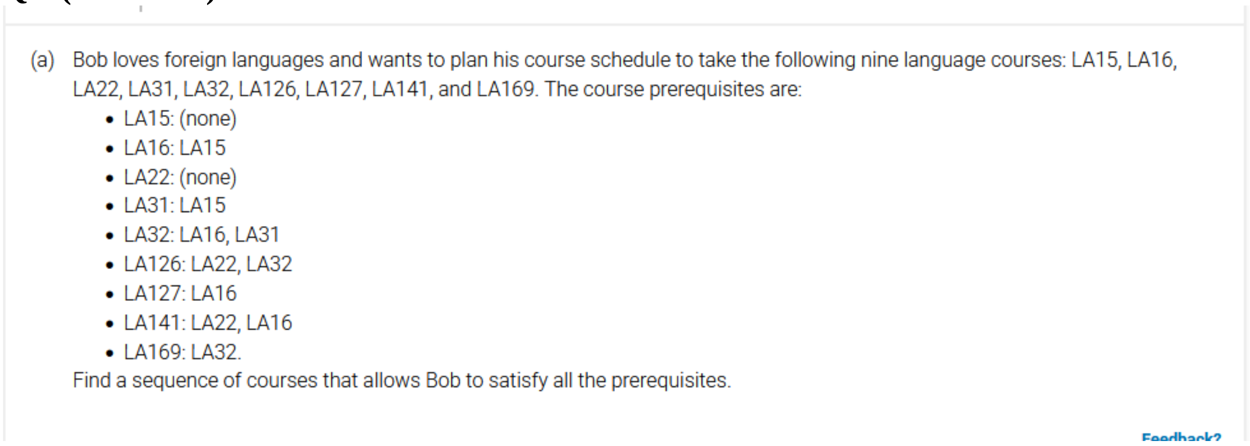
Ali Abdullah Ahmad
20031246
CS 600

Q1. (No. 13.8.4)



(a) Bob loves foreign languages and wants to plan his course schedule to take the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169. The course prerequisites are:

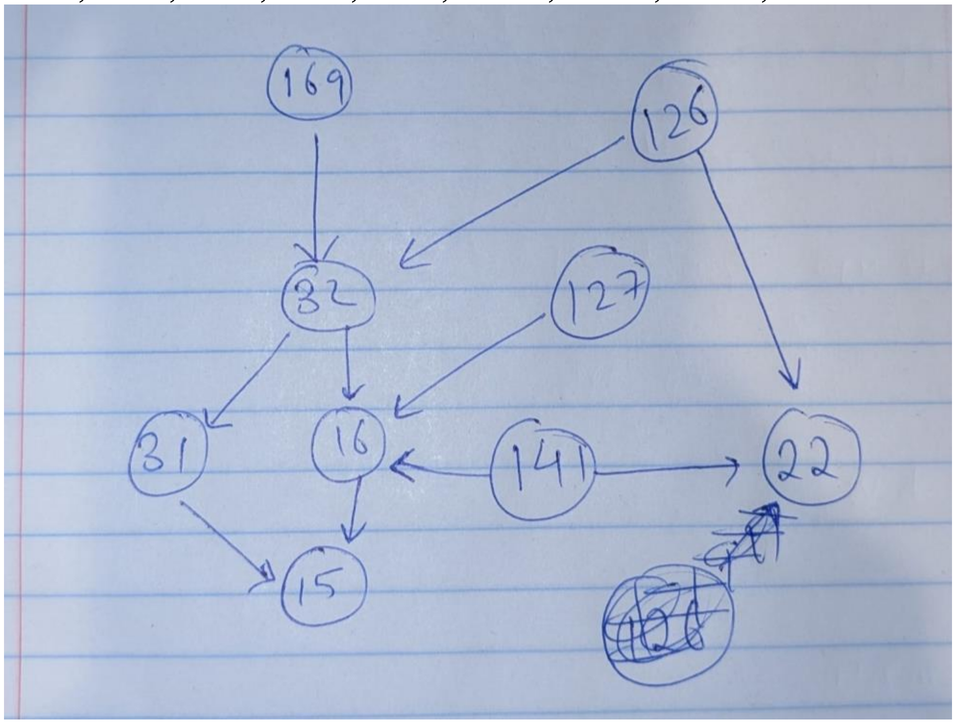
- LA15: (none)
- LA16: LA15
- LA22: (none)
- LA31: LA15
- LA32: LA16, LA31
- LA126: LA22, LA32
- LA127: LA16
- LA141: LA22, LA16
- LA169: LA32.

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

Ans.

