one is the DFS search path from m to v, another is DFS search path from m to u extended with cross edge from u to v. Furthermore, there exist two distinct paths from s to v.

Forward edge points from the ancestor vertex u to the descendant vertex v, implying that there are two different paths from u to v. Then there must be two distinct paths from s to v.

Then we only need to check the existence of cross edges and forward edges. It turns out (the conclusion introduced in the lecture) that if (u, v) is either type, when we visit this edge, v must have finished exploration, otherwise it would be tree edge or back edge.

With this property, we can mark a node as “finish” when its exploration is complete. When visiting a previously “visited” vertex v, if it is also “finished”, then we identified a vertex that there are two paths from s to v. If the algorithm ends without finding such a vertex, it implies the absence of cross edge or forward edge, then we can output “no”.

2. (25 points) Given an undirected connected graph $G = (V, E)$ and a vertex s, if the DFS tree and the BFS tree rooted at s are identical, prove that G is a tree.

Solution:

I’ll show this by contradiction.

Suppose G isn’t a tree. Then let DFS run on this graph, and we can obtain a DFS search tree. In the undirected version, there are only two types of edges: tree edge and back edge. Otherwise, suppose there exists cross edge, we will visit another branch when exploring one branch through this cross edge, which changes the structure of the tree. Forward edge is exactly the back edge in the undirected version. Therefore, there are only back edge besides tree edge.

Since G isn’t a tree, there must be at least one back edge in DFS search tree. Let $\{u, v\}$ denotes this edge, where u is closer to the root vertex, i.e., u is the ancestor of v. So there are two paths from u to v, and v is closer to the root vertex through back edge $\{u, v\}$, while farther through DFS search path. So when applying BFS to this graph, it will choose this back edge instead of DFS search path. Then DFS and BFS tree are not identical, which leads to contradiction.

3. (25 points) Consider a directed *acyclic* edge-weighted graph $G = (V, E, w)$ where edge weights can be negative and a vertex $s \in V$.

   (a) (20 points) Adapt the Bellman-Ford algorithm to find the distances of all vertices from s. Your algorithm must run in $O(|V| + |E|)$ time. Prove the correctness of your algorithm and analyze its time complexity.

   (b) (5 points) Suppose now we want to find the length of the longest path from s to each vertex $u \in V$ (the length is measured by the sum of the weights of the edges on the path, not by the number of the edges). Can we negate the weight of every edge and use the algorithm from the first part? If so, prove it; if not, provide a counterexample.

Solution:

(a) The algorithm is below. It consists of two parts: one is performing topological sorting, another is applying Bellman-Ford algorithm based on topological order.

Algorithm 2 Topological Sorting

```
function DFS($s$): {Boolean vector $visit[\,]$ is initialized to $False$, $order = [\,]$.}
 1: $visit[s] = True$
 2: for each $v \in N(s)$ do
 3:   if $visit[v] = False$ then
 4:     DFS($v$)
 5:   end if
 6: end for
 7: Append $s$ to $order$.
```

Algorithm 3 Bellman-Ford Algorithm Adapted for DAG

```
function DFS($s$): {$order$ is obtained from algorithm 2.}
 1: Reverse the order of elements in $order$.
 2: Initialize $dist(u) = \infty$ for all $u \in V$, $dist(s) = 0$.
 3: for $i = 0, 1, ..., n-1$ do
 4:   $u = order[i]$
 5:   for each $v \in N(u)$ do
 6:     if $dist(v) > dist(u) + w(u, v)$ then
 7:       Update $dist(v) = dist(u) + w(u, v)$.
 8:     end if
 9:   end for
10: end for
```

For the first part, which is a variant of DFS, the time complexity is $O(|V| + |E|)$. For the second part, all edges and vertices will be visited only once, the time complexity is the same. So the total time complexity is $O(|V| + |E|)$.

Now I'll show the correctness of the algorithm. Claim that in the algorithm 3, when we visit a vertex u in topological order, $dist(u)$ reflects the correct distance from s. I'll show this by induction.

Base step:

When $i = 0$, $dist(s) = 0$ goes trivial.

Inductive step:

Let u_i denotes $order[i]$. Assumes that before the $(k-1)\,th$ iteration, $dist(u)$, $u \in \{u_0, u_1, ...u_{k-1}\}$ will reflect the correct shotest distance from s to u.

