Problem Set 3

> *Which of the following is true for our dynamic programming algorithm for computing a maximum-weight independent set of a path graph? (Assume there are no ties.)*
>
> - *As long as the input graph has at least two vertices, the algorithm never selects the minimum-weight vertex.*
> - *The algorithm always selects the maximum-weight vertex.*
> - *If a vertex is excluded from the optimal solution of two consecutive subproblems, then it is excluded from the optimal solutions of all bigger subproblems.*
> - *If a vertex is excluded from the optimal solution of a subproblem, then it is excluded from the optimal solutions of all bigger subproblems.*

ANSWER: Consider the path $10 - 2 - 1 - 4$. The vertices selected by the algorithm are $10, 1$, where 1, the minimum, is selected. Thus, option 1 is incorrect.

Consider the path $1 - 3 - 10 - 9$. The vertices selected by the algorithm are $3, 9$, where the maximum 10 isn't selected. Thus, option 2 is incorrect.

Consider the path $1 - 9 - 7 - 1 - 5$. The vertices selected by the algorithm are $1, 7, 5$. However, 7 was not included in the optional solution of the subproblem $1 - 9 - 7$. Note that, 7 was not included in the optional solution of the subproblem $1 - 9 - 7 - 1$ either, because the its previous vertex was "heavier", and since all weights are positive, the sum of the next weight and the heavier vertex is certainly greater than 7. Thus, option 4 is incorrect.

Option 3 is correct. This follows from induction, since the optimal solution to a subproblem depends only on the solutions of the previous two subproblems.

> *Recall our dynamic programming algorithm for computing the maximum-weight independent set of a path graph. Consider the following proposed extension to more general graphs. Consider an undirected graph with positive vertex weights. For a vertex v, obtain the graph $G'(v)$ by deleting v and its incident edges from G, and obtain the graph $G''(v)$ from G by deleting v, its neighbors, and all of the corresponding incident edges from G. Let $OPT(H)$ denote the value of a maximum-weight independent set of a graph H. Consider the formula $OPT(G) = \max\{OPT(G'(v)), w_v + OPT(G''(v))\}$, where v is an arbitrary vertex of G of weight w_v. Which of the following statements is true?*
>
> - *The formula is always correct in trees, and it leads to an efficient dynamic programming algorithm.*
> - *The formula is always correct in trees, but does not lead to an efficient dynamic programming algorithm.*
> - *The formula is correct in path graphs but is not always correct in trees.*
> - *The formula is always correct in general graphs, and it leads to an efficient dynamic programming algorithm.*

ANSWER: Clearly, both G' and G'' are trees if G is a tree. If there was no cycle in G, there're none in G' and G'' either. Also, removing an edge and all its incident edges preserves connectivity among the remaining vertices. Thus, G' and G'' exhibit optimal substructures, and hence the algorithm is correct for trees. However, we can't prove optimal substructure property for arbitrary general graphs. Thus, option 1 is correct, and the rest are not. The time complexity is $O(|V|)$; see MWIS in a Tree - Chen + Kuo + Sheu].

> *Consider a variation of the Knapsack problem where we have two knapsacks, with integer capacities W_1 and W_2. As usual, we are given n items with positive values and positive integer weights. We want to pick subsets S_1, S_2 with maximum total value (i.e., $\sum_{i \in S_1} v_i + \sum_{i \in S_2} v_i$) such that the total weights of S_1 and S_1 are at most W_1 and W_2, respectively. Assume that every item fits in either knapsack (i.e., $w_i \leq \min\{W_1, W_2\}$ for every item i). Consider the following two algorithmic approaches. (1) Use the algorithm from lecture to pick a max-value feasible solution S_1 for the first knapsack, and then run it again on the remaining items to pick a max-value feasible solution S_2 for the second knapsack. (2) Use the algorithm from lecture to pick a max-value feasible solution for a knapsack with capacity $W_1 + W_2$, and then split the chosen items into two sets $S_1 + S_2$ that have size at most W_1 and W_2, respectively.*
>
> *Which of the following statements is true?*
>
> - *Algorithm (1) is guaranteed to produce an optimal feasible solution to the original problem provided $W_1 = W_2$.*
> - *Algorithm (1) is guaranteed to produce an optimal feasible solution to the original problem but algorithm (2) is not.*
> - *Algorithm (2) is guaranteed to produce an optimal feasible solution to the original problem but algorithm (1) is not.*
> - *Neither algorithm is guaranteed to produce an optimal feasible solution to the original problem.*

