Informatics 2 – Introduction to Algorithms and Data Structures

Tutorial 6 - Greedy and Dynamic Programming Solutions

January 30, 2024

1. In this question, you are asked to execute Dijkstra's Shortest Path Algorithm starting from node 0 in Figure 1. Show the steps of the algorithm and the value of $d(v)$ computed for each node v added to the set S of explored nodes. Also show the shortest path P_u which is computed for each such new node u throughout the execution of the algorithm.

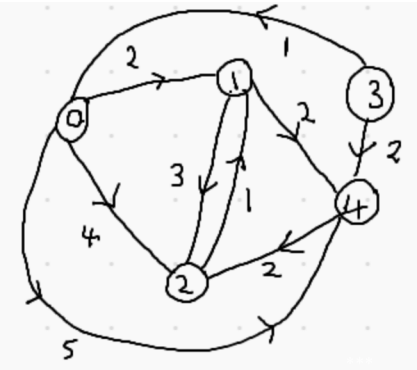

Figure 1: A directed graph with edge lengths indicated over the edges.

Solution: Let $G = (V, E)$ be the graph of the figure. Initially we have $d[u] = \infty$ and $P_u = \text{NULL}$ for each $u \in V\{0\}$, and $d[0] = 0$, $P_0 = (0)$.

First Iteration: *Candidate nodes to be added to S:* 1, 2, 4.

$$d'(1) = d(0) + \ell_{(0,1)} = 0 + 2 = 2$$
$$d'(2) = d(0) + \ell_{(0,2)} = 0 + 4 = 4$$
$$d'(4) = d(0) + \ell_{(0,4)} = 0 + 5 = 5$$

The minimum distance is $d'(1)$, so we have $S = S \cup \{1\} = \{0, 1\}$. We record $d(1) = d'(1) = 2$ and $P_1 = (0 \to 1)$.

Second Iteration: *Candidate nodes to be added to S:* 2, 4.

Remark: Now nodes 2 and 4 can also be reached via the edges (1,2) and (1,4) in addition to (0,2) and (0,4) respectively. That means that we might need to calculate

$d'(2)$ and $d'(4)$ again.

$$d'(2) = \min\{d(0) + \ell_{(0,2)}, d(1) + \ell_{(1,2)}\} = \min\{4, 5\} = 4$$
$$d'(4) = \min\{d(0) + \ell_{(0,4)}, d(1) + \ell_{(1,4)}\} = \min\{5, 4\} = 4$$

The minimum distance if $d'(2) = d'(4) = 4$ so both options are possible. Assume that we choose node 2 to be added, so we have $S = S \cup \{2\} = \{0, 1, 2\}$. We record $d(2) = d'(2) = 4$ and $P_2 = (0 \to 2)$.

Third Iteration: *Candidate nodes to be added to S:* 4.

Remark: Node 4 can be reached via the edges (0,4) and (1,4) as before, so we don't need to recalculate.

We have $d'(4) = 4$ and we add node 4 to S, so we have $S = S \cup \{4\} = \{0, 1, 2, 4\}$. We record $d(4) = d'(4) = 4$ and $P_4 = (0 \to 1 \to 4)$.

Third Iteration: *Candidate nodes to be added to S:* None.

Remark: Node 3 has not been visited, but it cannot be reached from 0. This is different from what we saw in the lectures, where we assumed that every node is reachable from the starting node. If this is not the case, the algorithm only needs to explore nodes in the connected component of the starting node, and then terminate.

2. Consider the *fractional knapsack problem*, in which there is a set of n *infinitely divisible* items with values v_i, for $i = 1, \ldots, n$ and weights w_i, for $i = 1, \ldots, n$, and there is a total weight constraint W. The goal is to find fractions $(x_1, \ldots, x_n)$ of each item, with $0 \leq x_i \leq 1$ such that $\sum_{i=1}^{n} x_i \cdot v_i$ is maximised, subject to the total weight constraint $\sum_{i=1}^{n} x_i \cdot w_i \leq W$.

   A greedy algorithm for the fractional knapsack problem in each step chooses an item i based on some criterion and adds the largest fraction x_i of item i that is possible to fit in the knapsack, without violating the weight constraint, given the items already added to the knapsack in previous steps. The algorithm terminates when the total weight of items included in the knapsack exactly meets the weight constraint, or when all of the items have entirely been added (i.e., $x_i = 1$ for all i) in the knapsack.

   Consider the following three criteria for selecting the item to be added to the knapsack:

   (a) Add the item with the largest value v_i among those not already considered (Criterion a).

   (b) Add the item with the smallest weight w_i among those not already considered (Criterion b).

