CS 170 HW 5

Due on 2019-02-25, at 10:00 pm

1 Study Group

List the names and SIDs of the members in your study group.

2 Updating Labels

You are given a tree $T = (V, E)$ with a designated root node r, and for each vertex $v \in V$, a non-negative integer label $l(v)$. If $l(v) = k$, we wish to relabel v, such that $l_{\text{new}}(v)$ is equal to $l(w)$, where w is the kth ancestor of v in the tree. We follow the convention that the root node, r, is its own parent. Give a linear time algorithm to compute the new label, $l_{\text{new}}(v)$ for each v in V

Slightly more formally, the *parent* of any $v \neq r$, is defined to be the node adjacent to v in the path from r to v. By convention, $p(r) = r$. For $k > 1$, define $p^k(v) = p^{k-1}(p(v))$ and $p^1(v) = p(v)$ (so p^k is the kth ancestor of v). Each vertex v of the tree has an associated non-negative integer label $l(v)$. We want to find a linear-time algorithm to update the labels of all vertices in T according to the following rule: $l_{\text{new}}(v) = l(p^{l(v)}(v))$.

Solution:

Main Idea When we implement DFS with a stack, the stack at any given moment will always contain all the ancestors of the current node we're visiting. We want to maintain the labels of the relevant vertices currently on the stack, in a separate array. To ensure that our array only contains vertices on our current path down the DFS tree, we'll only add a vertex to our array (at index equal to the current depth) when we've actually visited it once (not when we first dd it to the stack). Since a path can have at most n vertices, the length of this array is at most n. Once we've processed all the children of a node, we can index into the array and set its label equal to the index of its kth ancestor. Notice that if we relabel the vertex before processing its children, we overwrite a label that the children of the vertex could depend on.

Runtime Analysis Since we add only a constant number of operations at each step of DFS, the algorithm is still linear time.

3 Count Four Cycle

Given as input an undirected graph $G = (V, E)$ design an algorithm to decide whether G contains a four cycle (A cycle $v - u_1 - u_2 - u_3 - u_4$ where $u_1 \neq u_2 \neq u_3 \neq u_1$ and $u_i \neq v$). Your algorithm should run in time $O(|V|^3)$. You may assume that the graph is given as either an adjacency matrix or an adjacency list.

Solution: For the solution, we will assume that the input is provided to us in the form of an adjacency matrix, A. We will now compute the matrix $B = A^2$ and $C = A^4$. Notice that the entries corresponding to B_{ii} and C_{ii} correspond to walks of length 2 and 4 from vertex i back to the vertex i. Furthermore, note that walks of length 4 from i back to i can either be two walks of length 2 from i back to i of the form $i - j - i - k - i$ or walks of the form $i - j - k - j - i$ or a cycle of length 4 from i to i. Therefore, the number of cycles of length 4 from i back to i, denoted by c_i^4 is given by:

$$c_i^4 = C_{ii} - B_{ii}^2 - \sum_{j \neq i} B_{ij}$$

Therefore, we compute c_i^4 for all vertices i and output, "YES", if we find a vertex with $c_i^4 > 0$ and "NO", otherwise. The overall running time of the algorithm is the time taken to multiply two $n \times n$ matrices which is at most $O(n^3)$ plus the time taken to compute c_i^4 for all vertices, which is $O(n^2)$ giving an overall running time of $O(n^3)$.

4 Constrained Dijkstra

Given as input a directed graph $G = (V, E)$, positive edge weights, ℓ_e, for each edge $e \in E$ and a particular vertex $v_o \in V$. Compute the shortest paths between all pairs of vertices in $O((|V| + |E|) \log|E|)$ time with the restriction that each of these paths pass through v_0.

Solution: Main Idea We start by computing the shortest paths from v_0 to every other vertex v in the graph using Dijkstra's algorithm. For each $v \in V$, we have a shortest path p_v^1 from v_0 to v. We now reverse the edges of the graph to obtain a new graph G^R with the same edge weights. We run Dijkstra's algorithm on G^R again starting at vertex v_0 to obtain for each vertex $v \in V$, a new path in G^R, p'_v from v_0 to v. We now reverse the edges, p'_v to obtain paths p_v^2 in G from v to v_0 for each vertex $v \in V$. Our algorithm returns for each vertex, $v \in V$, the pair of paths (p_v^1, p_v^2) and the shortest path between a pair of vertices, v_1 and v_2 is the concatenation of the paths $p_{v_1}^2$ and $p_{v_2}^1$.

