MPCS 55001 Algorithms Autumn 2023
Homework 5

Zhiwei Cao zhiweic
Collaborators: Student A, Student B

Instructions: Write up your answers in LaTeX and submit them to Gradescope. **Handwritten homework will not be graded.**

Collaboration policy: You may work together with other students. If you do, state it at the beginning of your solution: **give the name(s) of collaborator(s)** and the nature and extent of the collaboration. Giving/receiving a hint counts as collaboration. Note: you must write up solutions **independently without assistance**.

Internet sources: You must include the url of any internet source in your homework submission.

1 UPS Routing (12 points)

UPS trucks avoid making left turns when possible to save fuel and avoid traffic accidents. We are interested in determining an algorithm to find a route for a truck that uses the fewest left turns.

Before we solve this problem, we will discuss a related one. Given any directed weighted graph where each edge weight is either 0 or 1, we want to be able to find a path with the minimum possible weight.

(a) (3 points) Write pseudocode for an algorithm that takes as input a directed weighted graph with 0-1 edge weights, as well as a start node s and end node t. Your algorithm should return a path from s to t with minimum weight. Your algorithm should run in $O(|V| + |E|)$ time.

(b) (2 points) Prove your algorithm is correct, and runs in $O(|V| + |E|)$ time.

The running time is $O(|V| + |E|)$ since $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Each vertex added to the queue has a constant time (when the edge has a 0 weight value), but the number of operations performed for the vertex is still constant which only takes O(1), and the same as the edges. In sum the running time is $O(|V| + |E|)$.

We are now prepared to solve the original problem. We will consider the case where the road network is axis-aligned, meaning all roads travel in a cardinal direction (north, east, south, or west). We can represent this as a directed, unweighted graph $G = (V, E)$, where each intersection is represented by a vertex, and each road is represented as a edge between the two intersections it connects. Each edge has a `dir` field that indicates the direction of travel on that road: N, E, S, or W. Due to the axis-aligned structure, each vertex will have at most 4 incoming edges and 4 outgoing.

(c) (3 points) Write **pseudocode** for an efficient algorithm that takes as input an axis-aligned road network graph (as described above), as well as s a start node s and end node t. It should return a path from vertex s to vertex t that minimizes the number of left turns.

(d) (2 points) Prove that your algorithm is correct.
Initialization: Before the while loop iteration, there is only the start node s stores into the queue structure, which has 0 turns, 0 distance away from the start node, and NIL parent which is correct for the start node. Since the start node is the root node, which correctly represents the minimum number of left turns, the distance and parent from s to s itself. As a result, it is true prior to the first iteration of the loop.
Maintenance: During the iteration, the u with the smallest number of left turns will pop up. Then check the adjacent nodes (v) of the u, and check if there is a better path to v from u. If the value of d[v], t[v], and $\Pi[v]$ is changed by the if statement, which means the algorithm makes sure that the path to v from u is updated to a better one.

Algorithm 1 1 (a): BFS to find the path with minimum weight

```
1: function BFS(G, s)
2: Input: a directed weighted graph with 0-1 edge weights, G = (V, E)
3: Input: a start node s
4: Input: a end node t
5: Output: returns the parent of t on BFS tree
6: for each u in V do
7:     color[u] ← WHITE
8:     d[u] ← ∞
9:     Π[u] ← NIL
10: end for
11: color[s] ← GRAY
12: d[s] ← 0
13: Π[s] ← NIL
14: Q ← ∅ // deque structure
15: APPEND(Q, (s, 0)) // O(1)
16: while Q ≠ ∅ do
17:     (u, weight) = DEQUEUE(Q)
18:     for each v in Adj[u] do
19:         if d[v] > d[u] + weight(u, v) then // double check did we get the minimumu
20:             d[v] ← d[u] + weight(u, v)
21:             Π[v] ← u
22:             if weight(u, v) == 0 then
23:                 APPENDLEFT(Q, (v, 0)) // O(1)
24:             else
25:                 APPEND(Q, (v, 1)) // O(1)
26:             end if
27:         end if
28:     end for
29: end while
30: if t == s then
31:     print(s)
32: else if Π[t] == NIL then
33:     print("No Path")
34: else
35:     BFS-Print-Tree(G, s, Π[t])
36:     print(v)
37: end if
```

