CS 514

Assignment 4 – Graph

report4.txt

Q1. You will be implementing breadth-first search to solve a sliding tile puzzle called 8-puzzle (a smaller version of 15-puzzle).

8-puzzle has 8 square tiles, numbered from 1 to 8, arranged in a 3 X 3 square area with one of the 9 cells left vacant. Any tile which is adjacent to the empty cell may be moved into the empty cell. The problem is to take one configuration of the puzzle -- the initial state -- to another configuration -- the goal state -- with a sequence of moves. There is an underlying state space graph,'' where each vertex corresponds to a configuration of the puzzle and there is an edge from u to v if there is a move that can take the configuration from u to v. Each edge also has a label associated with it - D for down move, U for up move, L for left and R for right. The sequence of labels in the path from the initial state to the goal state is called a solution.

For example, the following problem can be solved in 8 moves: D, R, R, U, L, L, D, R.

```
start        goal
1 2 3        1 2 3
4 5 6  ==>   8   4
8 7          7 6 5
```

- **Your function should be called "ShortestPath", which takes a single goal state and a list of initial states as inputs, then return the list of lengths of the shortest paths from the initial states to the goal (one per each initial state). Please note that the initial states may be ordered arbitrarily by the testing program.**
- **Your program uses** single breadth first search **to solve all problems by searching backwards from the goal and collects the solution lengths for all the initial states. Use hashing to check if a node has been already visited.**

Input-output format:

- **Both initial and goal state is a 3x3 matrix represented by a Python list, from left to right, and from top to bottom, where the empty cell is represented by 0. For example, the start and goal states above would be represented as [1,2,3,4,5,6,8,7,0] and [1,2,3,8,0,4,7,6,5] respectively.**
- **There might be multiple solutions (i.e. shortest path) for each problem, thus we ask you to return the length of the shortest path**

Answer-

```
import queue

def ShortestPath(goal, initial):
    def move(state, direction):
        # Helper function to move the empty cell in the given direction if
it's a valid move
        new_state = list(state)
        empty_idx = new_state.index(0)
        if direction == 'U' and empty_idx >= 3:
            new_state[empty_idx], new_state[empty_idx - 3] =
new_state[empty_idx - 3], new_state[empty_idx]
        elif direction == 'D' and empty_idx < 6:
            new_state[empty_idx], new_state[empty_idx + 3] =
new_state[empty_idx + 3], new_state[empty_idx]
        elif direction == 'L' and empty_idx % 3 != 0:
            new_state[empty_idx], new_state[empty_idx - 1] =
new_state[empty_idx - 1], new_state[empty_idx]
        elif direction == 'R' and empty_idx % 3 != 2:
            new_state[empty_idx], new_state[empty_idx + 1] =
new_state[empty_idx + 1], new_state[empty_idx]

        return tuple(new_state)

    def bfs(initial_state, goal_state):
        visited = set()
        q = [(initial_state, 0)]  # (state, steps)
        while q:
            current_state, steps = q.pop(0)  # Pop from the front of the list
            if current_state == goal_state:
                return steps
            if current_state in visited:
                continue
            visited.add(current_state)
            for direction in ['U', 'D', 'L', 'R']:
                new_state = move(current_state, direction)
                if new_state is not None:  # Check if the move is valid
                    q.append((new_state, steps + 1))
        return -1  # No solution found

    goal_state = tuple(goal)
    shortest_paths = []
    for initial_state in initial:
        initial_state = tuple(initial_state)
        path_length = bfs(goal_state, initial_state)
        shortest_paths.append(path_length)

    return shortest_paths

# Example usage:
if __name__ == "__main__":

    # initial_states = [[1, 2, 3, 8, 0, 4, 7, 6, 5]]
    # goal_state = [1, 3, 4, 8, 6, 2, 7, 0, 5]
    #
    # initial_states = [[1,2,3,8,0,4,7,6,5]]
    # goal_state = [2,8,1,0,4,3,7,6,5]

    # initial_states = [
```

```
    #     [1, 2, 3, 8, 0, 4, 7, 6, 5]
    # ]
    #
    # goal_state = [2, 8, 1, 4, 6, 3, 0, 7, 5]

    # initial_states = [
    #     [1,3,4,8,0,5,7,2,6],
    #     [2,3,1,7,0,8,6,5,4],
    #     [2,3,1,8,0,4,7,6,5],
    #     [2,8,3,1,0,4,7,6,5],
    #     [8,7,6,1,0,5,2,3,4]
    #     ]
    #
    # goal_state = [1,2,3,8,0,4,7,6,5]
    #
    # initial_states = [
    #     [8,6,7,2,5,4,3,0,1],
    #     [6,4,7,8,5,0,3,2,1],
    #     [4,1,2,0,8,7,6,3,5],
    #     [1,6,2,5,7,3,0,4,8]
    # ]
    #
    # goal_state = [1,2,3,4,5,6,7,8,0]

    initial_states = [
        [8, 0, 6, 5, 4, 7, 2, 3, 1],
        [6, 4, 1, 3, 0, 2, 7, 5, 8],
        [1, 5, 8, 3, 2, 7, 0, 6, 4],
        [3, 2, 8, 4, 5, 1, 6, 7, 0],
        [0, 3, 5, 4, 2, 8, 6, 1, 7],
        [7, 2, 5, 3, 1, 0, 6, 4, 8]
    ]

    goal_state = [0, 1, 2, 3, 4, 5, 6, 7, 8]

    print(ShortestPath(goal_state, initial_states))
```