We can use the graph data structure to represent the subjects. Nodes will represent the subjects and the incoming edges to the node represent the prerequisites for the adjacent courses.

LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, LA169



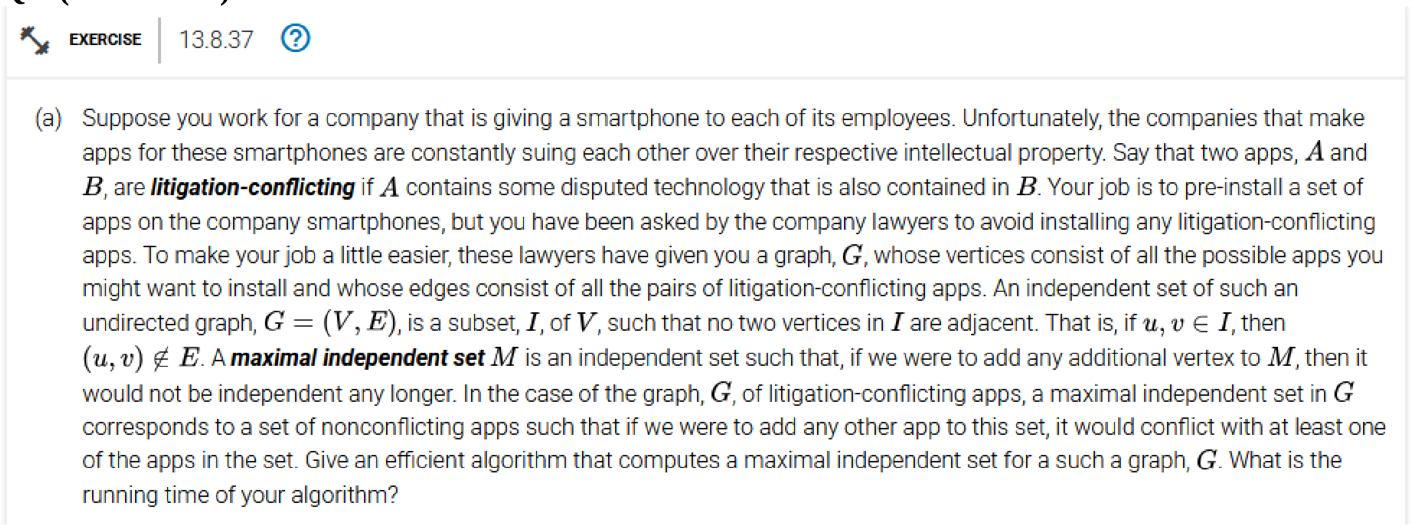
Q2. (No. 13.8.19)
Suppose G is a graph with n vertices and m edges. Describe a way to represent G using O(n+m) space so as to support in O(log n)time an operation that can test, for any two vertices v and w, whether v and w are adjacent.

Ans.

1. Use an adjacency list to represent the graph.
2. For each vertex v, store its adjacent vertices in a balanced binary search tree (BST) or a sorted dynamic array.
3. The space complexity will be: O(n+m) where n = vertices, m = edges.
4. To check if v and w are adjacent:
    - Perform a binary search in v's adjacent list.
    - This will take O(logd) time, where d is the degree of v.
    - In the worst case d=n−1, so the time complexity is: O(logn)

This approach uses O(log n) time for adjacency testing.

Q3. (No. 13.8.37)



EXERCISE 13.8.37

(a) Suppose you work for a company that is giving a smartphone to each of its employees. Unfortunately, the companies that make apps for these smartphones are constantly suing each other over their respective intellectual property. Say that two apps, A and B, are ***litigation-conflicting*** if A contains some disputed technology that is also contained in B. Your job is to pre-install a set of apps on the company smartphones, but you have been asked by the company lawyers to avoid installing any litigation-conflicting apps. To make your job a little easier, these lawyers have given you a graph, G, whose vertices consist of all the possible apps you might want to install and whose edges consist of all the pairs of litigation-conflicting apps. An independent set of such an undirected graph, $G = (V, E)$, is a subset, I, of V, such that no two vertices in I are adjacent. That is, if $u, v \in I$, then $(u, v) \notin E$. A ***maximal independent set*** M is an independent set such that, if we were to add any additional vertex to M, then it would not be independent any longer. In the case of the graph, G, of litigation-conflicting apps, a maximal independent set in G corresponds to a set of nonconflicting apps such that if we were to add any other app to this set, it would conflict with at least one of the apps in the set. Give an efficient algorithm that computes a maximal independent set for a such a graph, G. What is the running time of your algorithm?

