CIS 675 — Asif Salekin

due tuesday nov 26, 2019 at 11.59 pm est via Blackboard

H4.Peinan Weng

- The assignment is due at Blackboard on Tuesday Nov 26 at 11:59pm. There is no Late submission. You can do multiple submissions. Submit early and often. Only the last submission will be considered.
- You are permitted to study with friends and discuss the problems; however, *you must write up you own solutions, in your own words*. Do not submit anything you cannot explain. If you do collaborate with any of the other students on any problem, please do list all your collaborators in your submission for each problem. You can have at-most 3 collaborators.
- You can study online. But, finding and writing solutions of homework problems from the web, is strictly prohibited.
- We hope that all homework submissions are prepared in Latex. If you need to draw any diagrams, however, you may draw them with your hand. Please use *exactly 1 page* for each of the answers. You can choose not to write the homework in Latex, but you can not take a photo of your text book and attach it. You have to submit it in a pdf format.
- If you take photo of your textbook and attach it, or if you do not submit your assignment as PDF file, your assignment will not be graded.

problem 1 *Vanilla*

is one of the most complex tastes in the world; it contains hundreds of different organic compounds that contribute to its flavor. Suppose a group of food enthusiasts are given n samples of vanilla: $s_1, ..., s_n$ by a sceptic. The sample is either a Mexican vanilla specimen or a Bourbon (French) vanilla specimen. The foodies are given each pair (s_i, s_j) of vanillas to taste, and they must collectively decide whether (a) both are the same type of vanilla, (b) they are different types of vanilla, or (c) they cannot decide. Note: all pairs are tested, including pairs such as (s_i, s_i), (s_j, s_i) and (s_i, s_j), but not all pairs have 'same' or 'different' decisions.

At the end of the tasting, suppose the foodies have made m judgements of 'same' or 'different'. Give an algorithm that takes these m judgements and determines whether they are consistent. The m judgements are consistent if there is a way to label each sample s_i with 'Mexican' or 'Bourbon' such that for every taste-test (s_i, s_j) labelled 'same', both s_i and s_j have the same label, and for every taste-test labelled 'different', both s_i and s_j are labelled differently. Your algorithm should run in time $O(m + n)$.

Solution 1 BFS - Graph Coloring

We need an array $color[i]$ to store the color status of each node, $color[i] = -1$ means node i has not been colored yet, $color[i] = 0$ means node i has already been colored with 0, $color[i] = 1$ means node i has already been colored with 1. Initialize every $color[i]$ with -1.

Our task is to color those nodes until every node has been colored. Each step, we select one node without any color. Then we do BFS starting from that node. During BFS, for a certain pair (S_i, S_j), if $S_i = S_j$, color S_j in the same color, if $S_i \neq S_j$, color S_j in the opposite color. Before each step, check if S_j has already been colored, if it has already been colored, check if it is consistent between that color and the color we are going to use, if there is an inconsistency, then the whole graph is inconsistent, stop traversing.

If we successfully colored every node in the graph without any inconsistency, then again we check every judgement pair, check if there is an inconsistency, if not, then the graph is consistent. $O(m)$

(When traversing the graph, if we find a pair with both nodes colored, after we check it's consistency, we do not need to do further search based on the latter node, this pruning method ensure each node will only be traversed once.) $O(n)$

Time Complexity is $O(m + n)$.

Solution 2 Disjoint Set

Build a disjoint set for this problem, traverse every pair, there are 2 cases.

Case 1, $S_i = S_j$, then $union(S_i, S_j)$.

Case 2, $S_i \neq S_j$, if $find(S_i) = find(S_j)$, that means inconsistent. Otherwise, continue traversing.

After we traversed all pairs without inconsistent, then return true.

Notice that the union and find operation in a disjoint set with path compression are both $O(1)$ and initialization cost $O(n)$. Time Complexity is $O(m + n)$.

PROBLEM 2 *All pairs shortest path*

Set $BSHORT_{i,j,k}$ to be the shortest path from i to j that uses only k hops. Note this is different from the ASHORT variable that we used in class in that we do not restrict the intermediate nodes to be $1 \ldots k$ in this formulation. State a recursive formula for $BSHORT$. Devise an algorithm that uses this recurrence. The run-time should be $O(V^2E)$. Explain the running time of this algorithm?

