Exercises - Dynamic Programming 2 - Algoritmiek

Tutorial February 15, 2024

1. **LCS:** Consider the dynamic programming algorithm for LCS discussed during the lecture.

   (a) Show how to find a solution (so a longest common subsequence) of two strings without using the helper table b.

   (b) Use your dynamic programming algorithm to find a solution for $X = \langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $Y = \langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.
   Compare your answer and the table using the following visualization:
   `https://www.cs.usfca.edu/~galles/visualization/DPLCS.html`

   Solution:

   (a) Start by considering $L[m, n]$. Suppose you consider $L[i, j]$. If $x_i = y_j$, then output x_i and continue your inspection at $L[i-1, j-1]$. Otherwise, if $L[i, j] = L[i-1, j]$, continue your inspection at $L[i-1, j]$; else, continue your inspection at $L[i, j-1]$.
   Be sure to reverse the output to get the actual LCS.

   (b) A possible LCS is 010101.

2. **The Poor Pilgrim:** A pilgrim goes to Rome on foot. The distance is K km. On a day, the pilgrim can walk at most L km. Fortunately, there are $n+1$ lodges on the pilgrim's way. Lodge i lies at distance d_i from the start of the pilgrim's journey; assume $d_0 \leq \cdots \leq d_n$, where d_0 is the start and $d_n = K$ is in Rome. The distance between any two lodges is at most L. Spending the night at lodge i costs the pilgrim p_i gold coins.

   (a) Give an algorithm that computes the smallest number of gold coins the pilgrim needs to complete his journey. Argue that your algorithm is correct and analyze its running time.

   (b) Suppose the pilgrim has B gold coins. Give an algorithm that computes in $O(Bn^2)$ time how the pilgrim can reach Rome by spending as few nights in lodges as possible, while not spending more than B gold coins.

   Solution:

   (a) i. Your top-choice is the last lodge in which the pilgrim spends the night. The subproblem is what the smallest number of gold coins is that the pilgrim needs to go from the start to spending the night in lodge i, while never walking more than L kilometers per day. The general problem is a special case of the subproblem (choose $i = n$).

   ii. $P[i]$ is the smallest number of gold coins is that the pilgrim needs to go from the start to spending the night in lodge i, while never walking more than L kilometers per day.
   $P[i] = p_i$ for all i with $d_i \leq L$.
   $P[i] = p_i + \min_{j:d_i - d_j \leq L} P[j]$.

iii. Suppose $O \subseteq \{1, \ldots, i\}$ is an optimal solution to the subproblem for i; that is, the pilgrim spends the night in the lodges of O and spends $\sum_{j \in O} p_j$ gold coins. Let $i' \in O \setminus \{i\}$ be maximum (the last lodge used by the solution). Let O' be the optimal solution to the subproblem for i'. Since i is within reach of i', $O' \cup \{i\}$ is a feasible solution. By the definition of the subproblem, $\sum_{j \in O'} p_j \leq \sum_{j \in O \setminus \{i\}} p_j$, and thus $O' \cup \{i\}$ costs at most as much gold coin as O. Hence, $O' \cup \{i\}$ is an optimal solution too.

iv. Array P[0...n], initialized to ∞.
for i=0 to n {
if $d_i \leq L$ then $P[i] = p_i$ and continue
for j=i-1 downto 0 {
if $d_i - d_j \leq L$ then $P[i]$ = Math.min{ $P[i]$, $p_i + P[j]$ }
} }
return $P[n]$

v. The running time is $O(n^2)$. Note that by changing the implementation slightly (not looking back too far), the running time can be reduced to $O(nL)$.