Ans.
A simple greedy algorithm for computing a maximal independent set (MIS) is:

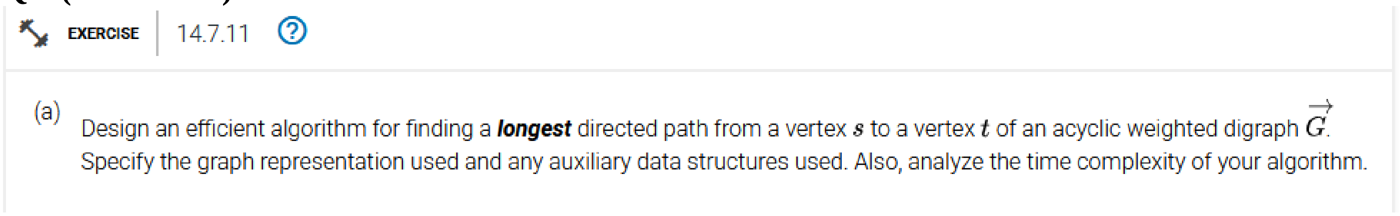
1. Initialize an empty set M.
2. Iterate through all vertices in some order.
3. For each vertex v, add it to M if none of its neighbours are already in M.
4. Stop when all vertices have been considered.

Running Time Analysis

- The naive implementation iterates over all vertices and checks each vertex's neighbours.
- If the graph is represented using an adjacency list, this takes O(V+E) time, where V is the number of vertices and E is the number of edges.
- If using an adjacency matrix, the time complexity would be O(V^2)

Thus, the greedy algorithm runs in O(V+E) with an adjacency list, which is quite efficient.

Q4. (No. 14.7.11)



EXERCISE 14.7.11

(a) Design an efficient algorithm for finding a ***longest*** directed path from a vertex s to a vertex t of an acyclic weighted digraph $\vec{G}$. Specify the graph representation used and any auxiliary data structures used. Also, analyze the time complexity of your algorithm.

Ans.

Since the graph is acyclic, we can solve the problem efficiently using topological sorting and dynamic programming (DP).

Algorithm:

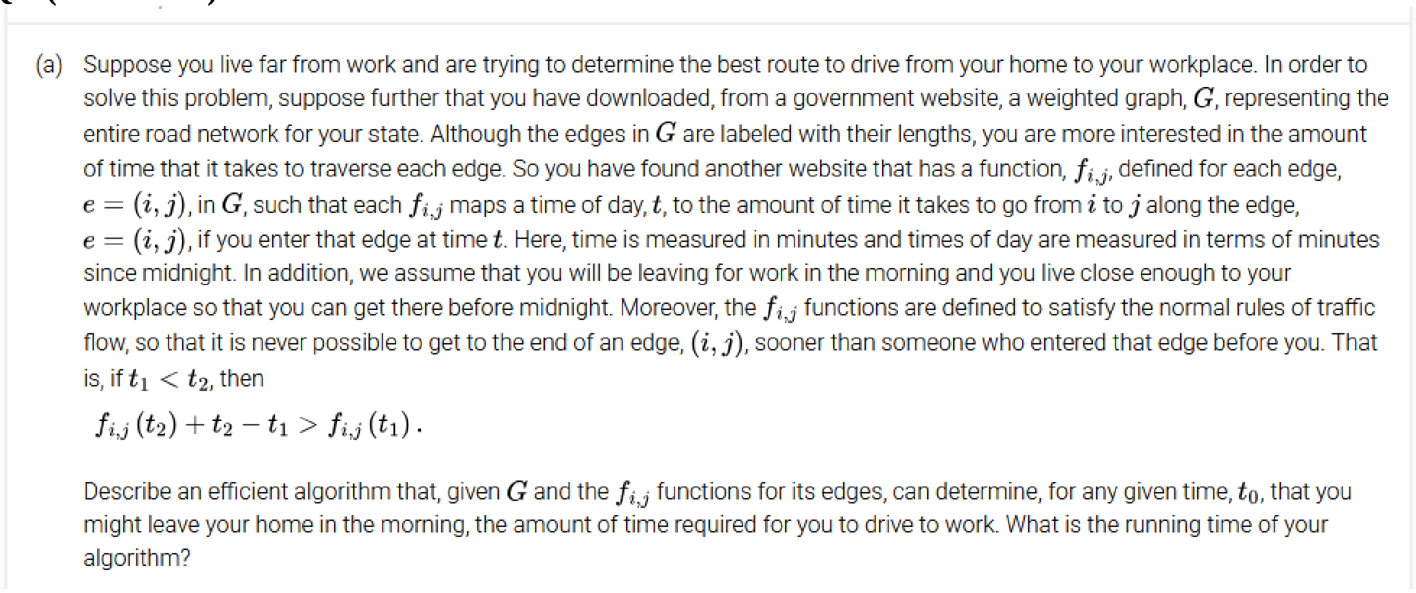
1. Topological Sort
   - Perform a topological sort on the DAG. This gives a linear ordering of vertices such that for every directed edge (u,v) , u appears before v.
   - This can be done using Kahn's algorithm (BFS-based) or DFS in O(V+E) time.
2. Initialize Distance Array
   - Create an array dist of size V, where dist[i] stores the longest path distance from s to i.
   - Set dist[s] = 0 and all other dist[i] = - infinity (or a very small number).
3. Relax Edges in Topological Order
   - Process each vertex in topological order.
   - For each vertex u, update distances for all its neighbours v: dist[v]=max(dist[v],dist[u]+weight(u,v))
4. Return dist[t] (the longest path from s to t).

