CS 330 HW 4

Chandini Toleti U29391556
Collaborators: None

1A) ALGORITHM

#uses bfs from lecture; inputs ←undirected graph G and starting node h; outputs← (1) parents, hash table containing key-value pairs (node, parent) that represent shortest parenthood relationships. (2) dist, which is another hash table that stores nodes and their shortest distance from the start node (node, d).

Inputs:
G← undirected Graph in the form of an adjacency list
H←reference to starting node, or home in the scenario provided

```
def unique_cycle(G, h):

    Initialize:
    #run bfs to obtain bfs tree from given graph G and starting node h
    Parents, dist ← bfs (G,h)

    #reverse the dist hash table so keys are distances and pair values are nodes
    Levels ← empty hash table
    For (node,d) in dist.items()do:
        If d not in Levels do:
            Levels[d]= []
        Levels[d].append(node)

    Do:

    #using tree outputted by bfs, we search each level for (1) edges connecting nodes
    on same level or (2) edges connecting nodes on different levels

    #for every level
    For i=1 to i=max(dist.values()) do:
        #For every node in current level
        For n in Levels.get(i,[]) do:
            #For every other node  in the same level
            Stop ←0
            If (i==1) do:
                Stop = 1
            For j=i to j=stop (dec. loop) do:
                For u in Levels.get(j,[]) do:
                    If (parents[n]!= u) and (parents[u]!=n) and (u!=n)do:
                        If (n in G[u]) or (u in G[n]) do:
                            return true;

    return false;
```

CS 330 HW 4

Chandini Toleti U29391556
Collaborators: None

1B) PROOF OF CORRECTNESS

Assertion 1: By observation of the BFS algorithm discussed in lecture, the hash table dist. Represents the least number of steps to each node from the starting node h. Similarly, the hash table parents stores last node visited for each node it considers. Consequently, the table stores nodes and their most closely associated parents. As a result, BFS produces a tree.

Assertion 2: By observation of a BFS tree, we see if an edge is included in the undirected graph and not in the BFS tree, that edge is *nonessential* in reaching the node in question. Therefore, if an edge not specified in the BFS tree exists in the undirected graph between two nodes x and y that are (a) in the same level or (b) between levels that aren't 0 and 1, there exists a unique path to x, and unique path from x to y (the edge found), and a unique path from y back to node h.

Consider the direct proof:

My algorithm calls BFS and utilizes the tree outlined in the first Assertion. Next, I create a reversed hash table. This makes it easier to retrieve nodes for a specific layer. I then consider every layer one at a time. For every layer, I look at every node in that layer.

Next, I check whether we are considering the first layer. I do this because if we look at the first layer of a BFS tree, each of these nodes must be direct children of node h, so there is no point in checking if an edge exists between this first-layer node and h, because if it does it will be present in the BFS tree, and therefore irrelevant. However, we still check for connections between nodes on the first layer, as this would imply a unique cycle. This is why we still loop through all elements of layer 1 twice, the only difference is our altered range (1 to 1 instead of 1 to 0).

Next, If we are not considering the first layer, then for every node in that layer, we check if non-BFS edges exist between that node and every other node in all other layers, including layer 0 where the start node h exists.

All-in-all, if there exists a node outlined in Assertion 2 that guarantees a unique cycle path, my algorithm will find it and return true. Else, it returns False.

1C) SPACE AND TIME COMPLEXITY

<u>SPACE COMPLEXITY</u> To analyze Space complexity, let's look at the inputs, temporary structures, and outputs

INPUTS: The adjacency list representing the undirected graph takes O(V+E) space + the singular node h, which takes O(1) space

CS 330 HW 4

Chandini Toleti U29391556
Collaborators: None

TEMP STRUCTS: BFS returns two hash tables: parents and dist. Each of these structures takes O(V) space since there is a key-value pair for each node. Levels takes another O(V) space since it is the same size as parents.
OUTPUTS: The Boolean and all other elementary data types take O(1) space

In total, the space complexity is given by O(3V+E+1) = O(V+E), as we disregard constants

TIME COMPLEXITY To analyze Time complexity, we analyze every step of the implementation.

1) BFS has a time complexity of O(V + E), given by principles discussed in lecture
2) Constructing a reverse-hash table has a time efficiency of O(V) since we have to consider every node's key-value pair
3) The total runtime for the nested loops is O(V^2) due to the following:
    a. The first two for-loops iterate through every node and therefore run for O(V)
    b. The next two for-loops of j and u in total iterate through all the nodes again, in the worst-case and have complexity O(V)
4) All other operations like comparisons run in constant time O(1)

In total, this algorithm's runtime complexity is O($V^2 + E$)

2A)

I would use BFS, or breadth-first search from lecture. BFS returns two hash tables, *parents* and *dist*. In this instance, I would look at *dist*, which stores key-value pairs of every node and the minimum number of steps to get to that node from the start.