Then, before the $k\,th$ iteration, suppose that $\{s, v_0, v_1, ..., v_m, u_k\}$ is the shortest distance path from s to u_k. Since we visit vertices in topological order, vertices in this path before u_k will be visited earlier (i.e., before $(k-1)\,th$ iteration), therefore $v_m \in \{u_0, u_1, ...u_{k-1}\}$ and $dist(v_m)$ reflects the correct distance. Then we can correctly update $dist(u_k)$ by $dist(v_m) + w(v_m, u_k)$. This proves the inductive step and the claim.

Therefore, after $n-1$ iterations, we can correctly update distances of all vertices from s.

(b) Yes, the longest path can be obtained by negating the weight of every edge and then apply the algorithm of the previous part. Claim that in the algorithm 3, when we visit a vertex u in topological order, $dist(u)$ is exactly the negation of the longest distance from s. I'll show this by induction.

Base step:

When $i = 0$, $dist(s) = 0$ goes trivial.

Inductive step:

Assumes that the claim holds for $i = 0, 1, ..., k-1$, and the longest path from s to u_k is $\{s, v_0, v_1, ..., v_m, u_k\}$. Since this algorithm can find the shortest path, and $dist(v_m)$ is correct according to the assumption, then $dist(u_k)$ can also be calculated correctly.

4. (25 points) It is possible that the shortest path from s to t in an edge-weighted graph is not unique. Given a directed edge-weighted graph $G = (V, E, w)$ with positive edge weights and $s \in V$, design an $O((|V| + |E|) \log |V|)$ time algorithm to output a Boolean array B, with vertices being the array indices, such that $B[u] =$ `true` if the shortest path from s to u is unique and $B[u] =$ `false` otherwise. You can assume that every vertex $u \in V$ is reachable from s. Prove the correctness of your algorithm.

Solution: The algorithm adapted from Dijkstra is below, where a dynamic Boolean array B is introduce. Note that the value of $B[u]$ may be continuously changed before u is popped.

Algorithm 4 Determine Unique Shortest Path

```
function Dijkstra($G = (V, E, w),\ s \in V$):
 1: Initialize $dist(u) = \infty$ for all $u \in V$, $dist(s) = 0$.
 2: Initialize all elements in $B$ to False.
 3: $Q = [(s, dist(s)]$ {the priority queue}
 4: while $Q \neq$ do
 5:     $u = Q$.popmin()
 6:     for each $v \in N(u)$ do
 7:         if $dist(v) > dist(u) + w(u, v)$ then
 8:             $dist(v) = dist(u) + w(u, v)$
 9:             $Q$.modify$(v, dist(v))$
10:             $B[v] = B[u]$ {The number of same length of paths to $v$ is equal to that to $u$.}
11:         else if $dist(v) = dist(u) + w(u, v)$ then
12:             $B[v] = False$ {Multiple paths with the same length.}
13:         end if
14:     end for
15: end while
```

If Q is a priority queue implemented by standard heap, the time complexity is exactly same to Dijkstra $O((|V| + |E|) \log |V|)$.

Now I'll show the correctness of the algorithm.

First, claim that after vertex u is popped from Q, there won't be path from s to u whose distance is equal to $dist(u)$ without being updated. Otherwise, let v denotes the vertex before u on this path, and $dist(v) \geq dist(u)$ therefore $w(v, u) \leq 0$, violating the assumption of positive edge.

So when we visit (u, v), we only need to set $B[v]$ to `false` when $dist(v) = dist(u) + w(u, v)$, which implies there are multiple paths to v with the length $dist(v)$, and set $B[v] = B[u]$ when $dist(v) > dist(u) + w(u, v)$, which implies previous path to v is no longer the shortest, and therefore the number of paths to v whose length are current $dist(v)$ is equals to that of u. Since we can obtain true $dist(v)$ in the end, and all paths whose length are equal to $dist(v)$ will be founded, we have proven the correctness of the algorithm.

5. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.

   Solution:

   These four questions cost 2h, 1.5h, 4h, 1.5h respectively (typing also included).

   The difficulty scores are 2, 2, 4, 2 respectively.

   I complete this assignment independently.