(b) i. The top-choice is the last lodge in which the pilgrim spends the night. The subproblem is what the smallest number of nights needed is for the pilgrim to go from start to spending the night in lodge i using at most D gold coins, while never walking more than L kilometers per day. The general problem is a special case of the subproblem (choose $i = n$ and $D = B$).

ii. $P[i, D]$ is the smallest number of nights needed is for the pilgrim to go from start to spending the night in lodge i using at most D gold coins, while never walking more than L kilometers per day.
$P[0, D] = 0$ for all $0 \leq D \leq B$.
$P[i, D] = 1 + \min_{j : d_i < d_j \leq L} P[j, D - p_i]$

iii. Suppose $O \subseteq \{1, \ldots, i\}$ is an optimal solution to the subproblem for i and D; that is, the pilgrim spends the night in the lodges of O and spends $\sum_{j \in O} p_j \leq D$ gold coins. Let $i' \in O$ be maximum (the last lodge used by the solution). Let O' be the optimal solution to the subproblem for i' and $D - p_i$. Since O can get to i' by spending at most $D - p_i$, O' exists. Since i is within reach of i', $O' \cup \{i\}$ is a feasible solution. By the definition of the subproblem, $|O'| \leq |O \setminus \{i\}|$, and thus $|O' \cup \{i\}| \leq |O|$. Hence, $O' \cup \{i\}$ is an optimal solution too.

iv. Array P[0...n, 0...B], initialized to ∞
for D=0 to B {
P[0,D] = 0
}
for i=1 to n {
for D=0 to B {
for j=0 to n {
if $d_i - d_j \leq L$ then $P[i, D]$ = Math.min{ P[i,D], $1 + P[i, D - p_i]$ }
} } }
return P[n,B]

v. The running time is $O(n^2 B)$. Again, this can be sharpened to $O(nLB)$.

3. **Snapshots:** A city has n famous attractions, $A_1, \ldots, A_n$. You are at your AirHotel, A_0. As a photographer, you want to snap snapshots of the attractions. A snapshot of A_i has value g_i. However, you also have limited time T before darkness sets in and you need to be back at your AirHotel. It takes $d[i, j]$ time to travel from A_i to A_j. Use dynamic programming to decide on a tour that yields snapshots of highest value. Analyze the running time of your algorithm

Solution:

(a) The top-choice is the final attraction snapshotted. The subproblem is the highest value that can be obtained by touring attractions $S \subseteq \{1, \ldots, n\}$ and then going to and photographing A_k with $k \notin S$, while not spending more than T' time. The general problem is a special case of the subproblem by taking $k = 0$ and considering the best value over all $S \subseteq \{1, \ldots, n\}$.

(b) $R[S, k, T']$ is the maximum value that can be obtained by touring attractions $S \subseteq \{1, \ldots, n\}$ and then going to and photographing A_k with $k \notin S$, while not spending more than T' time.
$R[\emptyset, k, T'] = g_k$ if $d[0, k] \leq T'$
$R[\emptyset, k, T'] = 0$ otherwise
$R[S, k, T'] = g_k + \max_{k' \in S, d[k', k] \leq T'} \{R[S \setminus \{k'\}, k', T' - d[k', k]]\}$

(c) Let R be a tour for $S \subseteq \{1, \ldots, n\}$ with the last snapshot taken at $k \notin S$ and within time T'. Suppose $k' \in S$ is the snapshot taken in R before k. Let R' be an optimal tour for the subproblem for $S \setminus \{k'\}$ and $T' - d[k', k]$. Note that R' followed by going from k' to k (call this U) reaches all attractions in S and k within time T'. Then $g(R') \leq g(R[S \setminus \{i\}])$ and thus $g(U) \leq g(R)$. Hence, U is also an optimal solution.

(d) Array $R[0 \ldots 2^n, \{1, \ldots, n\}, 0 \ldots T]$, initialized to 0
for k=0 to n {
for T'=0 to T {
if $d[0, k] \leq T'$ then $R[\emptyset, k, T'] = g_k$ } }
for all $S \subseteq \{1, \ldots, n\}$, $S \neq \emptyset$ { for k=0 to n {
for T'=0 to T {
for all $k' \in S$ {
if $d[k', k] \leq T'$ then R[S,k,T'] = Math.max{ R[S,k,T'], $g_k + R[S \setminus \{k'\}, k', T' - d[k', k]]$ }
} } } }
ret = $\emptyset$
retmax = 0
for all $S \subseteq \{1, \ldots, n\}$, $S \neq \emptyset$ {
if retmax ¡ R[S,0,T] then ret = S; retmax = R[S,0,T]
}
return retmax

(e) The running time is $O(2^n n^2 T)$.