To create *hops* I would reverse the *dist* hash table. This way, the key would be the least number of steps, and the values are corresponding nodes. This could look like the following:

```
hops ← empty hash table
For (node,d) in dist.items() do:
    If d not in hops do:
        hops[d]= []
        hops[d].append(node)
```

CS 330 HW 4

Chandini Toleti U29391556
Collaborators: None

2B.1) ALGORITHM

Inputs:
G ←undirected graph with weights, represented by a nested Hash table of key values pairs (u,v)- where G(u)(v) represent an edge weight
s ← starting node
x ← ending node

```
def best_train_path(hops,G,s,x):
        Initialize:
        Parents, dist ←bfs(G,s) #runs bfs
        Current_best ← empty hash table to store (node, best weight to get there)
        #fill empty hash table with bfs's current_best values
        for level in range(0,dist[x]) do:
                for node in hops[level+1] do:
                        if current_best[node] not in current_best:
                                current_best[node]= [];
                        current_best[node]= G[(node, Parents(node))]

        do:
        for level in range(1,dist[x]) do:
          for u in hops[level] do:
            for v in hops[level+1] do:
              if [(u,v) not in Parents] do:
                curr←Parents[v]
                if G[u][v] + current_best[u] <G[curr][v] + current_best[curr] do:
                  current_best[v] = G[u][v] + current_best[u]
                  Parents[v] = u
                  Parents.delete(u) #deletes u's original child

        #get path from updated Parents hash table to return
        Path← empty array nodes to represent the ideal path we found
        Trav←x, we look at the last node's place in Parents first

        #fill array from Parents hash table
        While (trav!=null) do:
                Path.append(trav)
                Trav = Parents.get(trav)

        #return path, but reversed as we traversed from end to start!
        Return path[::-1]
```

CS 330 HW 4

Chandini Toleti U29391556
Collaborators: None

2B.2) PROOF OF CORRECTNESS

Assertion 1: BFS returns a tree that denotes the shortest path to every node. From any node, there is only one path to the starting node (observation of BFS from lecture)
Assertion 2: The only cross paths being considered in the algorithm are between adjacent levels. Therefore, changing a cross path in the BFS tree cannot increase the number of steps. Since (1) the path is already achieved in the minimum number of steps and (2) changing cross paths do not increase the number of steps, the final tree considered by the algorithm still contains only shortest paths

By working with a BFS tree, the algorithm ensures that every path being considered is the shortest to that node, whether the node is a step to the final node or the final node. This is given by assertion 1 that observes the properties of BFS. Next, the algorithm checks cross paths. Of the valid cross paths that maintain the shortest path (given by assertion 2), only the smallest weighted of these paths will remain in the tree. The condition G[u][v] + current_best[u] <G[curr][v] + current_best[curr] ensures this because it compares the current shortest path to every other possible shortest path and only stores the shortest.

By the direct proof, we see that the algorithm updates the BFS tree such that there is only 1 path from any node x to s, and that path is one of the shortest paths. Of the shortest path, that path is the one of least weight. This is the only path in the final tree and the one returned by the path array in the algorithm.

2B.3) SPACE AND TIME COMPLEXITY

SPACE COMPLEXITY
To analyze space complexity, we observe all inputs, temporary structures, and outputs

Inputs: hops, Parents, and dist have space complexity O(V) since they store at most one key,value per node, G has complexity (V+E) as observed in lectures
Temp. Structs: current_best has space complexity O(V) since it stores at most one weight per node
Outputs: Path is an array of max length V and therefore has space complexity O(V)

All other temporary elementary data structures take constant, or O(1) space. Therefore, the total space complexity is denoted as O(4V+E), which we simplify to O(V + E) disregarding constants.

TIME COMPLEXITY

To analyze time complexity, we observe each step of the algorithm. For all steps, V will indicate the number of nodes, while E indicates the number of edges in the Graph input

1) Running BFS takes O(V + E) time

CS 330 HW 4

Chandini Toleti U29391556
Collaborators: None

2) First nested loops: Initializing the *current_best* hash table looks at all nodes V, and every nodes' edges which runs in O(VE) in the worst case
3) Second nested loops: Looking at every levels' nodes and comparing it to all their edges, again runs in O(VE) time complexity
4) Finally, reconstructing the path runs in linear, or O(V) time since we look at every node once in the worst-case

All other elementary operations run in constant time, and therefore the total runtime complexity can be given as O(V+E + 2VE)= O(VE) in the worst case, since we disregard constants.