CSCI 665 Foundation of Algorithms

Due 11/21 11:59pm

Homework 5 **Guo, Zizhun (zg2808)**

Problem 1

Algorithm Description

For this problem, I implemented a modified Breadth First Search algorithm to find the number of shortest paths from given s vertex to t vertex.

First, allocate an integer array called *dist*, where *dist[v*] represents the distance between the starting vertex s to the current vertex v. While BFS traversing to each vertex, dynamically check the current vertex neighbors. If the neighbor's distance has not been visited, assign the current vertex (distance + 1) to the neighbor, and initialize the neighbor's number of paths with its own number.

$\mathrm{P[v]} = P(parent)\ when\ v\ is\ visited\ by\ one\ of\ parents\ for\ the\ first\ time.$

$\mathrm{P[v]} = P[p1] + P[p2] + \cdots + P[pi]\ \forall\ v's\ parents' P$

Correctness:

This algorithm is processed dynamically, after traversed a vertex, all it's children would be assigned and accumulated with the number of paths, when it is done, the ending vertex would have a number in total, which is correct for the output.

Running Time Estimate

Since it is modified by BFS, so the time complexity is bounded by $O(n + m)$.

Pseudocode

numPaths(array):

initialize an array of dist[]

initialize an array of path[]

create a queue for BFS

seen[s] is true

dist[s] is 0

path[s] is 1

push s in the queue

while queue is not empty:

pop v in the queue

for all v's neigbors n:

if n is not seen:

mark seen[v] is true

push n in the queue

if dist[n] is greater than dist[v] + 1: (first time have neighbor visited)

dist[n] = dist[v] + 1

path[n] = path[s]

else if dist[n] is equal to dist[v] + 1 (detected n' parent is v)

path[n] = path[s]

Problem 2

Algorithm Description

For this problem, I implemented the dynamic programming algorithm to longest sequence of courses.

The problem can be comprehended as to output the longest path of an unweighted directive graph, for the prerequisite of current course is the current course's parent node. So here goes the heart of solution:

$$\mathrm{S[v]} = the\ longest\ length\ of\ the\ path\ ended\ at\ vertex\ v$$

$$\mathrm{S[v]} = \begin{cases} 1 & if\ the\ vertex\ v\ has\ no\ parent \\ \max(S[v1], S[v2], \dots, S[vi]) & \forall\ v's\ parents\ from\ 1\ to\ i \end{cases}$$

I also implemented a Topological Sort Algorithm to sort all vertices in topological order, which ensures it is possible to traverse all directed paths. I employed a stack to contains the sorted vertices in topological order, so when I used it to performs the DP algorithm, I can just pop out each vertex. I also created two adjacent linked list to contains the original directed graph and its reversed version.

Running Time Estimate

For DP algorithm, the running time estimate is $O(n+m)$.. The topological sort takes $O(n+m)$.. So overall time estimate for this algorithm is $O(n+m)$..

Pseudocode

maxPrerequisits (G(v)):

int max;

for each vertex among V vertices:

S[v] = 1

While sorted vertices' length > 0:

v has no neighbors:

S[v] = 1

Else:

S[v] = max (S[v1], S[v2], …, S[vi]) v1, v2, …, Vi are v's neighbors

max = Max(S[v], max)

return max

Problem 3

Algorithm Description

For this problem, I implemented a Breadth First Search algorithm to find the smallest number of moves for Thing One and Thing Two to get out.

The algorithm is the same as the typical BFS. There has a State Class to represent each state including the current maze (walls and movable grids) and the current coordinates of Thing One and Thing Two. The positions of two Things are also considered to be a wall when configures it.

In the section of traversing all next states from a current state, there are four directions that are available for Things to choose from. Take example as going to the West for a single move, there are three 4 situations: 1. Thing One move, thing Two stays. 2. Thing One stays, thing Two move. 3. Thing One moves, thing Two moves. 4. Both thing One and thing Two stays.

