COM S 311 HOMEWORK 3

NATHAN TUCKER (NJTUCKER@IASTATE.EDU)

This assignment represents my own work in accordance with University regulations. Collaboration done with Haadi Majeed (hmajeed@iastate.edu) and Matthew Hoskins (mgh@iastate.edu)

- Nathan Tucker

Homework 3 Submission

(1) The Elevator Problem

At a prestigious computer science department, there is an elevator that is serving n floors. The elevator has k buttons, each annotated by an integer that, when pressed, travels the integer number of floors. Note that when pressing a button that would move to a floor not served by the elevator, the elevator will not move.

For example, the elevator could serve 72 floors and have buttons annotated with integers -5 and 2. If the elevator is on floor 17 and pressing the button with integer -5 will move the elevator to floor 12. Then pressing two times the button with integer 2 will move the elevator to level 16.

It is a high sport culture for the inhabitants of this building to find out if it is possible to travel with the elevator from floor i to floor j by pressing a sequence of buttons. If so, then the inhabitants want to know a shortest such sequence and how many shortest sequences there are.

Write an efficient dynamic programming algorithm in pseudo-code that answers these questions, give a brief justification of its correctness, and analyze its runtime.

The input parameters for your algorithm should include (i) $n \in \mathbb{N}$ representing the number of floors the elevator is serving, (ii) $b_1, \ldots, b_k \in \mathbb{Z}$ ($k \in \mathbb{N}$) representing the buttons with their integer values, and (iii) $i, j \in \{1, \ldots, n\}$ where we want to know whether the elevator can travel from floor i to floor j. As output, the algorithm should state "NO" when moving from floor i to floor j is not possible. Otherwise, the algorithm should output (i) a shortest sequence of integers describing the buttons pressed to move the elevator from floor i to floor j, and (ii) the number of such sequences.

```
1  ShortestSequence(i, j, n, b[])
       Input: current floor i, destination floor j, floors served n, array of possible buttons b
2      count = 0
3      let currentFloors be an array tracking the currently available floors
4      if i == j then
5          return count, currentArray
6      return ShortestSequenceRec(i, j, n, b[], count, currentArray[])
```

```
1  ShortestSequenceRec(i, j, n, b[], count, currentArray[])
       Input: current floor i, destination floor j, floors served n, array of possible buttons b,
              the current counter, the current array of buttons
2      if i == j then
3          return count + 1, currentArray
4      returnVal = "No"
5      if visitedFloors.contains(i) || i > n || i < 1 then
6          return returnVal
7      else
8          visitedFloors.add(i)
9      for k = 1 to b.length do
10         returnVal = ShortestSequenceRec(i + b[k], j, n, b[], count,
            currentArray.append(b[k]))
11     return returnVal
```

For this algorithm, the idea is "can we reach the destination floor from the current floor given the number of buttons we have now" and that was the base case of my algorithm that I started from. This also stuck out to me as a graph-theory problem, so I incorporated some elements of that into my solution with tracking visited and non-visited floors. If it is possible to reach the destination floor given the served floors and buttons, we can return the sequence of buttons and the number of sequences. Because the algorithm searches every value of b[], as well as looping dependent on n, we can say that the algorithm has an overall runtime of $O(k*n)$ where k is the number of buttons and n is the number of floors being served.

(2) The Cookie Game

Riley and Morgan play the following cookie game. Given is one set of n red cookies and another set of m green cookies. At every turn, a player must eat two cookies from one set and one cookie from the other set (i.e., either (i) two green cookies and one red cookie, or (ii) two red cookies and one green cookie). The player who cannot move loses. Assuming Riley will begin the game, which player will win?

Write an efficient dynamic programming algorithm in pseudo-code that decides whether a winning strategy for one of the players exists, give a brief justification of the correctness of your algorithm, and analyze its runtime.

Your algorithm's input parameters should include $n, m \in \mathbb{N}_0$ representing the given numbers of green and red cookies, respectively. As output, the algorithm should state whether Riley or Morgan will win, or if there is no winning strategy for either player.

```
1  Alg1(A)
       Input: Array of integers of length n
2      let dp[1...m][1...n] be a 2d array holding the result of the cookie game
3      for i = 1 to m do
4          for j = 1 to n do
5              dp[i][j] = ∞
6              if i < 3||j < 3 then
7                  dp[i][j] = 0
8              else
9                  if dp[i - 2][j - 1] == 1 and dp[i - 1][j - 2] == 1 then
10                     dp[i][j] = 1
11                 else
12                     dp[i][j] = 0
13     if dp[m][n] == ∞ then
14         return "No Such Strategy Exists"
15     if dp[m][n] == 0 then
16         return "Morgan Wins!"
17     return "Riley Wins!"
```

For this algorithm, all we need to do is first store all the possible game outcomes of the cookie game from 1 to m and 1 to n cookies in a 2d array, just as in other, similar problems like "Nim" or the Rod Cutting Problem. All that's done is a nested loop from first 1 to m then 1 to n, which is iterating over the array, then at each coordinate i and j, we can calculate the depth to the end of the game. All it's doing is, at this current number of cookies in the iteration, are the 2 outcomes that the player could take going to end the game? One example is at 4 red and 2 green. If the next move is to take 2 green and 1 red, the next move would for sure be a game over situation, meaning the player who has taken that move would win. So if either of the situations would end in a next-move loss, we would not want to take that, and assign it a 0.

