Homework 7

Date: 2023-08-01 07:17
type: #Homework
Category: 📌cs 311 MOC

Dynamic Programming & Negative edge weights

Question 1

> Let A be a set of non-negative integers $(a_1, a_2, \ldots, a_n)$. Write an algorithm that gets A as input and determines if A has a subset A_1 such that $\sum_{a \in A_1} a = \frac{a_1+a_2+\cdots+a_n}{2}$ I.e, sum of elements in A_1 is exactly half of $(a_1 + \ldots a_n)$. Your algorithm must use the *dynamic programming paradigm*. First write the recurrence relation, explain the correctness of the recurrnce relation. Design an algorithm based on the recurrence relation. State and derive the time bound of the algorithm. Your algorithm should not use recursion.

First, let's denote the sum of all elements in A as S. The problem can be solved with dynamic programming by recognizing that it is essentially asking whether it can be partitioned into two subsets with the sum of each subset equal to $S/2$.

If S is not divisible by 2, we can immediately return false since it is impossible to split A into two subsets with equal sums.

We can create a 2D boolean table `dp[][]`, where `dp[i][j]` will be true if there is a subset of $a_1, \ldots, a_i$ with sum equal to j. We initialize the table with all entries as false except `dp[0][0]`, which is true, since there is a subset with sum 0, that is, the empty set.

The recurrence relation can be described as follows:

- For each number, we have two options, we either include it in our subset or we exclude it.
- If we exclude the number, then the sum remains the same i.e., if `dp[i-1][j]` was true, then `dp[i][j]` will also be true.
- If we include the number, then `dp[i][j]` will be true if `dp[i-1][j-a[i]]` was true.

Recurrence Relation:

$$dp[i][j] = dp[i-1][j] \text{ or } dp[i-1][j-a[i]], \text{for all i in } [1, n] \text{ and for all j in } [1, S/2].$$

Correctness of Recurrence Relation:

The variable i iterates through each element of the set A, while j ranges from 0 to $S/2$. At each step, we are checking if we can form a sum j using the elements up to i in the set A. If either the sum j is already formed using $i-1$ elements or it can be formed by including the current element $a[i]$ in the sum formed by the previous $i-1$ elements $(j - a[i])$, we say that the sum j can be formed using the elements up to i.

```
hasSubsetWithHalfSum(A){
   n = length(A)
```

```
    S = sum(A)

    if S mod 2 != 0:
        return false

    S = S/2

    dp = new boolean[n+1][S+1]
    for i = 0 to n:
        dp[i][0] = true
    for i = 1 to S:
        dp[0][i] = false

    for i = 1 to n:
        for j = 1 to S:
            dp[i][j] = dp[i-1][j]
            if j >= A[i-1]:
                dp[i][j] = dp[i][j] or dp[i-1][j-A[i-1]]

    return dp[n][S]
}
```

Time Complexity:

The time complexity is $O(n \cdot S)$, where n is the number of elements in the set A and S is the sum of all the elements. The reason is that we have to fill up the `dp` table for all subsets of A (which are n) and all sums up to $S/2$.

Question 2

> You are in a rectangular maze organized in the form of $M \cdot N$ cells/locations. You are starting at the upper left corner (grid location: (1, 1)) and you want to go to the lower right corner (grid location: (M, N)). From any location, you can move either to the right or to the bottom, or go diagonal. I.e., from (i, j) you can move to $(i, j+1)$ or $(i+1, j)$ or to $(i+1, j+1)$. Cost of moving right or down is 2, while the cost of moving diagonally is 3. The grid has several cells that contain diamonds of whose value lies between 1 and 10.
> I.e, if you land in such cells you earn an amount that is equal to the value of the diamond in the cell. Your objective is to go from the start corner to the destination corner. Your profit along a path is the total value of the diamonds you picked minus the sum of the all the costs incurred along the path. Your goal is to find a path that maximizes the profit.
> Write a dynamic programming algorithm to address the problem. Your algorithm must take a 2-d array representing the maze as input and outputs the maximum possible profit. Your algorithm need not output the path that gives the maximum possible profit. First write the recurrence relation to capture the maximum profit, explain the correctness of the recurrence relation. Design an algorithm based on the recurrence relation. State and derive the time bound of the algorithm. Your algorithm should not use recursion.