Time Complexity Analysis:

1. Topological Sort: O(V+E)
2. Relaxation of Edges: O(V+E) (each edge is processed once)

Total Complexity: O(V+E)

Q5. (No. 14.7.17)



(a) Suppose you live far from work and are trying to determine the best route to drive from your home to your workplace. In order to solve this problem, suppose further that you have downloaded, from a government website, a weighted graph, G, representing the entire road network for your state. Although the edges in G are labeled with their lengths, you are more interested in the amount of time that it takes to traverse each edge. So you have found another website that has a function, $f_{i,j}$, defined for each edge, $e = (i, j)$, in G, such that each $f_{i,j}$ maps a time of day, t, to the amount of time it takes to go from i to j along the edge, $e = (i, j)$, if you enter that edge at time t. Here, time is measured in minutes and times of day are measured in terms of minutes since midnight. In addition, we assume that you will be leaving for work in the morning and you live close enough to your workplace so that you can get there before midnight. Moreover, the $f_{i,j}$ functions are defined to satisfy the normal rules of traffic flow, so that it is never possible to get to the end of an edge, (i, j), sooner than someone who entered that edge before you. That is, if $t_1 < t_2$, then

$$f_{i,j}(t_2) + t_2 - t_1 > f_{i,j}(t_1).$$

Describe an efficient algorithm that, given G and the $f_{i,j}$ functions for its edges, can determine, for any given time, t_0, that you might leave your home in the morning, the amount of time required for you to drive to work. What is the running time of your algorithm?

Ans.

Modified Dijkstra's Algorithm

Since edge weights (travel times) depend on the time of arrival, a modified Dijkstra's algorithm works efficiently.

Steps:

