CS 600 A - Advanced Algorithms – Homework 6

Name: Deepali Nagwade | 20013393

Question 13.7.4: Bob loves foreign languages and wants to plan his course schedule for the following years. He is interested in the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA 169. The course prerequisites are.

i.) LA15: (None)
ii.) LA16: LA 15
iii.) LA22: (None)
iv.) LA31:LA 15
v.) LA32:LA16, LA31
vi.) LA126: LA22, LA32
vii.) LA127: LA 16
viii.) LA141:LA22, LA 16
ix.) LA 169: LA32.

Find the sequence of courses that allows Bob to satisfy all the prerequisites.

Answer: We will use topological sorting for this scenario. This approach helps determine the correct sequence of courses by considering their prerequisites.

Topological sorting is a technique used to arrange items or tasks in a specific order based on their dependencies or prerequisites. It is commonly applied to solve problems where there are dependencies between different elements, and a valid sequence that satisfies those dependencies is found out.

find the sequence of courses that allows Bob to satisfy all the prerequisites, first, identify the courses that have no prerequisites. These courses can be taken first.

- Based on the given information, the courses with no prerequisites are LA15 and LA22. So, Bob can take LA15 and LA22 as his first courses.
- Next, look for the courses that have LA15 as a prerequisite. From the options given, LA16 and LA31 have LA15 as a prerequisite. Bob can take LA16 and LA31 after completing LA15.
- Consider the courses that have both LA16 and LA31 as prerequisites. LA32 and LA141 require both LA16 and LA31. Bob can take LA32 and LA141 after completing LA16 and LA31.
- Now, the courses that require LA32 as a prerequisite. LA126 and LA169 have LA32 as a prerequisite. Bob can take LA126 and LA169 after completing LA32.
- Lastly, LA127 remaining, which only requires LA16 as a prerequisite. Bob can take LA127 after completing LA16.

So, the final Sequence would be:

LA15 -> LA22 -> LA16 -> LA31 -> LA32 -> LA141 -> LA126 -> LA169 -> LA127

Question 13.7.19: suppose G is a graph with n vertices and m edges. Describe a way to represent G using O(n + m) space so as to support in O(log n) time an operation that can test, for any two vertices v and w, whether v and w are adjacent.

Answer: Create an array of size n, where each element is an empty list. This array will represent the adjacency list for graph G. The index of each element in the array will correspond to a vertex in the graph.

For example, if we have 5 vertices in graph G, we will create an array of size 5:

adjacencyList = [[], [], [], [], []]

For each edge (u, v) in the graph, add **v** to the adjacency list of **u** and add **u** to the adjacency list of **v**.

Let's say we have the following edges in graph G: (1, 2), (1, 3), (2, 4)

We would update the adjacency list as follows:

adjacencyList = [2, 3, 1, 4, 1, 2]

- The adjacency list for vertex 1 contains vertices 2 and 3 because it is adjacent to them.
- The adjacency list for vertex 2 contains vertices 1 and 4 because it is adjacent to them.
- The adjacency list for vertex 3 contains vertex 1 because it is adjacent to it.
- The adjacency list for vertex 4 contains vertex 2 because it is adjacent to it.

To test whether two vertices **v** and **w** are adjacent, we perform a binary search on the adjacency list of vertex **v**.

Let's say we want to check if vertex 1 and vertex 4 are adjacent.

We would perform a binary search on the adjacency list of vertex 1, which is 2,
We check the middle element, which is 2.

- Since 2 is not equal to 4, we compare 4 with 2 and find that it is greater.
- So, we search the right half of the list.
- The right half contains only one element, which is 3.
- Since 3 is not equal to 4, we compare 4 with 3 and find that it is greater.
- We reach the end of the list without finding 4.
- Therefore, vertex 1 and vertex 4 are not adjacent.

The binary search takes O(log n) time, as we divide the list in half at each step until we find the element or reach the end of the list.

This representation uses O(n + m) space because we store an array of size n to represent the adjacency lists, and each adjacency list can have at most m elements (the number of edges in the graph).

The adjacency check operation has a time complexity of O(log n) because it involves performing a binary search on the adjacency list of vertex v, which has at most n elements.
The graph G can be represented using an adjacency list, which is an array of size n where each element is a list containing adjacent vertices for that vertex. This representation requires O(n + m) space, where n is the number of vertices and m is the number of edges.

To test if two vertices v and w are adjacent, we perform a binary search on the adjacency list of vertex v. If w is found in the adjacency list, v and w are adjacent; otherwise, they are not adjacent. This operation takes O(log n) time.

Overall, this representation allows us to determine adjacency between two vertices in O(log n) time and uses O(n + m) space.