Algorithm 2 1 (c): BFS to find the path with the latest left turn

```
1: function BFS(G, s)
2: Input: an axis-aligned road network graph, G = (V, E)
3: Input: a start node s
4: Input: a end node t
5: Output: return a path from vertex s to vertex t that minimizes the number of left turns
6: for each u in V do
7:     t[u] ← ∞ // number of left turns of current node
8:     d[u] ← ∞
9:     Π[u] ← NIL
10: end for
11: t[s] ← 0
12: d[s] ← 0
13: Π[s] ← NIL
14: Q ← ∅ // deque structure
15: ENQUEUE(Q, s) // O(1)
16: while Q ≠ ∅ do
17:     u = DEQUEUE(Q)
18:     for each v in Adj[u] do
19:         nextDir ← dir(u, v)
20:         dist ← d[u] + 1
21:         if newDir is a left turn base on the u's position then
22:             turns ← t[v] + 1
23:         else
24:             turns ← t[v] + 0
25:         end if
26:         if t[v] > turns then
27:             d[v] ← dist
28:             t[v] ← turns
29:             Π[v] = u
30:             ENQUEUE(Q, v) // O(1)
31:         end if
32:     end for
33: end while
34: if t == s then
35:     print(s)
36: else if Π[t] == NIL then
37:     print("No Path")
38: else
39:     BFS-Print-Tree(G, s, Π[t])
40:     print(v)
41: end if
```

Termination: At the end of the while loop iteration, the queue will go through all the vertices of the graph and store them into the parent list which ensures that the number of left turns is the minimum number. At the same time, if the distance number of the vertex is not infinity it means the vertex is in the BFS tree. In the BFS print function, it will figure out if the t node has a parent until it reaches the s node with the shortest path with the minimum left turns.

(e) (2 points) Analyze the running time of your algorithm.

The running time is $O(|V| + |E|)$ since $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Each vertex added to the queue has a constant time, but the number of operations performed for the vertex is still constant which only takes O(1), and the same as the edges. At the same time, when there is an optimal path between s and t (Line 26-30), the if statement also takes O(1) to check the condition. In sum, the running time is $O(|V| + |E|)$.

2 New Language (12 points)

You are given an alphabet (a set of letters) and a list of words over this alphabet. The lexicographical order of the alphabet is not known, e.g., even if the alphabet is the same as English, "b" may come before "a" in this language.

The words are **sorted** according to the unknown ordering of the alphabet. Your task is to write an algorithm which returns a possible ordering for the alphabet. You may call algorithms used in class.

Example Input:

- `alphabet = {"a", "b", "c", "d"}`
- `sorted_words = ["bca", "baa", "dac", "aba"]`

Example Output (any of the answers below is correct):

- `["b", "d", "c", "a"]`
- `["b", "c", "d", "a"]`
- `["c", "b", "d", "a"]`

Note: There may be several possible answers because the sorted list may not give us enough information to determine the relative positioning of all letters. In this example, "c" must come before "a", but its relationship to "b" and "d" is undetermined.

(a) (5 points) Write **pseudocode** for an efficient algorithm to solve this problem. You may call algorithms presented in class.

(b) (5 points) Prove that your algorithm is correct.

Initialization of the graph At the beginning, the algorithm only add all element of the alphabet list as the vertices into the graph. Construct a graph with edge only.
Maintenance of the graph Compare each two words and find the difference between words based on the basic idea of alphabet order. Then construct the graph in an adjacent list format, which means the letter' comes after the letter in the "new unknown ordering of the alphabet" must be adjacent to the letter. In other words, the letter $\rightarrow$ letter' in the directed graph.
Termination of the graph After checking all elements of the *sorted_words*, each vertex of the graph will get its necessary adjacent elements. As a result, a directed graph in adjacent list formation was created.