   (c) Add the item with the largest ratio v_i/w_i among those not already considered (Criterion c).

   The first objective is to prove that:

   (a) the greedy algorithms with Criterion a or Criterion b are not optimal.

   (b) the greedy algorithm with Criterion c is optimal.

   Consider the greedy algorithm with Criterion c for the (0/1)-Knapsack problem that we saw in the lectures, which either adds a job entirely to the knapsack or not at all. Does it solve the 0/1-Knapsack problem optimally? Provide either a correctness proof or a counterexample to support your claim.

   Solution:

(a) For Criterion a, consider an instance with three items with values $v_1 = 10$, $v_2 = 9$, $v_3 = 9$, weights $w_1 = 10$, $w_2 = 5$ and $w_3 = 5$ and $W = 10$. The algorithm will add $x_1 = 1, x_2 = x_3 = 0$ for a total value of 10. The optimal solution will add $x_1 = 0, x_2 = x_3 = 1$ for a total value of 18. For Criterion b, consider an instance with three items with values $v_1 = 1$, $v_2 = 1$ and $v_3 = 10$, weights $w_1 = w_2 = 5$ and $w_3 = 10$, and $W = 10$. The algorithm will add $x_1 = x_2 = 1, x_3 = 0$ for a total value of 2, whereas the optimal solution will add $x_1 = x_2 = 0, x_3 = 1$ for a total value of 10.

(b) This is Dantzig's greedy algorithm for solving the fractional knapsack problem. The algorithm works as follows:

- First, sort the items in terms on non-increasing ratio v_i/w_i (this is sometimes called the "bang-per-buck").
- Start putting items in the knapsack in that order, until you encounter an item that can not fit in the knapsack anymore.
- Put as large a fraction of that item in the knapsack, until you reach the total weight constraint W.

We claim that this algorithm solves the fractional knapsack problem optimally. Suppose without loss of generality that the items are sorted in terms of non-increasing v_i/w_i and that no two items have the same such ratio (therefore the order is actually decreasing). Let $o_1, o_2, \ldots, o_n$ be the fractions of items that are put in the knapsack in the optimal solution in that order, and let $g_1, \ldots, g_n$ be the fractions of items selected by the greedy algorithm (in both cases, some fractions might be 0). By definition of the optimal solution, we have that $\sum_{i=1}^{n} o_i \cdot v_i \geq \sum_{i=1}^{n} g_i \cdot v_i$.
If $o_i = g_i$ for every index, then the two solutions are the same and we are done. Let j be the first index for which the two solutions differ. By definition of the greedy algorithm, it must hold that $g_j > o_j$, as item j has the largest ratio v_j/w_j over all remaining items (not already in the knapsack), and the greedy algorithm selects as large a fraction of it as possible. By the definition of the optimal, there must exist another index $\ell > j$ such that $o_\ell > g_\ell$. Now, construct a new solution $o' = \{o'_1, o'_2, \ldots, o'_n\}$ such that

- $o'_k = o_k$ for all $k \neq j, \ell$,
- $o'_j = o_j + \varepsilon$,
- $o'_\ell = o_\ell - \varepsilon \cdot \frac{w_j}{w_\ell}$.

Note that $\sum_{i=1}^{n} o_i \cdot w_i = \sum_{i=1}^{n} o'_i \cdot w_i$ and therefore o'_i is a feasible solution. Furthermore, we have that

$$\sum_{i=1}^{n} o'_i \cdot v_i = \sum_{i=1}^{n} o_i \cdot v_i + \varepsilon v_j - \varepsilon \cdot \frac{w_j}{w_\ell} \cdot v_\ell > \sum_{i=1}^{n} o_i \cdot v_i,$$

where the last inequality holds since $v_j/w_j > v_\ell/w_\ell$. This means that o is a feasible solution with largest total value than o, contradicting the optimality of o.

Dantzig's algorithm does not solve the 0/1-Knapsack problem optimally. To see this, consider an instance with 3 jobs such that $v_1 = 300$, $v_2 = 190$ and $v_3 = 180$, $w_1 = 100$, $w_2 = 95$, and $w_3 = 90$, and $W = 185$. We have $v_1/w_1 = 3$, $v_2/w_2 = 2$, $v_3/w_3 = 2$.

Therefore the algorithm will add the entire item 1 (i.e., $x_1 = 1$) to the knapsack only, for a total value of 300. The optimal solution will add items 2 and 3 instead ($x_2 = x_3 = 1$) for a total value of $190 + 180 = 370$.