Time Complexity Analysis –

A. Converting the input states into tuples: The initial and goal states are converted into tuples, which have a constant time complexity of O(1).

B. BFS Function: The BFS function is the core of the algorithm. It explores the state space starting from the initial state and continues until it reaches the goal state or determines that there is no solution. The time complexity of this function depends on the size of the state space.

1. Enqueuing the initial state and steps (initialization): O(1)
2. While loop: This loop runs until the queue is empty or a solution is found. In the worst case, it explores all possible states in the state space.
3. Checking if the current state is the goal state: O(1)
4. Checking if the current state is in the "visited" set: O(1)
5. Adding the current state to the "visited" set: O(1)
6. Iterating through all possible moves ('U', 'D', 'L', 'R') and generating new states: In the worst case, this loop iterates 4 times.
7. Checking if the move is valid: O(1)

8. Enqueuing the new state and steps: O(1)
9. Returning -1 if no solution is found: O(1)

In the worst case, where the BFS explores the entire state space, the time complexity of the BFS function can be considered as O(b^d), where - b is the branching factor (the number of possible moves, which is 4 in this case) and d is the depth of the shallowest solution in the state space.

C. ShortestPath Function: The ShortestPath function is responsible for finding the shortest path from each initial state to the goal state. It iterates through the list of initial states and calls the BFS function for each initial state.
1. Iterating through the list of initial states: O(n), where n is the number of initial states.
2. Calling the BFS function for each initial state: O(b^d) as explained above.

Overall Time Complexity:

The overall time complexity of the ShortestPath function can be expressed as O(n * b^d), where - n is the number of initial states, b is the branching factor (4 in this case) and d is the depth of the shallowest solution in the state space.

Q2. Suppose that the only negative edges are those that leave the starting node s. Does Dijkstra's algorithm find the shortest path from s to every other node in this case? Justify your answer carefully.

Answer –

In the given scenario where only negative edges exist leaving the starting node – node 1, applying Dijkstra's algorithm does not guarantee the discovery of the correct shortest path to all other nodes. Let's illustrate this with a specific example of finding the shortest path from node 1 to node 3 using Dijkstra's algorithm.

Dijkstra's algorithm begins by initializing the distances from node 1 to node 3 and node 1 to node 2 as infinity, as it starts with no prior knowledge of distances. The algorithm proceeds to explore the neighboring nodes, which in this case are node 3 and node 2. During this process, it updates the distance to node 3 as -1 and the distance to node 2 as -5. Both node 3 and node 2 are marked as visited.

The issue here is that the algorithm erroneously concludes that the shortest path from node 1 to node 3 is -1, which is not the correct shortest path. The actual shortest path is node1 -> node 2-> node3, with a total distance of -4. The discrepancy arises because Dijkstra's algorithm is designed for graphs with non-negative edge weights, and it operates under the assumption that, as it iteratively explores nodes, it always selects the node with the smallest distance to be visited next. However, the presence of negative edge weights departing from the starting node S can lead to counterintuitive outcomes.

In this particular case, the negative weight in the graph causes the algorithm to fail in correctly identifying the shortest path. This conclusion is reached through a proof by contradiction, highlighting that Dijkstra's algorithm is not suitable for scenarios with negative edge weights.

Q3. Give an O(n^3) algorithm that takes a directed graph as input and returns the length of the shortest cycle in the graph where n is the number of nodes.

To discover the shortest cycle within a directed graph, we employ Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) which is an algorithm for finding the shortest paths

between nodes in a graph.It returns a search tree for all the paths the given node can take. It operates by establishing a search tree that encompasses all possible paths originating from a specific node and its execution time typically depends on the number of nodes and edges in the graph.