Use the induction to prove the Topological sort.
Basis Case: When V = 1, it means there is only one vertex in the DAG. As a result, the algorithm will print this out as a sorted vertex since there are not adjacent vertices.
Inductive Step: Assume the topological sort will produce a correct order for a given DAG with k vertices. Now consider the k+1 vertices case.
Pick the "root vertex" of the DAG graph which has no incoming edges, and remove it from the DAG which will produce a graph with k vertices. By the induction hypothesis, the topological sort will produce a correct order with k vertices. Then add the "root vertex" back to the graph. It could be placed in the beginning of the sorted order of the sub-graph with k vertices. Since the vertex doesn't have any incoming edges which means doesn't rely on any other vertex. Thus topological sort works for a DAG with k+1 vertices as well.
Conclusion: Since the topological sort works well for the basis case which has only one vertex and also works for $k+1$ vertex DAG graph. Then the correctness of the topological sort is proved.

(c) (2 points) Let k be the size of the alphabet and m be the number of words provided. Analyze the running time of your algorithm in terms of m and k. You may assume that each word has at most 10 letters.

Graph construction takes $O(k+m)$ time. Since there are k vertices and need to check $(m-1)$ times for the comparison of the words which will take $O(k+(m-1))$, could be concluded to $O(k+m)$.
Topological sort also takes $O(k+m)$, it will check each k and k's adjacent element which takes time on check edges. As a result, the time could be concluded to $O(k+m)$.
Overall Time: $O(k+m)$

Algorithm 3 2 (a): a possible ordering for the alphabet

```
function LetterOrder(alphabet, sorted_words)
Input: a set of letters
Input: a list of words over this alphabet
Output: return a possible ordering for the alphabet
// generate a directed graph G in adjacent list format
for each letter in alphabet do:
    add letter to G.V // G only has the node at this time
end for
// add edge to the graph
for i ← 0 to sorted_words.length − 1 do
    w1 ← sorted_words[i]
    w2 ← sorted_words[i + 1]
    checkRange ← min(w1.length, w2.length)
    for char ← 0 to checkRange do
        if  thenw1[char] ≠ w2[char]
            add w2[char] to G.Adj[w1[char]]
            Break the iteration
        end if
    end for
end for
//at this moment I get a Graph G in adjacent list format
//a → /
//b → d
//c → a
//d → a

// reference from https://www.geeksforgeeks.org/topological-sorting/
function: TopologicalSortUtil(s, visited, stack)
assign s as visited
for each node in Adj[s] do
    if node is not visited then
        TopologicalSortUtil(node, visited, stack)
    end if
end for
stack.append(s)

function TopologicalSort(G)
stack ← [] // init a stack to store the order
Mark all vertices as not visited
for each s in G do:
    TopologicalSortUtil(s, visited, stack)
end for
res ← []
while stack is not empty do
    res.append(stack.pop())
end while
return res
```

3 Oh so many routes (12 points)

Konstantinos is close to graduation and is already making plans for a road trip to celebrate. Chicago to Los Angeles is a long way, and he needs to decide on a route. The obvious answer would be Route 66, but that would be too conventional. He has identified all the interesting places to visit in the US and wants to find a route passing through some of them. Before finding the best route, he wonders how many **possible** routes exist.

Example: in the following graph there are three routes from S to T.

- $S \to B \to T$
- $S \to B \to D \to T$
- $S \to C \to D \to T$

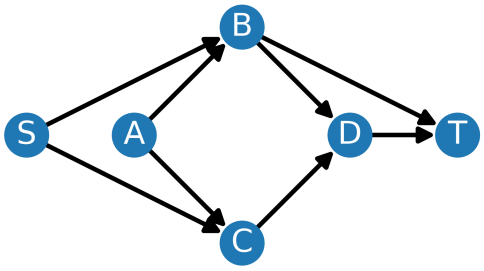

Figure 1: Example graph. S represents Chicago and T represents Los Angeles

You are given a (unweighted) directed acyclic graph of all the Points of Interest (POIs) between Chicago and Los Angeles. You must write a dynamic programming algorithm to return how many possible routes there are between Chicago and Los Angeles.