Question 13.7.37: Tamarindo University and many other schools worldwide are doing a joint project on multimedia. A computer network is built to connect these schools using communication links that form a free tree. The schools decided to install a file server at one of the schools to share data among all the schools. Since the transmission time on a link is dominated by the link setup and synchronization, the cost of a data transfer is proportional to the number of links used. Hence, it is desirable to choose a "central" location for the file server. Given a free tree T and a node v of T, the eccentricity of v is the length of a longest path from v to any other node of T. A node of T with minimum eccentricity is called a center of T.

a) Design an efficient algorithm that, given an n-node free tree T, computes a center of T.
b) Is the center unique? If not, how many distinct centers can a free tree have?

Answer:

a) Algorithm: The reason for this algorithm is that the center of a free tree is the node with the minimum eccentricity, which is the midpoint of the longest path in the tree.

```
Insert all leaf nodes into FIFO queue Q.
traverse (using DFS or BFS) from the root node, and insert all leaf nodes in Q.
while Q has more than 2 nodes
        node n <-- Q.firstnode
        p <-- n.parent
        p.numChildrens = p.numChildrens - 1
        if p.numChildrens=0
                insert p into Queue
Return Q
```

Number of nodes in Q(1 or 2) is/are the centers of the Tree T.

Time complexity of the algorithm is O(n). Traversing the tree is done in O(n). The "while loop" processes each node only one time. Hence, the overall complexity is O(n).

b) The center is not unique in general. A free tree can have one or two centers. The number of distinct centers depends on the tree's structure and whether it has an even or odd number of nodes:
   - If the tree has an even number of nodes, there will be two distinct centers.
   - If the tree has an odd number of nodes, there will be only one distinct center.

In the case of an odd number of nodes, the single center is the midpoint of the longest path in the tree. In the case of an even number of nodes, there are two centers, which are equidistant from each other and lie on either side of the longest path.

So, the number of centers in a free tree can be either one or two, depending on the tree's size and structure.

Question 14.7.11: There is an alternative way of implementing Dijkstra's algorithm that avoids use of the locator pattern but increases the space used for the priority queue, Q, from O(n) to O(m) for a weighted graph, G, with n vertices and m edges. The main idea of this approach is simply to insert a new key-value pair, (D[v], v), each time the D[v] value for a vertex, v, changes, without ever removing the old key-value pair for v. This approach still works, even with multiple copies of each vertex being stored in Q, since the first copy of a vertex that is removed from Q is the copy with the smallest key. Describe the other changes that would be needed to the description of Dijsktra's algorithm for this approach to work.

Answer: The alternative way of implementing Dijkstra's algorithm that avoids using the locator pattern but increases the space used for the priority queue from O(V) to O(E) for a weighted graph G with V vertices and E edges involves inserting new key-value pairs each time the distance (or cost) value for a vertex changes. This approach eliminates the need to remove old key-value pairs. However, this approach requires some adjustments to the standard Dijkstra's algorithm. Here's how it works:

- Initialize a priority queue (min-heap) Q, initially empty, to store pairs of vertices and their distance values (vertex, distance).
- Initialize a distance array dist[], where dist[v] represents the shortest distance from the source vertex to vertex v. Initially, set dist[source] = 0 and dist[v] = infinity for all other vertices.
- Insert (source, 0) into the priority queue Q.
- While Q is not empty, do the following:
  - Remove the vertex u with the smallest distance from Q. If there are multiple copies of u with different distances, pick the one with the smallest distance.
  - For each neighbor v of u:
    a) Calculate the new distance d = dist[u] + weight(u, v).
    b) If d is less than dist[v], update dist[v] to d and insert (v, d) into Q.
- Continue until all vertices have been processed or until the destination vertex has been reached.

This modified approach works because you always maintain the smallest distance value for each vertex in the priority queue, and when a shorter path is found, the updated (v, d) pair is inserted into the queue without removing the previous, now less important, key-value pair.

The running time of Dijkstra's algorithm using this approach with a heap-based priority queue (min-heap) is O((V + E) * log(V)) where V is the number of vertices and E is the number of edges in the graph. The O(V + E) term arises from the fact that each vertex is processed once, and for each vertex, you relax its outgoing edges. The log(V) term comes from the insertion and deletion operations in the priority queue. The choice of a min-heap ensures efficient log(V) time complexity for these operations, which is crucial to the overall time complexity of the algorithm.