4. **Dicy race:** Bert and Ernie are playing a simple board game. They start at place 1 on a board with n places. Bert starts. Taking turns alternatingly, they throw a regular six-sided dice. They move forward as many places as the number they throw with the dice. Who ends up at the last place, or passes it, wins the game.
Because Bert starts, he has a probability more than $\frac{1}{2}$ to win the game. Given a value of n, we want to compute this probability. We will do this using dynamic programming.
Suppose $B(i, j)$ is the probability that Bert wins if Bert still has i steps until the end and Ernie still has j steps until the end, and it's Bert's turn.
Suppose $E(i, j)$ is the probability that Ernie wins if Bert still has i steps until the end and Ernie still has j steps until the end, and it's Ernie's turn.

   (a) Why do we want to know $B(n-1, n-1)$?

   (b) Give recurrences for $B(i, j)$ and $E(i, j)$ if $i \geq 6$.

   (c) Give the base cases for the recurrence: $B(1, j)$, $B(2, j)$, $B(3, j)$, $B(4, j)$, $B(5, j)$.

   (d) What is the order of computation for the recurrence?

   (e) Give pseudocode for an algorithm that computes $B(n-1, n-1)$ in $O(n^2)$ time.

Solution:

   (a) This is the probability that Bert wins the game. Both Bert and Ernie need $n-1$ steps until the end.

   (b) $B(i,j) = \frac{1}{6}E(i-1,j)+\frac{1}{6}E(i-2,j)+\frac{1}{6}E(i-3,j)+\frac{1}{6}E(i-4,j)+\frac{1}{6}E(i-5,j)+\frac{1}{6}E(i-6,j)$.
   $E(i,j) = \frac{1}{6}B(i,j-1)+\frac{1}{6}B(i,j-2)+\frac{1}{6}B(i,j-3)+\frac{1}{6}B(i,j-4)+\frac{1}{6}B(i,j-5)+\frac{1}{6}B(i,j-6)$

   (c) $B(1, j) = 1$, because Bert always gets to the last place from the second to last place in a single turn.
   $B(2, j) = (5/6) + (1/6) \cdot (1 - E(1, j))$
   $B(3, j) = (4/6) + (1/6) \cdot (1 - E(2, j)) + (1/6) \cdot (1 - E(1, j))$
   etc.

   (d) Start with the base cases $B(1, j)$, $B(2, j)$, $B(3, j)$, $B(4, j)$, $B(5, j)$, $E(1, j)$, $E(2, j)$, $E(3, j)$, $E(4, j)$, $E(5, j)$. To compute $B(i, j)$, we need $E(i', j)$ for smaller values of $i' < i$. To compute $E(i, j)$, we need $B(i, j')$ for $j' < j$. So we iterate over all i and then over all j.

   (e) Array B[1... n-1,1... n-1], E[1... n,1... n], initialized to 0
   Set the base cases.
   for i=1 to n-1 {
   for j=1 to n-1 {
   $B[i,j] = \frac{1}{6}E[i-1,j]+\frac{1}{6}E[i-2,j]+\frac{1}{6}E[i-3,j]+\frac{1}{6}E[i-4,j]+\frac{1}{6}E[i-5,j]+\frac{1}{6}E[i-6,j]$
   $E[i,j] = \frac{1}{6}B[i,j-1]+\frac{1}{6}B[i,j-2]+\frac{1}{6}B[i,j-3]+\frac{1}{6}B[i,j-4]+\frac{1}{6}B[i,j-5]+\frac{1}{6}B[i,j-6]$
   } }
   return B[n-1,n-1]

5. **Party!:** You are the organizer of a party at the company for which you intern and need to send out invitations. The company has a strict hierarchy: every employee has a single supervisor, who also has a supervisor, etc., right until the CEO. Every supervisor can have multiple direct subordinates (who again have multiple direct subordinates etc.). Partying with your supervisor or your direct subordinates is always a bit weird; therefore, if you invite an employee to the party, you are not allowed to invite their supervisor nor their direct subordinates. Still, you want to invite as many employees to the party as possible.

