jongminl-311-hw3

Jongmin Lee

April 2024

1 Question 1

```
If(person = 0){
    for(i = 1 to n){ //for loop #1
        if((T[0][i]) == 1){
            person = i
        }
    }
    for(i = 0 to n){ //for loop #2
        if(i != person && T[i][person] == 0 || T[person][i] == 1){
        return -1
        }
    }
    return person
}

```

Since 2 for loop is not nested which means each for loop's runtime is O(n) it's runtime is just O(n)

2 Question 2

```
function find_optimal_starting_island(L, B):
    //L is the set of islands
    //B is the set of bi-directional bridges,
    represented as a set of tuples (island1, island2)

    //Create a graph using adjacency list representation
    graph = initialize_adjacency_list(L, B)

    //Initialize variables to track the optimal starting island
    max_reachable = 0
    optimal_starting_island = None
    optimal_reachable_islands = empty_set()

    //Iterate through each island in L
    for island in L:
        //Perform BFS from the current island
        reachable_islands = bfs(graph, island)

```

```
        //Calculate the number of reachable islands
        num_reachable = length(reachable_islands)

        //Update optimal starting island if the current
        island reaches more islands
        if num_reachable > max_reachable:
            max_reachable = num_reachable
            optimal_starting_island = island
            optimal_reachable_islands = reachable_islands

//Return the optimal starting island and the set of reachable islands
return optimal_starting_island, optimal_reachable_islands

//Helper function: BFS from a starting island
function bfs(graph, start_island):
    //Initialize a set to track visited islands
    visited = empty_set()

    //Initialize a queue for BFS
    queue = initialize_queue()
    enqueue(queue, start_island)

    //Mark the starting island as visited
    add_to_set(visited, start_island)

    //Perform BFS
    while queue is not empty:
        //Dequeue the current island
        current_island = dequeue(queue)

        //Iterate through each neighbor of the current island
        for neighbor in graph[current_island]:
            //If the neighbor is not visited
            if neighbor not in visited:
                //Mark the neighbor as visited and enqueue it
                add_to_set(visited, neighbor)
                enqueue(queue, neighbor)

    //Return the set of visited islands
    return visited

//Helper function: Initialize the adjacency list representation of the graph
function initialize_adjacency_list(L, B):
    graph = empty_adjacency_list()

    //Iterate through each bridge in B
    for bridge in B:
        island1, island2 = bridge

        //Add each island to the adjacency list of the other island
        add_to_adjacency_list(graph, island1, island2)
        add_to_adjacency_list(graph, island2, island1)
    //Return the graph
    return graph
```

For the runtime of algorithm, building the adjacency list from the list of islands and bridges will take O(—L— + —B—) time, where —L— is the number of islands, and —B— is the number of bridges.

3 Question 3

```
function isSchedulingPossible(courses, prerequisites, simultaneousCourses):
//Initialize graph
graph = defaultdict(list)

//Build the graph based on prerequisites
for course, prereq in prerequisites.items():
    for p in prereq:
        graph[p].append(course)

/Merge simultaneous courses
for group in simultaneousCourses:
    representative = group.pop()  # Choose one course as the representative

    //Merge the group
    for course in group:
        //Redirect edges to and from the representative course
        for neighbor in graph[course]:
            graph[representative].append(neighbor)
        for prereq in prerequisites.get(course, set()):
            graph[prereq].append(representative)

        //Remove course from the graph
        if course in graph:
            del graph[course]

    //Update courses set
    courses.difference_update(group)
    courses.add(representative)

//Perform cycle detection using DFS
def hasCycle(course, visited, stack):
    //If the course is being processed in the current DFS, a cycle is found
    if stack[course]:
        return True
    //If the course is already visited, skip it
    if visited[course]:
        return False

    //Mark the course as visited and add it to the recursion stack
    visited[course] = True
    stack[course] = True

    //Recur for all neighbors of the course
    for neighbor in graph[course]:
        if hasCycle(neighbor, visited, stack):
```

```
            return True

    //Remove the course from the recursion stack
    stack[course] = False
    return False

//Initialize visited and stack dictionaries
visited = defaultdict(bool)
stack = defaultdict(bool)

//Check each course using DFS for cycle detection
for course in courses:
    if not visited[course]:
        if hasCycle(course, visited, stack):
            //If a cycle is detected, return False
            return False

//If no cycles were found, return True
return True
```

Building the graph based on prerequisites takes O(V + E), where V is the number of courses and E is the number of prerequisite relationships. In the worst case, each course is processed once, so this part of the algorithm can be approximated as O(V + E).

4 Question 4

```
function minSemestersToComplete(courses, prerequisites, simultaneousCourses):
//Initialize graph and in-degree counts
graph = defaultdict(list)
inDegree = defaultdict(int)

//Step 1: Build graph from prerequisites
for course in courses:
    inDegree[course] = 0
for course, prereq in prerequisites.items():
    for p in prereq:
        graph[p].append(course)
        inDegree[course] += 1

//Step 2: Merge simultaneous courses into super-courses
for group in simultaneousCourses:
    //Choose a representative course for each simultaneous group
    representative = group.pop()

    //Redirect edges to and from the representative course
    for course in group:
        //Redirect outgoing edges
        for neighbor in graph[course]:
            graph[representative].append(neighbor)
            inDegree[neighbor] += 1

```

```
            //Redirect incoming edges
            for prereq in prerequisites.get(course, set()):
                graph[prereq].append(representative)
                inDegree[representative] += 1

            //Remove merged course from graph and in-degree
            if course in graph:
                del graph[course]
            if course in inDegree:
                del inDegree[course]

        //Update the courses set to include the representative course
        courses.difference_update(group)
        courses.add(representative)

    //Step 3: Perform topological sort using Kahn's algorithm
    zeroInDegreeQueue = deque()
    for course in courses:
        if inDegree[course] == 0:
            zeroInDegreeQueue.append(course)

    topologicalOrder = []
    while zeroInDegreeQueue:
        course = zeroInDegreeQueue.popleft()
        topologicalOrder.append(course)

        for neighbor in graph[course]:
            inDegree[neighbor] -= 1
            if inDegree[neighbor] == 0:
                zeroInDegreeQueue.append(neighbor)

    //Step 4: Calculate the longest path ending at each course
    longestPath = defaultdict(int)
    for course in topologicalOrder:
        currentLongestPath = 0

        //Calculate the longest path ending at this course
        for neighbor in graph[course]:
            currentLongestPath = max(currentLongestPath, longestPath[neighbor])

        //Update the longest path length for the current course
        longestPath[course] = currentLongestPath + 1

    //Step 5: Determine the minimum number of semesters required
    //Minimum number of semesters required is the maximum value in longest path
    return max(longestPath.values())
```

The algorithm's runtime complexity is O(V + E), where V is the number of courses and E is the number of prerequisite relationships. This is because the topological sort and calculation of the longest path both operate in linear time relative to the size of the graph.