ANSWER: Consider the set of items $S(w_i, v_i) = (1,2), (2,1), (3,10), (4,7)$ and two sacks of equal capacities 5. Algorithm 1 picks s_1 and s_3 in the first iteration, and s_4 in the second (s_2 is not picked). Total value of both sacks $12 + 7 = 19$. However, the optimal solution would put s_1 and s_4 in one sack, and s_2 and s_3 in the other, bringing the total value of both sacks to $9 + 11 = 20$. Thus, option 1 is incorrect.

Consider the set of items $S(w_i, v_i) = (3,10), (3,10), (4,2)$ and two sacks of capacities 7 and 3, respectively. Algorithm 1 picks s_1 and s_2 in the first iteration, and nothing in the second iteration. Total value of both sacks 20. However, the optimal solution would put s_1 and s_3 in one sack, and s_2 in the other, bringing the total value of both sacks to $12 + 10 = 22$. Thus, option 2 is incorrect.

Consider the set of items $S(w_i, v_i) = (4,10), (4,10), (4,10), (2,1)$ and two sacks of equal capacities 6. Algorithm 2 picks s_1, s_2, s_3, but this set cannot fit into two sacks, so one is dropped when redistributing. Total value of both sacks $10 + 10 = 20$. However, the optimal solution would put s_1 and s_4 in one sack, and s_2 in the other (s_3 is not picked), bringing the total value of both sacks to $11 + 10 = 21$. Thus, option 3 is incorrect.

Option 4 is correct based on the above counterexamples.

A correct algorithm would consider both sacks, and the following three cases:

1. Item is put into sack 1.
2. Item is put into sack 2.
3. Item is not put into any sack.

The recurrence, therefore, is:

$$\begin{aligned} a[i][j][k] = \max\{ & \\ & a[i-1][j][k], \\ & a[i-1][j-w_i][k] + c_i, \\ & a[i-1][j][k-w_i] + c_i \\ & \} \end{aligned}$$

where i is the index corresponding to the items, j corresponding to sack 1, and k corresponding to sack 2.

> *Recall the dynamic programming algorithms from lecture for the Knapsack and sequence alignment problems. Both fill in a two-dimensional table using a double-for loop. Suppose we reverse the order of the two for loops. (I.e., cut and paste the second for loop in front of the first for loop, without otherwise changing the text in any way.)*
>
> *Are the resulting algorithms still well defined and correct?*
>
> - *Neither algorithm remains well defined and correct after reversing the order of the for loops.*
> - *The Knapsack algorithm remains well defined and correct after reversing the order of the for loops, but the sequence alignment algorithm does not.*
> - *The sequence alignment algorithm remains well defined and correct after reversing the order of the for loops, but the Knapsack algorithm does not.*
> - *Both algorithms remain well defined and correct after reversing the order of the for loops.*

ANSWER: The recurrence for the knapsack problem is $a[i][x] = \max\{a[i-1][x], a[i-1][x-w_i]\}$. What we care about is that the array entries on the R.H.S. have already been populated by the time we attempt to populate $a[i][x]$. Clearly, the order of the for loops doesn't matter; the lecture has x in the inner loop, but if we have it in the outer, the $(i-1)$-th entries for some x are still populated before i-th entry. Thus, option 4 is correct.

> *Consider an instance of the optimal binary search tree problem with 7 keys (say 1,2,3,4,5,6,7 in sorted order) and frequencies* $w_1 = .05, w_2 = .4, w_3 = .08, w_4 = .04, w_5 = .1, w_6 = .1, w_7 = .23$.
>
> *What is the minimum-possible average search time of a binary search tree with these keys?*
>
> - *2.08*
> - *2.42*
> - *2.9*
> - *2.18*

ANSWER: Option 4 is correct. See the code on GitHub.