Proof of Correctness To show that our algorithm is correct, first note that a shortest path from v_1 to v_2 containing v_0 consists of a path from v_1 to v_0 and a subsequent path from v_0 to v_2. Therefore, the shortest path from v_1 to v_2 containing the vertex v_0 is the concatenation of the shortest paths from v_1 to v_0 and the shortest path from v_0 to v_2. Our algorithm is correct if we simply show that we compute these two shortest paths correctly. The shortest path from v_0 to v_2 is correctly computed from the guarantees of Dijkstra's algorithm. To verify that we correctly compute the shortest path from v_1 to v_0 correctly, note that for every path in G from v_1 to v_0, we have a path from v_0 to v_1 in G^R. Therefore, the path p'_{v_1} is of length at most the length of the shortest path from v_1 to v_0. This proves the correctness of the algorithm because our algorithm correctly returns a pair of paths from v_1 to v_0 and from v_0 to v_2 which are the shortest possible paths.

Runtime Analysis Our algorithm consists of two runs of Dijkstra's algorithm along with a step where we reverse the edges of the graph. Therefore, our final runtime is $O((|V| + |E|) \log|V|)$.

5 Arbitrage

Shortest-path algorithms can also be applied to currency trading. Suppose we have n currencies $C = \{c_1, c_2, \ldots, c_n\}$: e.g., dollars, Euros, bitcoins, dogecoins, etc. For any pair i, j of currencies, there is an exchange rate $r_{i,j}$: you can buy $r_{i,j}$ units of currency c_j at the price of one unit of currency c_i. Assume that $r_{i,i} = 1$ and $r_{i,j} \geq 0$ for all i, j.

The Foreign Exchange Market Organization (FEMO) has hired Oski, a CS170 alumnus, to make sure that it is not possible to generate a profit through a cycle of exchanges; that is, for any currency $i \in C$, it is not possible to start with one unit of currency i, perform a series of exchanges, and end with more than one unit of currency i. (That is called *arbitrage*.)

More precisely, arbitrage is possible when there is a sequence of currencies $c_{i_1}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. This means that by starting with one unit of currency c_{i_1} and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$ and finally back to c_{i_1}, you would end up with more than one unit of currency c_{i_1}. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for profit.

We say that a set of exchange rates is arbitrage-free when there is no such sequence, i.e. it is not possible to profit by a series of exchanges.

(a) Give an efficient algorithm for the following problem: given a set of exchange rates $r_{i,j}$ which is *arbitrage-free*, and two specific currencies s, t, find the most advantageous sequence of currency exchanges for converting currency s into currency t.

Hint: represent the currencies and rates by a graph whose edge weights are real numbers.

(b) Oski is fed up of manually checking exchange rates, and has asked you for help to write a computer program to do his job for him. Give an efficient algorithm for detecting the possibility of arbitrage. You may use the same graph representation as for part (a).

Solution:

(a) **Main Idea:**
We represent the currencies as the vertex set V of a complete directed graph G and the exchange rates as the edges E in the graph. Finding the best exchange rate from s to t corresponds to finding the path with the largest product of exchange rates. To turn this into a shortest path problem, we weigh the edges with the negative log of each exchange rate. Since edges can be negative, we use Bellman-Ford to help us find this shortest path.

Pseudocode:

```
1: function BestConversion($s, t$)
2:     $G \leftarrow$ Complete directed graph, $c_i$ as vertices, edge lengths $l = \{-\log(r_{i,j}) \mid (i,j) \in E\}$.
3:     dist, prev $\leftarrow$ BellmanFord($G, l, s$)
4:     return Best rate: $e^{-\texttt{dist}[t]}$, Conversion Path: Follow pointers from $t$ to $s$ in prev
```

Proof of Correctness:
To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1} (-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph, taking s as origin. The correctness of the entire algorithm follows from the proof of correctness of Bellman-Ford.

Runtime:
Same as runtime of Bellman-Ford, $O(|V|^3)$ since the graph is complete.