3. In the UK, coins have demoninations 1p, 2p, 5p, 10p, 20p, 50p, £1 and £2. A frequently-executed task in the retail sector involves taking an input value (say 88p) and calculating a collection of coins (which may include duplicates) which will sum to that value. We assume that we have an unlimited supply of coins of each value. Formally, the *coin changing problem* is the following:

   Input: An input value $v \in \mathbb{N}_0$, and a sequence of coin values $c_0, c_1, \ldots c_k \in \mathbb{N}_0$.

   Output: A *multiset* S of coins with values that sum to v, whose size is the minimum possible for v in this system. The solution will be represented as a list S of length k, with $S[i]$ being the number of coins of value c_i, for each $i \in [0, k]$.

   We may assume that a solution is always possible (e.g., by assuming that $c_0 = 1$).

   Design a dynamic programming-based algorithm which solves the coin changing problem. Run your algorithm on the following input with $k = 2$: $v = 18$, $c_0 = 1$, $c_1 = 5$, and $c_2 = 7$.

   Instead of the dynamic programming algorithm, perhaps a simple greedy algorithm could work: as the greedy criterion, always choose the coin of maximum value among those whose value is smaller than the remaining value. What is the outcome of the algorithm on the example above?

   Solution: Let $C(v)$ denote the value of the optimal solution (i.e., the number of coins) on input $v \in \mathbb{N}_0$. In the optimal solution, there must exist *some* coin of denomination c_i. Furthermore, the remaining value $v - c_i$ that we need to make with our remaining coins is an optimal solution to the coin changing problem on input $v - c_i$; our problem has therefore the optimal substructure property. Specifically, we have that

   $$C(v) = 1 + C(v - c_i).$$

   One potential issue here is that we do not know the denomination c_i of the first coin i to be included in $C(v)$. We can however check all k possibilities and find the minimum value. In other words, we can write

   $$C(v) = \begin{cases} 1, \text{ if there exists } i \in [0,k] \text{ such that } c_i = v \\ 1 + \min_{j \in [0,k]} C(v - c_j), \text{ otherwise.} \end{cases}$$

   We have therefore a recurrence relation that allows us to compute the value $C(v)$ via computing the value $C(v - c_j)$ for other denominations c_j. In dynamic programming fashion, we will precompute those and store them in an array for use when needed. In particular, we will compute $C(w)$ for all w from 1 to v (recall that 1 was our lowest demonination), where $C[w]$ will store the value of the optimal solution (i.e., the smallest number of coins) to make a value of w. We can also use an auxiliary array P to store the coins that led us to the values of the optimal solution (or we can compute those afterwards based on the values). We present the pseudocode of the algorithm below.

Algorithm CoinChanging$(v, c_0, \ldots, c_k)$

$C[v]$ will be the optimal number of coins, and S will contain the coins themselves.

```
Initialise C, P to contain ∞, S to contain 0.
C[0] ← 0, C[1] ← 1, P[1] ← 0.
for w = 2 to v do
    for i = 0 to k do
        if c_i ≤ w and C[w − c_i] + 1 < C[w] then
            C[w] ← 1 + C[w − c_i]
            P[w] ← i
        end if
    end for
end for
while v > 0 do
    i ← P[v]
    S[i] ← S[i] + 1
    v ← v − c_i
end while
```

Suppose now that we run the algorithm on the input with $k = 2$: $v = 18$, $c_0 = 1$, $c_1 = 5$, and $c_2 = 7$. We have $C[0] = 0$, $C[1] = 1$, and $P[0] = 0$ in the initialisation step. Our outer for loop will run for w from 2 to 18.

For $w = 2$, we first consider $i = 0$ in the inner loop, and hence the denomination $c_0 = 1$. We have $1 = c_i \leq 2 = w$ and $C[2] = \infty$ at this point, so the body of the inner loop will also run, and will compute $C[2] = 1 + C[1] = 2$, and $P[1] = 1$. Then our inner loop will run for $i = 1$. Since $5 = c_2 > 2 = w$, the body of the inner loop will not run. For the same reason, it will not run for $i = 2$.

For $w = 3$, it is not hard to see that our inner loop body will only run for $i = 0$. In that case, we have $1 = c_0 \leq 3$ and $C[3] = \infty$, so we will compute $C[3] = C[2] + 1 = 3$. We will also have $P[3] = 0$. For $w = 4$ the situation is similar and we compute $C[4] = 4$, $P[4] = 0$.