   (a) Give a dynamic programming algorithm for this problem. Use the steps discussed in class!

   (b) Analyze the running time of your algorithm.

   (c) The company has given each employee a party-ranking. The higher the total rank of the invited employees, the more fun the party will be. Adapt your algorithm to arrange the best party evah!

Solution:

(a) We can view the hierarchy as a rooted tree T, with the CEO as the root. For any node v of the tree, let T_v denote the subtree rooted at v (so v and all its descendants).

i. The top-choice is whether to invite the highest supervisor (so the root v of the tree). Inviting the root means that none of the direct subordinates (children of v) can come to the party. Not inviting the root means that the direct subordinates can be invited, but don't have to be. The subproblem is, given a vertex v and whether or not we are allowed to invite v, what is the maximum number of people to invite in T_v. Indeed, the general problem is a special case of this subproblem, taking v to be the CEO and considering both options whether to invite the CEO or not.

ii. For each vertex v and boolean a, $I[v, a]$ is the maximum number of people we can invite in T_v where a indicates whether or not we are allowed to invite v.
For a leaf (employee without subordinates), $I[v, \text{true}] = 1$ and $I[v, \text{false}] = 0$.
Otherwise, $I[v, \text{true}] = \max\{\sum_w I[w, \text{true}], 1+\sum_w I[w, \text{false}]\}$ and $I[v, \text{false}] = \sum_w I[w, \text{true}]$, where in all cases the sum is over all direct subordinates w of v (all children w of v).

iii. Consider a subproblem for some root v that we are not allowed to invite and let S be the optimal set of people invite in this case. Since we cannot invite v, we are allowed to invite the children of v. For each child w of v (so each direct subordinate of v), let U_w be the optimal solution for the subproblem with parameters w and allowing to invite w. Note that $S \cap T_w$ is such a solution. Replace $S \cap T_w$ by U_w and call the resulting solution U. Since $|U_w| \geq |S \cap T_w|$, it follows that $|U| \geq |S|$ (note that neither invite v). Hence, U is also an optimal solution.
Consider a subproblem for some root v that we are allowed to invite and let S be the optimal set of people invite in this case. Now the top-choice is whether or not to invite v. The case when v is not invited ($v \notin S$) is the same as before. So suppose we invite v ($v \in S$). Then we are not allowed to invite the children of v. For each child w of v (so each direct subordinate of v), let X_w be the optimal solution for the subproblem with parameters w and not allowing to invite w. Note that $S \cap T_w$ is such a solution. Replace $S \cap T_w$ by X_w and call the resulting solution, plus v, X. Since $|X_w| \geq |S \cap T_w|$, it follows that $|X| \geq |S|$ (note that both invite v). Hence, X is also an optimal solution.

iv. For the filling order, we use a *pre-order traversal* of the tree. That is, we visit the children of a vertex, before visiting the children.

v. Array I[all nodes, {true,false}], initialized to 0
for all leafs (employees with no subordinates) {
$I[v, \text{true}] = 1$
$I[v, \text{false}] = 0$
}
for all non-leaf employees v, using a pre-order traversal {
T = 0
F = 0
for all children w of v {
T += $I[w, \text{true}]$
F += $I[w, \text{false}]$
}
$I[v, \text{true}] = \max\{T, 1 + F\}$
$I[v, \text{false}] = T$
}
return $\max\{I[\text{CEO}, \text{true}], I[\text{CEO}, \text{true}]\}$

(b) The running time is $O(n)$, where n is the number of employees. We spend $O(1)$ time per employee.

(c) Denote the party ranking of employee v by p_v.
Array I[all nodes, {true,false}], initialized to 0
for all leafs (employees with no subordinates) {
$I[v, \text{true}] = p_v$
$I[v, \text{false}] = 0$
}
for all non-leaf employees v, using a pre-order traversal {
T = 0
F = 0
for all children w of v {
T += $I[w, \text{true}]$
F += $I[w, \text{false}]$
}
$I[v, \text{true}] = \max\{T, p_v + F\}$
$I[v, \text{false}] = T$
}
return $\max\{I[\text{CEO}, \text{true}], I[\text{CEO}, \text{true}]\}$