We define a 2D array `dp[][]` where `dp[i][j]` represents the maximum profit we can get from position $(1,1)$ to (i,j).

First, let's define our base case:
$dp[1][1] = maze[1][1] - 2$ (since we need to pay 2 to move from (1,1) to (1,1))

Next, for all other cells, we have 3 possible ways to arrive at a cell (i, j) :

1. From the cell just to its left, i.e., $(i, j-1)$. The cost of this move is 2.
2. From the cell just above it, i.e., $(i-1, j)$. The cost of this move is 2.
3. From the cell on its upper-left, i.e., $(i-1, j-1)$. The cost of this move is 3.

We want to take the path that maximizes the profit, so for each cell (i, j), we will choose the path that gives the maximum value after subtracting the cost.

So the recurrence relation becomes:

$$dp[i][j] = max(dp[i-1][j] - 2,\ dp[i][j-1] - 2,\ dp[i-1][j-1] - 3) + maze[i][j]$$

For all i in $[2, M]$ and for all j in $[2, N]$.

Correctness of the Recurrence Relation:

At each position (i, j), we are making an optimal decision considering all the possible ways we can arrive at this position and picking the path that gives the maximum profit. Since each $dp[i][j]$ is calculated using previously computed values, we ensure that we are making optimal decisions at each step.

```
maxProfit(maze){
   M = len(maze)
   N = len(maze[0])

   dp = new int[M+1][N+1]
   dp[1][1] = maze[1][1] - 2

   for i = 2 to M:
       dp[i][1] = dp[i-1][1] - 2 + maze[i][1]
   for j = 2 to N:
       dp[1][j] = dp[1][j-1] - 2 + maze[1][j]

   for i = 2 to M:
       for j = 2 to N:
           dp[i][j] = max(dp[i-1][j] - 2, dp[i][j-1] - 2, dp[i-1][j-1] - 3) + maze[i]
[j]

   return dp[M][N]
}
```

Time Complexity:

The time complexity of the algorithm is $O(M \cdot N)$ as we are visiting each cell exactly once.

Question 3

> Write an algorithm to find the number of shortest paths from a given source vertex s to a given destination vertex t in a graph where the weights of the edges can be positive or negative integers. You can assume that the graph does not contain any negative weight cycles. Give a brief justification that the algorithm works. State the runtime of the algorithm accompanied by a brief justification.

This problem can be solved using a modified version of the Bellman-Ford algorithm to handle graphs with negative weights. The Bellman-Ford algorithm is typically used for finding shortest paths from a single source vertex to all other vertices in the graph, so we can modify it here to count the number of shortest paths between a given source and target vertex.

```
countShortestPaths(graph, s, t){
   V = number of vertices in the graph
   distance = new array of size V, initialized to infinity
   count = new array of size V, initialized to 0

   distance[s] = 0
   count[s] = 1

   for i from 1 to V-1:
       for each edge (u, v) with weight w in graph:
           if distance[u] + w < distance[v]:
               distance[v] = distance[u] + w
               count[v] = count[u]
           else if distance[u] + w == distance[v]:
               count[v] += count[u]

   return count[t]
}
```

Justification for correctness:

The `distance[]` array keeps track of the shortest path from the source vertex s to each vertex in the graph. The `count[]` array keeps track of the number of shortest paths from s to each vertex.

In the case where the addition of an edge leads to an equal distance, it means that there's another path of the same length, and we should increment the count for that vertex. The count array will store the number of shortest paths from the source to each vertex, and by returning `count[t]`, we get the number of shortest paths from s to t.

Time Complexity:

The Bellman-Ford algorithm has a time complexity of $O(V \cdot E)$, where V is the number of vertices and E is the number of edges in the graph. This is because the algorithm consists of $V - 1$ iterations and in each iteration, it relaxes all E edges in the graph. Thus, the time complexity of this modified version of the Bellman-Ford algorithm is also $O(V \cdot E)$.