For $w = 5$, we will have two iterations of the inner for loop. For $i = 0$, the loop will calculate $C[5] = 5$ and $P[5] = 0$. For $i = 1$, we have $5 = c_1 \leq 5 = w$ and $C[0] + 1 < C[5] = 5$, since $C[0] = 0$. This means that the loop will update $C[5] = 0$ and $P[5] = 1$. This means that for $w = 5$, the best way to make the value is with a single coin of denomination $c_5 = 5$. For $w = 6$, the two iterations of the inner for loop similarly compute $C[6] = 2$ and $P[6] = 1$.

For $w = 7$, the inner for loop will be executed three times. When $i = 0$, it will compute $C[7] = 7$ and $P[7] = 0$. When $i = 1$, it will compute $C[7] = C[6] + 1 = 3$ and $P[7] = 1$. When $i = 2$, it will compute $C[7] = C[0] + 1 = 1$ and $P[7] = 2$.

By running the algorithm, we can compute moving forward the remaining values for $C[w]$ and $P[w]$ as

$$C[8] = 2, C[9] = 3, C[10] = 2, C[11] = 3, C[12] = 2,$$
$$C[13] = 3, C[14] = 2, C[15] = 3, C[16] = 4, C[17] = 3, C[18] = 4$$
$$P[8] = 2, P[9] = 2, P[10] = 1, P[11] = 1, P[12] = 2,$$
$$P[13] = 2, P[14] = 2, P[15] = 1, P[16] = 1, P[17] = 1, P[18] = 1$$

From there, we can compute the actual solutions in the while loop of the algorithm. For $u = 18$, we compute $i \leftarrow P[18] = 1$, and then $S[1] = 1$ (since $S[i]$ was initialised to 0 for all i). Our new value of v will be $v - c_1 = 18 - 5 = 13$. Now $i \leftarrow P[13] = 2$, and $S[2] = 1$. Our new value of v will be $v - c_2 = 13 - 7 = 6$. We compute $i \leftarrow P[6] = 1$, and $S[2] = 1$. Our new value of v will be $v - c_1 = 6 - 5 = 1$. We compute $i \leftarrow P[1] = 0$ and $S[0] = 1$. Our new value of v will be $v - c_0 = 0$ and the while loop will terminate. We end up with $S[0] = 1, S[1] = 2, S[2] = 3$, i.e., the minimum number of coins to make the number 18 is with one 1, two fives and 1 seven, which also matches the value $C[18] = 4$.

The proposed greedy algorithm does not produce an optimal solution. In particular, the algorithm would use two coins of demonination $c_2 = 7$, and four coins of denomination $c_0 = 1$, to achieve a sum of 18, for a total of six coins.

4. A *contiguous subsequence* of length k of a sequence S is a subsequence which consists of k consecutive elements of S. For instance, if S is $1, 2, 3, -11, 10, 6, -10, 11, -5$, then $3, -11, 10$ is a contiguous subsequence of S of length 3. Give an algorithm based on dynamic programming that, given a sequence S of n numbers as input, runs in linear time and outputs the contiguous subsequence of S of maximum sum. Assume that a subsequence of length 0 has sum 0. For the example above, the answer of the algorithm would be $10, 6, -10, 11$ with a sum of 17.

   Solution: We will use dynamic programming to design an algorithm that solves the contiguous subsequence problem. We will use $M[j]$ to denote the optimal solution (the sum of the subsequence of maximum sum) *ending at position* j. We will define $M[0] = 0$. To choose the value of $M[j]$ we have two options: either take the optimal subsequence ending in position $j - 1$ and add the element a_j, or only use the element a_j in the subsequence. The choice will be made based on the maximum of the two, which is the first term if $M[j-1]] \geq 0$ and the second term if $M[j-1] < 0$. We have the following relation:

$$M[j] = \max\{M[j] + a_j, a_j\},$$

   To find the contiguous subsequence S^* of maximum sum, we need to find the element i^* for which $M[i^*]$ is maximised. In dynamic programming fashion, we will calculate the values of $M[i]$ for all $i = 1, 2 \cdots, n$ using the recurrence relation above and the solution that we have computed to smaller problem. Since our array is 1-D, this calculation can be done in $O(n)$ time. Then we can compute the $\max_{i \in \{1,\ldots,n\}} M[i] = M[i^*]$. This way we can find were the optimal subsequence ends. To find where it starts, we can keep track of the lengths of the optimal subsequences. If we let $L[i]$ denote the *length* of the optimal subsequence ending at position i, we have that

$$L[i] = \begin{cases} L[i-1] + 1, & \text{if } M[i-1] \geq 0 \\ 1, & \text{if } M[i-1] < 0 \end{cases}$$

   Note that the conditions in the two cases above are exactly those that determine the value of $M[j] = \max\{M[j] + a_j, a_j\}$.