Solution

Traverse from node 1 to node k, for a certain node i, do BFS staring from that node, and fill the table of $BSHORT[i][j][k]$, $j = 0...V-1$, $k = 0...E$. During BFS, we should store the current length and store the distance from node i to each node in an array $visited[i]$ (Initialized with ∞). Notice that in this problem, if we find a path from node i to node j check if there is already a path from i to j (by checking whether $visited[j]$ is ∞) with less edges used. If there is, compare current cost and the cost stored in $visited[j]$, only if current cost is less than the cost stored in $visited[j]$, let $BSHORT[i][j][length]$ updated with current cost. That means we already have a path from i to j, but if we could use more edges, we could get a better solution, and we also need to update $visited[j]$ as well.

After traversing, we still have to deal with our result $BSHORT[i][j][k]$, traverse each array $BSHORT[i][j]$ from $k = 0$ to $k = E$, store the minimum value, and fill those unassigned value with this minimum value. We do this step because $BSHORT[i][j][k]$ denotes the minimum cost from i to j with no more than k hops, and after the first step, we still need to fill those values we do not assigned before. For example, it is obviously that $BSHORT[i][i][0] = 0$, but all other values in $BSHORT[i][i]$ array should also be assigned 0 as well.

Recursive formula, runtime explanation and C++ code is in the next page.

```cpp
class HW675_4 {
public:
    vector<vector<vector<int>>> problem_2(vector<vector<int>>& graph, int edgeNumber) {
        vector<vector<vector<int>>> result(graph.size(), vector<vector<int>>(graph.size(), vector<int>(edgeNumber + 1, 100000000)));
        for (int i = 0; i < graph.size(); ++i) {
            vector<int> visited(graph.size(), 100000000);
            queue<pair<int, int>> que;
            que.push({ i,0 });
            int length = 0;
            while (!que.empty()) {
                int size = que.size();
                while (size--) {
                    pair<int, int>& cur = que.front();
                    visited[cur.first] = min(visited[cur.first], cur.second);
                    result[i][cur.first][length] = min(result[i][cur.first][length], cur.second);
                    for (int j = 0; j < graph[cur.first].size(); ++j)
                        if (visited[j] > cur.second + graph[cur.first][j])
                            que.push({ j, cur.second + graph[cur.first][j] });
                    que.pop();
                }
                ++length;
            }
        }
        for (int i = 0; i < result.size(); ++i) {
            for (int j = 0; j < result[i].size(); ++j) {
                int mn = INT_MAX;
                for (int k = 0; k < result[i][j].size(); ++k) {
                    result[i][j][k] = mn = min(mn, result[i][j][k]);
                }
            }
        }
        return result;
    }
};
```

For the first part, outer loop with V steps, in each BFS, in worst case we need to traverse each node E times, and the runtime of each BFS is $O(VE)$, total runtime of the first part is $O(V^2E)$.

For the second part, 3 loops with V, V and E steps, runtime is $O(V^2E)$

Runtime of this algorithm is $O(V^2E)$.

PROBLEM 3 *Number of shortest paths*

Given a graph $G = (V, E)$, and a starting node s, let $\delta(s, v)$ be the length of the shortest path in terms of edges between s and v. Design an algorithm that computes the number of distinct paths from s to v that have length $\delta(s, v)$.

Solution

Using DP and BFS, we can solve this problem.

We need 2 arrays to store temporary results, $distance[i]$ - which means the current shortest length from s to i, and $count[i]$ - which means how many paths of the current shortest length from s to i in the graph.

In BFS, during each step, we refresh the *distance* array and the *count* array, there are several cases, for node i to node j.

Case 1, $distance[i] + 1 > distance[j]$. In this case, just do nothing to $distance[j]$ and $count[j]$.

Case 2, $distance[i] + 1 = distance[j]$. In this case, do nothing to $distance[j]$, however, add $count[i]$ to $count[j]$ ($count[j] = count[j] + count[i]$).

Case 3, $distance[i] + 1 < distance[j]$. In this case, replace $distance[j]$ with $distance[i] + 1$ ($distance[j] = distance[i] + 1$), and then replace $count[j]$ with $count[i]$ ($count[j] = count[i]$).

And we also need some base case. In this problem, $distance[s] = 0$, $count[s] = 1$ and all other $distance[i] = \infty$, all other $count[i] = 0$, then we could start BFS.

The final result is $count[v]$.