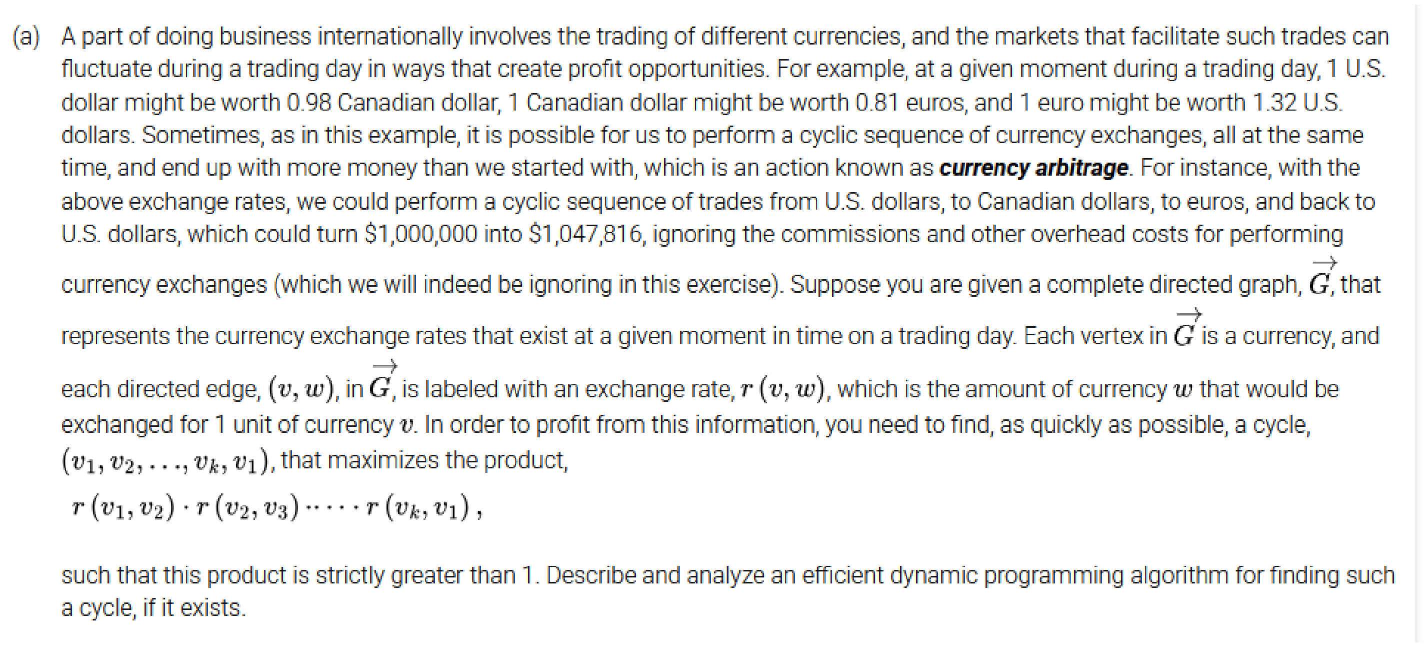
1. Initialize Data Structures
    - Use a priority queue (min-heap) to store pairs (t,v), where t is the earliest time we can reach vertex v.
    - Maintain an array arrival_time[v], initialized to infinity for all nodes except the start node, which is set to t0.
2. Apply Dijkstra's Algorithm with Time-Dependent Weights
    - Start from the home node at time t0.
    - Extract the node u with the earliest arrival time from the priority queue.
    - For each outgoing edge (u,v), compute the time-dependent travel time:
      new_arrival_time = current_time + fu,v(current_time)
    - If this new time is earlier than arrival_time[v], update it and push (new_arrival_time,v) into the priority queue.
3. Stop when Destination is Reached
    - Once the workplace node is extracted from the queue, return the earliest arrival time.

Time Complexity Analysis

- Each vertex is processed once: O(V)
- Each edge is relaxed once: O(E)
- Priority queue operations: O((V+E)logV) (with Fibonacci heap: O(E+VlogV)

Overall complexity: O((V+E)logV), which is the same as standard Dijkstra but adapted for time-dependent weights

Q6. (No. 14.7.20)



(a) A part of doing business internationally involves the trading of different currencies, and the markets that facilitate such trades can fluctuate during a trading day in ways that create profit opportunities. For example, at a given moment during a trading day, 1 U.S. dollar might be worth 0.98 Canadian dollar, 1 Canadian dollar might be worth 0.81 euros, and 1 euro might be worth 1.32 U.S. dollars. Sometimes, as in this example, it is possible for us to perform a cyclic sequence of currency exchanges, all at the same time, and end up with more money than we started with, which is an action known as ***currency arbitrage***. For instance, with the above exchange rates, we could perform a cyclic sequence of trades from U.S. dollars, to Canadian dollars, to euros, and back to U.S. dollars, which could turn $1,000,000 into $1,047,816, ignoring the commissions and other overhead costs for performing currency exchanges (which we will indeed be ignoring in this exercise). Suppose you are given a complete directed graph, $\vec{G}$, that represents the currency exchange rates that exist at a given moment in time on a trading day. Each vertex in $\vec{G}$ is a currency, and each directed edge, (v, w), in $\vec{G}$, is labeled with an exchange rate, $r(v, w)$, which is the amount of currency w that would be exchanged for 1 unit of currency v. In order to profit from this information, you need to find, as quickly as possible, a cycle, $(v_1, v_2, \ldots, v_k, v_1)$, that maximizes the product,

$$r(v_1, v_2) \cdot r(v_2, v_3) \cdots \cdot r(v_k, v_1),$$

such that this product is strictly greater than 1. Describe and analyze an efficient dynamic programming algorithm for finding such a cycle, if it exists.

Ans.

The Bellman-Ford algorithm is ideal for this problem because it can detect negative-weight cycles in O(VE) time complexity, where V is the number of currencies and E is the number of exchange rates.
Steps

1. Convert exchange rates into edge weights using w (v,w) = −log r(v,w).
2. Run the Bellman-Ford algorithm from an arbitrary starting currency.
3. If after V−1 relaxations, we can further relax an edge, a negative-weight cycle exists, indicating an arbitrage opportunity.

Complexity Analysis

- Graph Representation: Since the graph is complete, it has O(V^2) edges
- Bellman-Ford Execution: Runs in O(VE) = O(V^3)
- Final Complexity: O(V^3), which is efficient for a reasonable number of currencies