Once every possible outcome has been tested, we can see that at m, n, or the number of cookies we've been given, if the number of turns is 0, it will be a Morgan victory, as the last move in the game will be the losing move. If it is 1, then Riley will win, as the last move in the game will be hers, and will be the winning move. If no depth can be calculated, then the function does nothing to the output result, and it stays at ∞, meaning there is no winning strategy. Because all we do is loop over the values of m and n in a nested loop, the number of iterations will be n * m, meaning a runtime of $O(m * n)$.

(3) The Constrained LCS Problem

Let X and Y be strings over an alphabet Σ. For $\Gamma \subseteq \Sigma$ we define the Γ-*constrained longest common subsequence* of X and Y to be a longest common subsequence of X and Y that does NOT contain any of the characters in Γ.

Give a detailed proof that the problem of finding a constrained longest common subsequence exhibits optimal substructure.

There are 3 cases in which the solution provides optimal substructure for a constrained LCS

- if $X_m = Y_n$ then $\Gamma_k = X_m = Y_n$ and Γ_{k-1} is a constrained LCS of X_{m-1} and Y_{n-1}
- if $X_m \neq Y_n$ then $\Gamma_k \neq X_m$ and implies that Γ is a constrained LCS of X_{m-1} and Y
- if $X_m \neq Y_n$ then $\Gamma_k \neq Y_n$ and implies that Γ is a constrained LCS of X and Y_{n-1}

We can now prove these individually.

- Let us assume that $\Gamma_k \neq X_m$. We can append $X_m = Y_n$ to Γ to obtain a constrained LCS of X and Y of length $k+1$. This contradicts our hypothesis that Γ of length k is the constrained LCS of X_{m-1} and $Yn-1$

  Let us now assume that S is a constrained LCS of X_{m-1} and Y_{n-1} with a length greater than $k-1$. If we append $X_m = Y_n$ to S, we can observe that it is a constrained LCS of X and Y with length greater than k. This contradicts our assumption that Γ is a constrained LCS of X and Y.
- Proof 2: If $\Gamma_k \neq X_m$ then Γ is a constrained LCS of X_{m-1} and Y. Now we show that Γ is a constrained LCS of X_{m-1} and Y. Let us first assume that S is a constrained LCS of X_{m-1} and Y with length greater than k. This also means that S is a constrained LCS of X and Y with length greater than k. This contradicts our assumption that Γ is a constrained LCS of X and Y.
- Proof 3: If $\Gamma_k \neq Y_n$ then Γ is a constrained LCS of X and Y_{n-1}. Let us assume that S is a constrained LCS of X and Y_{n-1} with length greater than k. This also means that S is a constrained LCS of X and Y with length greater than k. This contradicts our assumption that Γ is a constrained LCS of X and Y.

Because of the above we can conclude that finding a constrained longest common subsequence ex-hibits optimal substructure

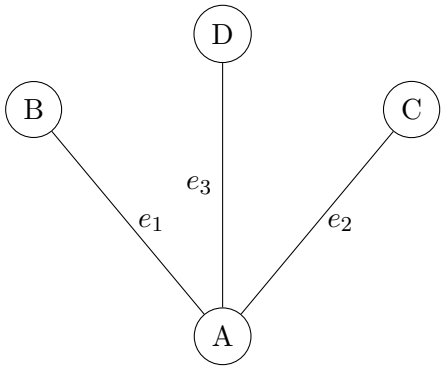
(4) Minimum Spanning Trees

Let $G = (V, E)$ be a connected and undirected graph with a weight function $c\colon E \to \mathbb{R}$.

Given an arbitrary vertex $v \in V$, is it true that an edge incident to v with the least weight always belongs to some minimum spanning tree of G?

Give a thorough proof showing the correctness of your answer.

To dissect this problem, we first need to set up what it means for vertex to be incedent. Graph theory states that a vertex incedent to the edge of the vertex is one of the two vertexes the edge connects to. An incedence is a pair where it is a vertex and in an edge incedent. Two edges are "incedent" if they share a common vertex.



In the above graph, edges e1, e2, and e3 are incident edges at the vertex A.
In the problem stated, a given vertex V that connects to the least weight, we need to state if it always belongs to some MST of G or not.

First, we need to make the assumption that the MST will be some subgraph of a given graph G that connects all the minimum possible weights where the edges are all uniquely weighted. We can Prove by Contradiction:

Let the edge be represnted by (U, V). Let is also assume that (U, V) does not belong to a minimum weight spanning tree, but an MST still exists from U and V that does not traverse the edge. If we include this edge (U, V) in the final graph, we see that a cycle forms. From this cycle, the edge with the largest weight should be removed. This edge will not be the edge (U, V) as we know it is already the edge of least weight in the graph G. This creates a new MST. An edge has been removed to eliminate cycles, such that we now have a valid MST. Additionally, all the vertexes are still accessible with minimum weight, meaning it is a valid MST.

We can now calulate the weight of this new MST by

weight of (original spanning tree) - weight of (original spanning tree -
(the removed edge - (U, V)))

This is due to the representation of the new MST weight is

weight of (original spanning tree - (edge removed - (U, V)))

represnted by the left side of the equation for the weight difference. Armed with this knowledge now, the weight of the new MST is less than the weight of the original MST. This proves that the weight of the new MST is, in fact, an MST for graph G when (U, V) is added which is incedent to $v \in V$.

$\therefore$ The smallest weighted edge of some V is always in the MST