> *The following problems all take as input two strings X and Y, of length m and n, over some alphabet Σ. Which of them can be solved in $O(mn)$ time? [Check all that apply.]*
>
> - *Compute the length of a longest common substring of X and Y. (A substring is a consecutive subsequence of a string. So "bcd" is a substring of "abcdef", whereas "bdf" is not.)*
> - *Compute the length of a longest common subsequence of X and Y. (Recall a subsequence need not be consecutive. For example, the longest common subsequence of "abcdef" and "afebcd" is "abcd".)*
> - *Assume that X and Y have the same length n. Does there exist a permutation f, mapping each $i \in \{1, 2, \ldots, n\}$ to a distinct $f(i) \in \{1, 2, \ldots, n\}$, such that $X_i = Y_{f(i)}$ for every $i = 1, 2, \ldots, n$?*
> - *Consider the following variation of sequence alignment. Instead of a single gap penalty α_{gap}, you're given two numbers a and b. The penalty of inserting k gaps in a row is now defined as $ak + b$, rather than $k\alpha_{gap}$. Other penalties (for matching two non-gaps) are defined as before. The goal is to compute the minimum-possible penalty of an alignment under this new cost model.*

ANSWER: Let X_i denote the prefix of X including the character at the i-th position, and Y_j denote the prefix of Y including the character at the j-th position. The characters at the last positions of X_i and Y_j either match or don't; a match increases the length of the longest common substring found so far by one. Thus, if L_i denotes the length of the longest common substring of X_i and Y_j, the recurrence can be written as:

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 \text{ if } X[i] = Y[j] \\ 0 \text{ otherwise} \end{cases}$$

To find the longest common substring, we can either keep track of the maximum positive value while populating the array, or find it later by scanning each element (yikes!). Once found, follow the usual reconstruction procedure as explained in the lectures.

Clearly, this can be solved in $O(mn)$ time. Thus, option 1 is correct.

The longest subsequence problem is similar to the longest substring, except for a no match, we pick up the match value so far. The recurrence can be written as:

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 \text{ if } X[i] = Y[j] \\ \max\{L_{i-1,j}, L_{i,j-1}\} \text{ otherwise} \end{cases}$$

Clearly, this can be solved in $O(mn)$ time. Thus, option 2 is correct. Alternatively, this reduces to sequence alignment by setting the gap penalty to 1 and making the penalty of matching two different characters to be very large.

Problem in option 3 can be solved in $O(n)$ time, without dynamic programming. We simply count the frequency of each symbol in each string. The permutation f exists if and only if every symbol occurs exactly the same number of times in each string.

Problem in option 4 is a variation of the original sequence alignment dynamic program. With each subproblem, we need to keep track of what gaps we insert, since the costs incurred at the current position depend on whether or not the previous subproblems inserted gaps. Blows up the number of subproblems and running time by a constant factor, but still can be solved in $O(n)$ time. Thus, option 4 is correct.

> *Recall our dynamic programming algorithms for maximum-weight independent set, sequence alignment, and optimal binary search trees. The space requirements of these algorithms are proportional to the number of subproblems that get solved: $\Theta(n)$ (where n is the number of vertices), $\Theta(mn)$ (where m, n are the lengths of the two strings), and $\Theta(n^2)$ (where n is the number of keys), respectively.*

Suppose we only want to compute the value of an optimal solution (the final answer of the first, forward pass) and don't care about actually reconstructing an optimal solution (i.e., we skip the second, reverse pass over the table). How much space do you then really need to run each of three algorithms?

- $\Theta(1), \Theta(n) \text{ and } \Theta(n^2)$
- $\Theta(n), \Theta(mn) \text{ and } \Theta(n^2)$
- $\Theta(1), \Theta(n) \text{ and } \Theta(n)$
- $\Theta(1), \Theta(1) \text{ and } \Theta(n)$

ANSWER: The MWIS problem only ever looks at the last two columns ($i-1$ and $i-2$), thus, we can get away with storing just those two values.

The sequence alignment problem may need to look at the previous row ($P_{i-1,j}$ or $P_{i-1,j-1}$), or the previous column in the current row ($P_{i,j-1}$). Thus, we can only store two rows, which takes linear space.

Unfortunately, the optional BST problems needs to keep track of n^2 subproblems, and can't be solved in reduced space.

Thus, option 1 is correct.

COMMENTS