CS 577 # Assignment 7 – Dynamic Programming Spring 2024

Answer the questions in the boxes provided on the question sheets. If you run out of room for an answer, add a page to the end of the document.

Name: Ki Min Kang Wisc id: 908 404 5062

Dynamic Programming

Do **NOT** write pseudocode when describing your dynamic programs. Rather give the Bellman Equation, describe the matrix, its axis and how to derive the desired solution from it.

1. *Kleinberg, Jon. Algorithm Design (p.313 q.2).*

   Suppose you are managing a consulting team and each week you have to choose one of two jobs for your team to undertake. The two jobs available to you each week are a low-stress job and a high-stress job.

   For week i, if you choose the low-stress job, you get paid ℓ_i dollars and, if you choose the high-stress job, you get paid h_i dollars. The difference with a high-stress job is that you can only schedule a high-stress job in week i if you have no job scheduled in week $i-1$.

   Given a sequence of n weeks, determine the schedule of maximum profit. The input is two sequences: $L := \langle \ell_1, \ell_2, \ldots, \ell_n \rangle$ and $H := \langle h_1, h_2, \ldots, h_n \rangle$ containing the (positive) value of the low and high jobs for each week. For Week 1, assume that you are able to schedule a high-stress job.

   (a) Show that the following algorithm does not correctly solve this problem.

Algorithm: JOBSEQUENCE

Input : The low (L) and high (H) stress jobs.
Output: The jobs to schedule for the n weeks
for *Each week i* **do**
  if $h_{i+1} > \ell_i + \ell_{i+1}$ **then**
    Output "Week i: no job"
    Output "Week i+1: high-stress job"
    Continue with week i+2
  else
    Output "Week i: low-stress job"
    Continue with week i+1
  end
end

We will show that Jobsequence algorithm doesn't work by a following counter example.
Let the sequences for low-stress and high-stress jobs of three weeks.

$L = (1,1,1)$ - Low stress job payments
$H = (4,1,10)$ - High stress job payments.

Using the Jobsequence algorithm:
Week1 : Choose a low-stress job ($L=1$) because H_2 is not greater than L_1+L_2
Week2 : Skip, because H_3 is greater than L_2+L_3.
Week3 : Choose a high-stress job ($H=10$)
Total = 1+0+10 = 11

Optimal Result:
Week1 : Choose a high-stress job ($H=4$)
Week2 : Skip
Week3 : Choose a high-stress job ($H=10$)
Total = 4+0+10 = 14

The example above shows that the algorithm does not correctly solve the problem.

(b) Give an efficient algorithm that takes in the sequences L and H and outputs the greatest possible profit.

We will use a dynamic programming algorithm that uses an array S of size n+2 to store the maximum profit for each week. The index of S starts at -1 to accommodate the base cases.

The algorithm uses the recurrence relation, known as the Bellman Equation to fill the array.
For each week i, compute:
$S[i] = \max(S[i-1] + l_i, S[i-2] + h_i)$

The equation calculates the maximum profit for week i by comparing the profit of taking a low-stress job this week versus taking a high-stress job this week.

The optimal profit after n week is found at S[n].

(c) Prove that your algorithm in part (c) is correct.

We will prove the algorithm by the induction.

Base Case 1: When i is -1 or 0,
There are no jobs to schedule, so the optimal profit is 0.

Base Case 2: When i is 1,
We should look at two scenarios: not working or picking either the low-stress or high-stress job.
The best option yields the maximum value of l_1 or h_1, and this will set the stage for the entire schedule.

Inductive Step: For each subsequent week i, the algorithm decides whether to take a low-stress job for the current week, upon the best schedule for i-1 weeks, or take a high-stress job (only possible if the prior week was rest), thus building upon the best schedule for i-2 weeks. We assume by induction that the values for S[i-1] and S[i-2] have been correctly determined to reflect the maximum profits
Once the array S is fully computed the algorithm backtracks to construct the optimal schedule.

Therefore, the algorithm above is correct.

2. *Kleinberg, Jon. Algorithm Design (p.315 q.4).*

Suppose you're running a small consulting company. You have clients in New York and clients in San Francisco. Each month you can be physically located in either New York or San Francisco, and the overall operating costs depend on the demands of your clients in a given month.

Given a sequence of n months, determine the work schedule that minimizes the operating costs, knowing that moving between locations from month i to month $i+1$ incurs a fixed moving cost of M. The input consists of two sequences N and S consisting of the operating costs when based in New York and San Francisco, respectively. For month 1, you can start in either city without a moving cost.

(a) Give an example of an instance where it is optimal to move at least 3 times. Explain where and why the optimal must move.

Assume that we have a 4-month schedule with the following operating costs:

moving cost $M = 2$

New York costs $N = [1, 6, 1, 6]$ 14

San Francisco costs $S = [5, 2, 5, 2]$ 14

Optimal schedule with at least three moves:

Starting in New York $1+2+1+2+3M = 12$

The schedule staying in one city the whole time or involving at least one move but less than 3 would be more expensive.