(b) **Main Idea:**

Just iterate the updating procedure once more after $|V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

Pseudocode: This algorithm takes in the same graph constructed in the previous part.

```
1: function HasArbitrage($G$)
2:     dist, prev ← BellmanFord($G, l, s$)
3:     dist$^*$ ← Update all edges one more time
4:     return True if for some $v$, dist[$v$] > dist$^*$[$v$]
```

Proof of Correctness:

Same as the proof for the modification of Bellman-Ford to find negative edges.

Runtime:

Same as Bellman-Ford, $O(|V|^3)$.

Note:

Both questions can be also solved with a variation of Bellman-Ford's algorithm that works for multiplication and maximizing instead of addition and minimizing.

6 Bounded Bellman-Ford

Modify the Bellman-Ford algorithm to find the weight of the lowest-weight path from s to t with the restriction that the path must have at most k edges.

Solution: The obvious instinct is to run the outer loop of Bellman-Ford for k steps instead of $|V|-1$ steps. However, what this does is to guarantee that all shortest paths using at most k edges would be found, but some shortest paths using more that k edges might also be found. For example, consider a path on 10 nodes starting at s and ending at t, and set $k = 2$. If Bellman-Ford processes the vertices in the order of their increasing distance from s (we cannot guarantee beforehand that this will **not** happen) then just one iteration of the outer loop finds the shortest path from s to t, which contains 10 edges, as opposed to our limit of 2. We therefore need to limit Bellman-Ford so that results computed during a given iteration of the outer loop are not used to improve the distance estimates of other vertices during the **same** iteration.

We therefore modify the Bellman-Ford algorithm to keep track of the distances calculated in the previous iteration.

Algorithm 1 Modified Bellman-Ford

```
Require: Directed Graph $G = (V, E)$; edge lengths $l_e$ on the edges, vertex $s \in V$, and an
    integer $k > 0$.
Ensure: For all vertices $u \in V$, $dist[u]$, which is the length of path of lowest weight from $s$
    to $u$ containing at most $k$ edges.
1: for $v \in V$ do
2:     dist$[u] \leftarrow \infty$
3:     new-dist$[u] \leftarrow \infty$
4: dist$[s] \leftarrow 0$
5: new-dist$[s] \leftarrow 0$
6: for $i = 1, \ldots, k$ do
7:     for $v \in V$ do
8:         previous-dist$[v] \leftarrow$ new-dist$[v]$
9:     for $e = (u, v) \in E$ do
10:        new-dist$[v] \leftarrow \min($new-dist$[v]$, previous-dist$[u] + l_e$
```

Assume that at the beginning of the ith iteration of the outer loop, `new-dist`$[v]$ contains the lowest possible weight of a path from s to v using at most $i-1$ edges, for all vertices v. Notice that this is true for $i = 1$, due to our initialization step. We will now show that the statement also remains true at the beginning of the $(i+1)$th iteration of the loop. This will prove the correctness of the algorithm by induction. We first consider the case where there is no path from s to v of length at most i. In this case, for all vertices u such that $(u, v) \in E$, we must have `new-dist`$[u] = \infty$ at the beginning of the loop. Thus, `new-dist`$[v] = \infty$ at the end of the loop as well. Now, suppose that there exists a path (not necessarily simple) of length at most i from s to v, and consider such a path of smallest possible weight w. We want to show that `new-dist`$[v] = w$.

Let u be the vertex just before v on this path. By the induction hypothesis, at the end of the loop on line 7, `previous-dist`$[u]$ stores the weight of the lowest weight path of length at most $i-1$ from s to u, so that when the edge (u, v) is proceed in the loop on line 9, we get `new-dist`$[v] \leq w$.

Now, we observe that at the end of the loop on line 9, we have

$$\texttt{new-dist}[v] = \min\left(\texttt{previous-dist}[v], \min_{u:(u,v)\in E}\left(\texttt{previous-dist}[u] + l_{(u,v)}\right)\right).$$

Note that by the induction hypothesis, each term in the minimum expression represents the length of a (not necessarily simple) path from s to v of length at most i. Thus, in particular, none of these terms can be smaller than w, so that `new-dist`$[v] \geq w$. Combining with `new-dist`$[v] \leq w$ obtained above, we get `new-dist`$[v] = w$ as required.