But Dijkstra's algorithm can yield paths that do not form cycles. To eliminate these non-cycle paths, we scrutinize each edge in the generated paths and remove those that do not include the starting node. This way, only cycle-related paths remain.

The cycle-related paths are then adjusted to account for the complete cycle distance. This adjustment entails adding the distance from the destination back to the source, which reflects the entire cycle.

This process is repeated for every node in the graph. In other words, we apply Dijkstra's algorithm successively to each node within the graph.

Dijkstra's algorithm has a known worst-case time complexity of O(n^2). Since we iterate this algorithm for all n nodes, the overall worst-case complexity of our approach becomes O(n^3).

In pseudo-code, the algorithm is summarized as follows:

```
def ShortestCycle(graph):
   cycle_length_list= []  # Initialize a list to store cycle lengths

   for s in graph.nodes:
      distances = Dijkstra(graph, s)  # Apply Dijkstra's algorithm
      cycle_lengths = []

      for u in graph.nodes:
         for v in graph.nodes:
            if u != s and v != s:  # Skip nodes connected to the source node
               cycle_length = distances[u] + graph.edgeWeight(u, v) + distances[v]
               cycle_lengths.append(cycle_length)

      shortest_cycle_length = min(cycle_lengths)
      cycle_length_list.append(shortest_cycle_length)

   return min(cycle_length_list)  # Return the length of the shortest cycle
```

Pseudocode explanation -

g. Append `shortest_cycle_length` to the `cycle_length_list` for the current source node `s`.
3. After processing all nodes, return the minimum value from the `cycle_length_list`, which represents the length of the shortest cycle in the graph.

So, this algorithm systematically processes each node, applies Dijkstra's Algorithm, filters out non-cycle paths, and adapts the distances. It ultimately assembles and returns the shortest cycle length within the directed graph in O(n^3) time complexity.

Q4. You are given a strongly connected directed graph G = (V, E) with positive edge weights. Give an efficient algorithm for finding the shortest paths between all pairs of nodes with the restriction that they all must pass through node A.

This issue can be divided into two distinct tasks:

1. Determine the shortest routes from all nodes to node A.
2. Compute the shortest path from A to all other nodes.

Both of these tasks can be addressed using Dijkstra's algorithm.

To solve the second task, Dijkstra's algorithm is applied. The pseudocode provided in the CLRS textbook maintains information about the distance from A and the parent nodes for each node. Dijkstra's algorithm is designed to find the shortest paths from a single source in a weighted, directed graph G = (V, E), where all edge weights are nonnegative. It is assumed that the weight of each edge $(u, v) \in E$ is greater than or equal to zero.

The algorithm works by managing a set S, which contains vertices for which the final shortest-path distances from the source s have already been determined. The process involves repeatedly selecting the vertex vertex $u \in V - S$ that has the smallest shortest-path estimate, adding u to the set S, and relaxing all edges leaving u. To facilitate this, a minimum-priority queue Q of vertices is used, with keys based on their d values.

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        For each vertex v ∈ G.Adj[u]
                RELAX(u,v,w)
```

Pseudocode explanation -

Initialize the shortest-path data structures 1. Initialize an empty list called `cycle_length_list` to store cycle lengths.

2. Iterate over each node `s` in the graph:

a. Use Dijkstra's algorithm to compute the shortest distances from node `s` to all other nodes in the graph, and store these distances in the `distances` array.

b. Create an empty list `cycle_lengths` to store the lengths of potential cycles involving nodes other than `s`.

c. Nest two loops to iterate over all pairs of nodes `u` and `v` in the graph.

d. Check that `u` and `v` are not equal to the source node `s` to avoid cycles that include the source node.

e. Calculate the length of a potential cycle by summing the distance from `s` to `u`, the weight of the edge from `u` to `v`, and the distance from `v` back to `s`. Append this length to the `cycle_lengths` list

f. Find the shortest cycle length among all potential cycles and store it in `shortest_cycle_length`.

1. for the graph G and source node s using INITIALIZE-SINGLE-SOURCE(G, s).
2. Initialize two sets, S (empty at the beginning) and Q (containing all vertices in G).
3. While there are still vertices in the set Q:
    a. Select the vertex u with the minimum shortest-path estimate by calling EXTRACT-MIN(Q).

b. Add u to the set S.
c. For each vertex v in the adjacency list of u (G.Adj[u])- Attempt to relax the edge (u, v) by calling the RELAX(u, v, w) function.

To tackle the first task, the approach involves iterating through the original graph and constructing a new graph by reversing the edges. Once this new graph is created, the same Dijkstra's algorithm procedure is applied to find the distances from all nodes to node A.

Thank You !