Therefore, the schedule with three moves is optimal in this example.

(b) Show that the following algorithm does not correctly solve this problem.

Algorithm: WorkLocSeq

Input : The NY (N) and SF (S) operating costs.
Output: The locations to work the n months
for *Each month i* **do**
  if $N_i < S_i$ **then**
    Output "Month i: NY"
  else
    Output "Month i: SF"
  end
end

Assume the operating costs are given as:

NewYork costs $NY = (1, 5)$

San Fransico Coss $SF = (5, 1)$

with moving cost $M = 10$

According to the algorithm, we will stay first month in NY and move to SF next month ($NY_2 > SF_2$)

The overall cost including M is 12, while staying in the city whole time is 6.

Therefore, the counter example shows that the algorithm does not work correctly.

(c) Give an efficient algorithm that takes in the sequences N and S and outputs the value of the optimal solution.

To minimize the operating costs over n months, we will use 2xn matrix S to track the optimal costs.
Each cell S[j][i] represents the minimum cost up to month i with j, indicating the city (1 for NY, 2 for SF).

1. Initialize the matrix S where S[1][1] = N_1 and S[2][1] = S_1, the operating costs for each city in the first month, respectively.
2. Use the Bellman Equations to fill the matrix for i=2 to n:
   - For NY in month i = S[1][i] = N_i + min(S[1][i-1], S[2][i-1]+M)
   - For SF in month i = S[2][i] = S_i + min(S[1][i-1]+M, S[2][i-1])
3. The optimal schedule value after n months is given by min(S[1][n], S[2][n])

(d) Prove that your algorithm in part (c) is correct.

We will prove the algorithm by induction.

Base case: For i=1, there are two possible schedules starting in NY or SF.
The cost of starting in NY is N_1 and the cost of starting in SF is S_1.
The algorithm selects the city with the lower cost as the optimal starting point.

Inductive Step: Assuming that the optimal schedule plans us in NY in month i, the cost will include either the optimal schedule for NY from the previous month or for SF from the previous month. This reasoning is based on the idea that the optimal cost for any month i in NY or SF must account for the potential move from the other city.
The Bellman equation ensures that we take the minimum of the two possible costs for each month, yielding the optimal value for S[1][i] and S[2][i] at every step.

The optimal cost for the schedule over n months is then the minimum value between S[1][n] and S[2][n], considering the cheapest option between staying in the current city or moving from the other city, including the moving cost.

Therefore, the algorithm provides the minimum total cost for the entire period by induction

3. *Based on: Kleinberg, Jon. Algorithm Design (p.333, q.26).*

Consider the following inventory planning problem. You are running a company that sells trucks and have good predictions for how many trucks you sell per month. Let d_i denote the number of trucks you expect to sell in month i. If you have unsold trucks any given month, you can store up to s of them in inventory to sell the next month instead. Storage cost is described by a function $c(i, j)$, where i is the month and j is the number of trucks stored. Every month you can buy any number of trucks, and each order has an associated ordering fee $k(i, j)$, that is a function of month i and number of trucks ordered j. Trucks ordered in month i can both be used to fulfill demand in month i, or be stored for future months. You start out with no trucks in storage. The problem is to design an algorithm that decides how many trucks to order each month to satisfy all the demands $\{d_i\}$, and minimize the total cost.

To summarize, every month you pay a storage cost $c(i, j)$, where i is the month and j is the number of trucks stored. Additionally, for each order you place you pay an ordering fee $k(i, j)$, where i is the month and j is the number of trucks ordered.You have to satisfy the demand for trucks d_i each month by either ordering trucks or having trucks in storage. In any month you can store at most s trucks.

(a) Give a recursive algorithm that takes in s, c, k, and the sequence $\{d_i\}$, and outputs the minimum cost. (The algorithm does not need to be efficient.)

We will define the recursive algorithm function $m(i, j)$, which calculates the minimum cost starting from month i when j trucks are carried over from the previous month.

The function $m(i, j)$ is defined as follows:

- If $j \geq d_i$, the cost is the cost of storing the leftover trucks, $m(i+1, j-d_i) + c \cdot (j-d_i)$
- If $j < d_i$, we need to order trucks, incurring the cost of the order fee plus the storage cost, $m(i+1, s) + c \cdot s + k$, as we store up to s trucks and reset the leftover count.

We start with $m(1,0)$, assuming no trucks in the first month. The full solution is found by calculating $m(1,0)$.

(b) Give an algorithm in time that is polynomial in n and s for the same problem.

We will devise a 2D matrix m where each element $m[i][j]$ stores the minimum cost from month i when we carry over j trucks. The algorithm goes backward from month n, where no extra trucks are carried over, ensuring the minimum cost is calculated for each month.