Question 14.7.17: In a side-scrolling video game, a character moves through an environment from, say, left-to- right, while encountering obstacles, attackers, and prizes. The goal is to avoid or destroy the obstacles, defeat, or avoid the attackers, and collect as many prizes as possible while moving from a starting position to an ending position. We can model such a game with a graph, G, where each vertex is a game position, given as an (, y) point in the plane, and two such vertices, v and w, are connected by an edge, given as a straight-line segment, f there is a single movement that connects v and w. Furthermore, we can define the cost, c(e), of an edge to be a combination of the time, health points, prizes, etc., that it costs our character to move along the edge e (where earning a prize on this edge would be modeled as a negative term in this cost). Path P, in G is monotone if traversing P involves a continuous sequence of left-to-right movements, with no right-to-left moves. Thus, we can model an optimal solution to such a side-scrolling computer game in terms of finding a minimum-cost monotone path in the graph, G, that represents this game. Describe and analyze an efficient algorithm for finding a minimum-cost monotone path in such a graph, G

Answer:

Description:

Finding a minimum-cost monotone path in the graph G representing a side-scrolling video game can be approached as a specialized case of the Shortest Path Problem, specifically the Shortest Path in a

Directed Acyclic Graph (DAG). In this case, graph G should be transformed into a DAG to take advantage of dynamic programming to efficiently find the minimum-cost monotone path.

Here's an algorithm to find the minimum-cost monotone path:

- **Create the DAG**: Transform the original graph into a DAG, creating a sequence of vertices from (x, 0) to (x, y) for each (x, y) in G.
- **Initialize Distances**: Initialize dist[i] to represent the minimum cost to reach the vertex (x, i), with dist[0] set to 0 and all others to infinity.
- **Topological Sort:** Perform a topological sort on the DAG for left-to-right processing.
- **Relax Edges:** For each vertex (x, y), consider outgoing edges to (x', y') with x' > x and y' >= y. Update dist[x'] if the cost through (x, y) is smaller.
- **Find Minimum-Cost Monotone Path:** The minimum cost to reach any (x, y) represents the minimum cost to reach that level (y-coordinate) in a monotone path.
- **Backtrack for Path:** Start from the rightmost vertex at the desired y-coordinate and backtrack using updated dist values, choosing edges that minimize the cost while moving left.

Complexity Analysis:

The algorithm's complexity depends on the number of vertices in the transformed DAG. The number of vertices is proportional to the maximum height the character can reach (the maximum y-coordinate). Let's denote this maximum height as H.

- Creating the DAG takes O(V), where V is the number of vertices in the original graph.
- Topological sorting takes O(V).
- Relaxing edges takes O(E) in total.
- Backtracking to find the path takes O(H).

Overall, the algorithm's time complexity is O(V + E + H), and the space complexity is O(V + H) to store the distances and backtrack path. This algorithm efficiently finds the minimum-cost monotone path in a side-scrolling video game graph.

Question 14.7.20: Suppose you are given a timetable, which consists of the following:

A set A of n airports, and for each airport a ∈ A, a minimum connecting time c(a)

A set F of m flights, and the following, for each flight f ∈ F:

- Origin airport a1(f) ∈ A
- Destination airport a2(f) ∈ A
- Departure time t1(f)
- Arrival time t2(f).

Describe an efficient algorithm (Pseudocode) for the flight scheduling problem. In this problem, we are given airports a and b, and a time t, and we wish to compute a sequence of flights that allows one to arrive at the earliest possible time in b when departing from a at or after time t. Minimum connecting times at intermediate airports should be observed.

What is the running time of your algorithm as a function of n and m?

Answer: The flight scheduling problem can be solved using Dijkstra's algorithm with some modifications to account for the minimum connecting times at intermediate airports.

Here's the pseudocode for the algorithm:

```
Function flightScheduling(startAirport, endAirport, startTime): Function
flightScheduling(startAirport, endAirport, startTime):
    Create an empty queue Q, dictonary dist and prev
    dist[startAirport] = startTime
    Q.push((startAirport, startTime, None))  # None indicates no previous flight
    while Q is not empty:
            (currentAirport, currentTime, prevFlight) = Q.pop()
            if currentAirport == endAirport:              break
            if currentTime < dist[currentAirport]:     continue
           for each flight f from currentAirport:
               if currentTime + c(a1(f)) <= t1(f):
                  nextTime = max(t1(f) + c(a1(f)), t2(prevFlight))
                  if nextTime < dist[a2(f)]:
                      dist[a2(f)] = n    extTime
                      prev[a2(f)] = f
                      Q.push((a2(f), nextTime, f))
    if endAirport not reached:
       return "No feasible route."
    flight = prev[endAirport]
    while flight is not None:
       route.append(flight)
       flight = prev[a1(flight)]
    route.reverse()
   return route
```

In this pseudocode, `c(a)` represents the minimum connecting time at airport `a`.

The algorithm iterates through flights using a priority queue and updates the minimum arrival time at each airport based on the given constraints.

The running time of the algorithm is determined by the number of flights and the priority queue operations. Let `n` be the number of airports and `m` be the number of flights.

In the worst case, each flight will be added to the priority queue once, and each flight will be extracted from the priority queue once.

The priority queue operations take O(log m) time.