(a) (3 points) Define the subproblem(s). Define each variable you introduce.
Define a dp list. $dp[i]$ represent the number of way to reach the vertex i from Chicago, the S vertex. The size of dp is the number of vertices including the S and T. And a directed graph in adjacent format.

(b) (3 points) Write a recurrence expressing the solution of the subproblem in terms of smaller subproblems: i.e., write a formula that can be used to calculate the entries of your DP table from previous entries. Specify the base case(s).
$dp[i] = \sum_{i' spredecessors} dp[j]$ Take the **Example** as an instance, T's predecessors are D and B, dp[D]=2 and dp[B]=1, then dp[T] = 3.
Base Case: dp[s] = dp[Chicago] = 1. Since there is only one way to reach Chicago at first.

(c) (2 points) Write **pseudocode** for your algorithm. **Note: You must write a dynamic programming algorithm to receive credit.**

(d) (3 points) Dynamic Programming is a bottom-up approach. The equivalent top-down approach is called memoization. In memoization, the result of every function call is saved and if the function is called again with the same arguments, the cached result is immediately returned instead of being recomputed. Unlike DP, only the subproblems relevant to the problem we care about will be computed. Write pseudocode to solve this problem using memoization.

(e) (1 point) Describe a case where using memoization is more efficient than DP.
Image that there are hundreds of places that Konstantinos wants to visit between Chicago and LA. However, she can not go to every place. In this scenario, the DP approach will still construct a dp table with the length of places, and fix each sub-problem of the DP table. However, many of the approaches don't relate to the solution at all. For the memorization approach, it comes with the final goal of the solution. Based on the final goal, the memorization approach only searches the related sub-problems, which is much more efficient in this case.

Algorithm 4 3 (c): Find all possible routes from S to T

```
1: function SumRoutes(G, S, T)
2: Input:  a (unweighted) directed acyclic graph in adjacent format
3: Input: S - the start vertex (Chicago)
4: Input: T - the end vertex (LA)
5: Output: return amount of possible routes between S(Chicago) and T(Los Angeles).
6: numOfPlace ← numberofverticesoftheDAG
7: dp[S] ← 1
8: sortedNodes = TopologicalSort(G) // [A, S, C, B, D, F] for the example case
9: for each node in sortedNodes do
10:     for each v in Adj[node] do
11:         dp[v] ← dp[v] + dp[node]
12:     end for
13: end for
14: // dp = [0, 1, 1, 1, 2, 3]
15: return dp[T]
16:
17: // reference from https://www.geeksforgeeks.org/topological-sorting/
18: function: TopologicalSortUtil(s, visited, stack)
19: assign s as visited
20: for each node in Adj[s] do
21:     if node is not visited then
22:         TopologicalSortUtil(node, visited, stack)
23:     end if
24: end for
25: stack.append(s)
26:
27: function TopologicalSort(G)
28: stack ← [] // init a stack to store the order
29: Mark all vertices as not visited
30: for each s in G do:
31:     TopologicalSortUtil(s, visited, stack)
32: end for
33: res ← []
34: while stack is not empty do
35:     res.append(stack.pop())
36: end while
37: return res
```

Algorithm 5 3 (d): Find all possible routes from S to T suing memeoization

```
Reference: https://chat.openai.com/share/98f7e764-afc2-4b1c-90e5-2d339cb93e6f
function SumRoutesMemorization(G, CurNode, T, memo)
Input:  a (unweighted) directed acyclic graph in adjacent format
Input: CurNode - the node that algorithm is checking
Input: T - the end vertex (LA)
Input: memo - the memoization table
Output: return amount of possible routes between S(Chicago) and T(Los Angeles).
count_route ← 0
if CurNode in memo then
    return memo[CurNode]
end if
if CurNode == T then
    return 1
end if
for each v in Adj[CurNode] do
    count_route ← count_route + SumRoutesMemorization(G,v,T,memo)
end for
// update the memorization
memo[CurNode] ← count_route
return memo[CurNode]
```

4 Programming: Noah's Ark (20 points)

Follow this GitHub Classroom link to accept the assignment. Your code should be pushed to GitHub; you do not need to include it here.