Please enter your name and uID below.

Name: Qianlang Chen
uID: u1172983

Submission notes

- Due at 11:59 pm on Friday, November 13.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Candy Corn Walks)

   My proposed algorithm has the following features:

   - The algorithm takes inputs of (1) the adjacency list representing graph G and (2) the starting vertex s. The adjacency list input should include the color of the incident edge for each neighbor as an integer: 1 for yellow, 2 for orange, and 3 for white.
   - The vertices and edges in this algorithm represent the vertices and edges in the given graph G.
   - The edges are directed, and each edge has an associated integer value according to its color in graph G.
   - The vertices will have an associated array of three boolean values, indexed 1, 2, and 3. The boolean at index k records whether the vertex has been "k-visited", which we'll describe later.
   - The algorithm will output a set containing the vertices that can be reached from the starting vertex via some candy corn walk.
   - The algorithm makes use of a Breadth-First Search with some modifications:
     - The queue used in the BFS will hold not just the next vertex, but also the color of the incident edge. For example, a pair $(a, 2)$ at the front of the queue indicates that a is our next vertex and we'll take an orange edge to get to it.
     - The queue will have the pair $(s, 0)$ as a start, where s is the input starting vertex.
     - The search will go on until the queue is empty. In each iteration, let's say the algorithm pops the queue and gets the pair (x, k). If $k = 3$, our algorithm will add x into the output set.
     - Next, unlike a regular BFS who'd add all x's unvisited neighbors into the queue, our algorithm will add a neighbor y only if (1) the color of xy is equal to either k or $k + 1$, and (2) y hasn't been marked l-visited, where l is the color of xy. If the algorithm ends up adding y to the queue, it'll then immediately mark y as l-visited.
   - When the search is over, the algorithm returns the output set constructed during the search.

   Let n and m be the number of vertices and edges in G, respectively. To analyze the run-time behavior of this algorithm, let's break it down into the initialization phase and the search phase:

   - The initialization phase assigns each vertex an array of three booleans filled with `false`. Since there are n vertices, this part will take $O(n)$.
   - The search phase will visit every edge in G at most once, just like in a regular BFS, and it'll visit every vertex at most three times. This is because a vertex is marked every time it's added to the queue, but it only gets three marks maximum, after which it'd never get added anymore. The only exception is the starting vertex, which may appear up to four times in the queue, but that's a constant. So, the search phase will take $O(3n + m) = O(n + m)$.

   Therefore, since the most time-consuming part of the algorithm takes $O(n + m)$, the entire algorithm takes $O(n + m)$.

2. (Graph DP)

My proposed algorithm is as follows. Note that, apart from the adjacency list representing the graph G, the algorithm also takes s and t, the only source and sink, as the inputs, assuming that the user knows them beforehand. We also assume that the vertices are named and referred to with strings.

Algorithm: `max_st_weight`

Input: `hash_map<string, hash_set<string>>` A, `string` s, `string` t, `function` w
Output: `number`

```
1 int n ← Number of keys in A
2 string L[1..n] ← Topological sort A
3 hash_map<string, number> M ← {}
4 M[t] ← 0
5 for int i = n − 1 to 1 do
6 |   string x ← L[i]
7 |   for string y in A[x] do
8 |   |   M[x] ← max(M[x], w(x, y) + M[y])
9 return M[s]
```

This algorithm uses the idea of the following backtracking algorithm, which returns the maximum weight from any vertex x to the only sink t:

$$\mathrm{BT}(x) = \begin{cases} 0, \ \textbf{if } x = t \\ \max[w(xy) + \mathrm{BT}(y) \ \textbf{for } y \text{ in out-neighborhood of } x], \ \textbf{otherwise} \end{cases}$$

The idea of the above backtracking algorithm is that, given some current vertex x, we have to make a choice on which edge to go down from x, and the maximum weight for each choice is just the chosen edge's weight plus the maximum weight after taking this edge. The maximum weight from x then is the maximum across all choices, except if x is already the destination vertex t, where there's no more edge or weight to take. Since the graph is acyclic, every step we take from x *will* be closer to t in terms of the number of steps left, which is advantageous because then any out-neighbor of x would not depend on x for its maximum weight to t. This paved the way for our dynamic programming algorithm:

- The algorithm can use memoization to avoid calculating the same value twice. I used a hash-map for its fast lookup with vertices' names.
- The topological sort conveniently gives us the order we need to calculate in (line 2).
- The base case is when the current vertex is t, which has a maximum weight of zero (line 4).
- Since the function value of x is purely dependent on its out-neighbors, which are positioned *later* in the topological order, I filled the memoization table in *reverse* order (line 5), using the strategy presented in the backtracking algorithm (lines 6-8).

Let's analyze the dynamic programming algorithm's run-time complexity. First, we know that topological sort runs in $O(n + m)$, where n and m are the number of vertices and edges in G. Assuming that the weight-function w has constant running time, our "table-filling" portion runs in

$$O(1 + \text{outdegree}(v_1)) + \cdots + O(1 + \text{outdegree}(v_n)) = O(n + \sum_{v \in V} \text{outdegree}(v)) = O(n + m)$$

Therefore, the entire algorithm runs in $O(n + m)$.

3. (MST Properties)

4. (Light-st Walk)

5. (MST Updates)