Correctness:

Here are the possible situations for each direction to choose:

$$\begin{cases} Choose\ to\ go\ to\ the\ West \begin{cases} Thing\ One\ moves, Thing\ two\ stays & if\ there\ a\ wall\ on\ the\ west\ of\ Thing\ Two \\ Thing\ One\ stays, Thing\ two\ moves & if\ there\ a\ wall\ on\ the\ west\ of\ Thing\ One \\ Thing\ One\ moves, Thing\ two\ moves & if\ there\ are\ no\ walls\ on\ the\ West \\ Both\ Thing\ staty & if\ there\ are\ no\ walls\ on\ the\ West \end{cases} \\ Same\ on\ the\ condition\ of\ East\ \dots \\ Same\ on\ the\ condition\ of\ North\ \dots \\ Same\ on\ the\ condition\ of\ South\ \dots \end{cases}$$

The algorithm dynamically generates a temporary linked list in the type of State class to represents all neighbors' states of the current state. There exists a function to check each time while traversing the neighbors that if the neighbor state is the exiting state (Thing One and Thing Two are both on the border of maze), print out the smallest number of moves recorded in the array.

Running Time Estimate

In worst case, if the existing state is the last state for the BFS algorithm to reach, and there are no walls at all in the maze, the algorithm is bounded by running row x column times, which is $O((ab)^2)$..

Pseudocode

doubleTrouble(current state s):

init: int move[row][column][row][column]

init: int seen[row][column][row][column]

seen[s.x1][s.y1][s.x2][s.y2] = true

Instantiate a new Queue

Euqueue(s)

```
While length of queue != 0:
    Dequeue state temp
    Get all neighbors of temp
    For all temp's neighbors:
        n is from neighbors
        If seen[temp.x1][temp.y1][temp.x2][temp.y2] = false
        Mark seen[temp.x1][temp.y1][temp.x2][temp.y2] = true
        move[n.x1][n.y1][n.x2][n.y2] = move[temp.x1][temp.y1][temp.x2][temp.y2] + 1
        if n is at exiting state
            print move[n.x1][n.y1][n.x2][n.y2]
            return
print "STUCK"
return
```

Problem 4

Algorithm Description

For this problem, I implemented a Union-Find algorithm to find the cost of the minimum-cost F-containing spanning rest of *G*. This algorithm is also referred as Kruskal Algorithm.

General description

Kruskal Algorithm for MST is a greedy algorithm, which means after the graph is created, it is necessary to sort the edges in increasing order of weight.

Firstly, unify all edges that is belongs to F (if the flag equals to 1), and accumulate their weights.

Then, traverse all rest non-F edges in the sorted edges array, check the two vertices on each edge whether they belong to the same set or not.

For those belongs to different set, unify them and accumulate the weights.

In the end, return the total weights.

Correctness

First, the algorithm excludes the special cases where the graph is either contains multiple sub-graphs or contains the circle in F. Since the Kruskal Algorithm is correct, the modified section that incrementally return the weights is correct. Also, the edges in F have been unified and been added up, so it must contain all edges' weights in the F. Then using greedy algorithm to add up all rest edges' weights in increasing order, so the result is correct.

Running time estimate

The Union function takes time complexity of $O(nlogn)$., while Sorting the edges based on their weights takes $O(mlogn)$., so the total time estimate would be $O(mlogn)$..

Pseudocode

```
F_spantree(G(V,E)):
    Weight = 0
    Check connectivity for the input graph
        return -1 if it contains more than one graph
    for edge e(u, v) among all edges in F:
        Union (u, v) if Set or e does not contain a cycle:
            Weight += e.weight
        If set or e contain a cycle:
            Return -1
    Sorted the edges array in increasing order
    For edge e(u, v) among all edges:
        Union (u, v) if Set or e does not contain a cycle:
        Weight += e.weight
Return Weight
```