1. Initialize the last month in the matrix so that $m[n][j] = k + c \cdot j$ if j equals the demand for the last month, d_n. For all $j > d_n$, set $m[n][j] = \infty$, indicating it's not possible to carry over extra trucks.
2. Define a function $f(i, j, j')$ that determines whether we need to order more trucks for month i. It returns 0 if the stored trucks j and the trucks we plan to carry over j' meet the demand; otherwise, it returns 1.
3. Use a recursive formula to populate the matrix m. For month i and stored trucks j, calculate $m[i][j]$ as the storage cost $c \cdot j$ plus the minimum of the future cost of storing or ordering trucks. If ordering is needed, add the ordering fee k.
4. Repeat step 3 for months $i = n-1$ to 1, ensuring each month accounts for the optimal future costs.
5. The final minimum cost for the n months is found in $m[1][0]$, the cost for month 1 starting with no trucks.

Therefore, the optimal cost can be deduced with the algorithm.

(c) Prove that your algorithm in part (b) is correct.

We will prove the algorithm by induction.

Base Case:
When $i = n$, the algorithm's output matches the minimum cost because there are no subsequent months to consider, and the cost is based solely on the final month's storage and necessary ordering fees.

Inductive Step:
We will evaluate each possible number of leftover trucks j for the month i and determine the cost using the Bellman equation. If the leftover trucks j exceed the demand d_i, no order is placed, and we only incur the cost of storage. If additional trucks are needed, the ordering fee K is also included in the cost. The lowest cost for future storage or ordering is selected for each scenario.

The running time for filling out the matrix is $O(n, s^2)$, given the matrix has $n \cdot (s+1)$ cells and we perform an operation considering $O(s)$ possibilities for each cell.

4. Alice and Bob are playing another coin game. This time, there are three stacks of n coins: A, B, C. Starting with Alice, each player takes turns taking a coin from the top of a stack – they may choose any nonempty stack, but they must only take the top coin in that stack. The coins have different values. From bottom to top, the coins in stack A have values $a_1, \ldots, a_n$. Similarly, the coins in stack B have values $b_1, \ldots, b_n$, and the coins in stack C have values $c_1, \ldots, c_n$. Both players try to play optimally in order to maximize the total value of their coins.

   (a) Give an algorithm that takes the sequences $a_1, \ldots, a_n$, $b_1, \ldots, b_n$, $c_1, \ldots, c_n$, and outputs the maximum total value of coins that Alice can take. The runtime should be polynomial in n.

We will use dynamic programming approach with two matrices:
AliceOpt[x][y][z] and BobOpt[x][y][z].

Matrix AliceOpt[x][y][z] records the maximum value Alice can collect with x, y, and z coins remaining in stacks A, B, and C.
Matrix BobOpt[x][y][z] does the same for Bob.

Both matrices are initialized with zero for the base case, where no coins are left.
Then we populate these matrices using Bellman Equations, which essentially make decisions based on whether taking a coin leads to a better outcome considering the current state and the opponent's next best move. After fill up the matrices using all coins, the maximum value Alice can secure is found in AliceOpt[n][n][n].

The algorithm runs in polynomial-time $O(n^3)$, considering the number of remaining coins as the variable 'n'.

   (b) Prove the correctness of your algorithm in part (a).

We will prove the algorithm by the induction.

Base case: If there are no coins left in any pile, neighter Alice nor Bob can collete more coins, so the value of the game for both AliceOpt[0][0][0] and BobOpt[0][0][0] is correctly 0.

Inductive step: Assume the algorithm works for all states with fewer than x, y, and z coins. When it's Alice's turn with x, y, z coins remaining, if she takes a coin from pile A with value a_x, then it's Bob's turn with x-1, y, z coins left. The maximum value Alice can achieve in this case is a_x plus the best outcome from BobOpt[x-1][y][z], based on the inductive hypothesis. Similar logic applies when she takes from piles B or C, and the same reasoning applies for Bob's turn.

There, the algorithm is true by the induction.

5. **Coding Question: WIS**

Implement the optimal algorithm for Weighted Interval Scheduling (for a definition of the problem, see the slides on Canvas) in either C, C++, C#, Java, Python, or Rust. Be efficient and implement it in $O(n^2)$ time, where n is the number of jobs. We saw this problem previously in HW3 Q2a, where we saw that there was no optimal greedy heuristic.

The input will start with an positive integer, giving the number of instances that follow. For each instance, there will be a positive integer, giving the number of jobs. For each job, there will be a trio of positive integers i, j and k, where $i < j$, and i is the start time, j is the end time, and k is the weight.

A sample input is the following:

```
2
1
1 4 5
3
1 2 1
3 4 2
2 6 4
```

The sample input has two instances. The first instance has one job to schedule with a start time of 1, an end time of 4, and a weight of 5. The second instance has 3 jobs.

The objective of the problem is to determine a schedule of non-overlapping intervals with maximum weight and to return this maximum weight. For each instance, your program should output the total weight of the intervals scheduled on a separate line. Each output line should be terminated by exactly one newline. The correct output to the sample input would be:

```
5
5
```

or, written with more explicit whitespace,

```
"5\n5\n"
```

Notes:

- Endpoints are exclusive, so it is okay to include a job ending at time t and a job starting at time t in the same schedule.
- In the third set of tests, some outputs will cause overflow on 32-bit signed integers.