Dynamic Programming I

Longest Monotone Subsequence Problem

Input: $d, a_1, a_2, \ldots, a_d \in \mathbb{N}$

Problem: Find L = length of the longest monotone non-decreasing subsequence.

Note: the subsequence need not be consecutive. Thus $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ is a monotone subsequence provided $1 \leq i_1 < i_2 < \ldots < i_k \leq d$ and $a_{i_1} \leq a_{i_2} \leq \ldots \leq a_{i_k}$.

Example 1 The length of the longest monotone subsequence of $\{4, 6, 5, 9, 1\}$ is 3.

We want to give a dynamic programming solution to the LMS Problem:

1.) Define an array of subproblems: $R(j)$ = length of the longest monotone subsequence which ends in a_j. Then the answer would be $L = \max_{1 \leq j \leq n} R(j)$.

2.) Find a recurrence: let $R(1) = 1$, and for $j > 1$,

$$R(j) = \begin{cases} 1 & \text{if } a_i > a_j \text{ for all } i \ 1 \leq i < j \\ 1 + \max_{1 \leq i < j}\{R(i) | a_i \leq a_j\} & \text{otherwise} \end{cases}$$

3.) Write an algorithm that computes R:

```
R(1) ← 1
for j : 2..d
    max ← 0
    for i : 1..j − 1
        if R(i) > max and a_i ≤ a_j then
            max ← R(i)
        end if
    end for
    R(j) ← max +1
end for
```

and from R it is possible to compute L as was indicated in 1.).

[1] The notes for the *Dynamic Programming Algorithm for the All pairs Shortest Path Problem* and *Dynamic Programming Algorithm for the Simple Knapsack Problem* were taken originally by Eugenia Ternovskaia in Steven Cook's csc364 lecture, Spring 99, and they have been edited for this term.

All Pairs Shortest Path Problem

Input: Directed graph $G(V, E)$, $V = \{1, 2, ..., n\}$. Cost $C(i, j) \in \mathbb{R}^+ \cup \{\infty\}$ (think of the costs as distances), $1 \leq i, j \leq n$, $C(i, j) = \infty$ if (i, j) is not an edge.

Problem: Find $D(i, j)$, the length of the shortest directed path from i to j.

Note that for a general graph on n nodes there are exponentially many (in n) paths, so exhaustive search is not feasible.

A general way of finding a Dynamic Programming solution to a problem:

1. Define a class of subproblems.
2. Give a recurrence based on solving each subproblem in terms of simpler subproblems.
3. Give an algorithm for computing the recurrence.

For our problem:

1.) Define an array of subproblems: Let $A(k, i, j)$ be the length of the shortest path from i to j s.t. all *intermediate* nodes on the path are in $\{1, 2, ..., k\}$. Then $A(n, i, j) = D(i, j)$ is the solution. Notational convention: if $k = 0$ then $\{1, 2, ..., k\} = \emptyset$.

2.) Define a recurrence: Initialization: for $k = 0$, $A(0, i, j) = C(i, j)$. Now for $k > 0$. To come up with a recurrence, notice that the shortest path between i and j either includes k or does not. Assume we know $A(k-1, r, s)$ for all r, s. Suppose node k is not included. Then, obviously, $A(k, i, j) = A(k-1, i, j)$. If node k occurs on a shortest path, then it occurs exactly once, so $A(k, i, j) = A(k-1, i, k) + A(k-1, k, j)$. The shortest path length is obtained by taking the minimum of these two cases:

$$A(k, i, j) = \min\{A(k-1, i, j), A(k-1, i, k) + A(k-1, k, j)\}.$$

3.) Write an algorithm:

It turns out that we only need space for a two-dimensional array $B(i, j) = A(k, i, j)$. To compute $A(k, *, *)$ from $A(k-1, *, *)$, we can overwrite $A(k-1, *, *)$. The following algorithm is known as *Floyd's algorithm*:

for $i : 1..n$ do
    for $j : 1..n$ do
        $B(i,j) \longleftarrow C(i,j)$ (*Initialize array B, i.e. copy C there*)
    end for
end for

for $k : 1..n$ do
(*at this point* $B(i,j) = A(k-1,i,j)$ *for all* i,j)
    for $i : 1..n$ do
        for $j : 1..n$ do
            $B(i,j) \longleftarrow \min\{B(i,j), B(i,k) + B(k,j)\}$
        end for
    end for
end for
$D \longleftarrow B$
return D

The running time of this algorithm is $O(n^3)$.

Why does it work? Worry: $B(i,k)$ or $B(k,j)$ may have already been updated, e.g. $k < j$ or $k < i$. But $B(i,k)$ and $B(k,j)$ are unchanged because the shortest path from i to k does not contain k as an intermediate node.

Simple Knapsack Problem (SKS)

Input: $d, w_1, ..., w_d, C \in \mathbb{N}$ (where C is the capacity)

Output: $\max_S\{K(S)|K(S) \leq C\}$, where $S \subseteq \{1, ..., d\}$, $K(S) = \sum_{i \in S} w_i$.

This is an NP-hard problem, which means that we cannot expect to find a polynomial time algorithm that works in general. We'll give a Dynamic Programming solution that works for relatively small C. It is important that the inputs $w_1, ..., w_d, C$ are integers (and in fact positive integers).

We will follow our standard three steps.

1.) Define an array of subproblems:

Only consider first i objects $1..i$ summing up to an *intermediate* weight limit j.

$$R(i,j) = \begin{cases} True & \text{if } \exists S \subseteq \{1, ..., i\} \text{ s.t. } K(S) = j, \text{ i.e., the sum of weights is exactly } j. \\ False & \text{otherwise} \end{cases}$$

$0 \leq i \leq d$, $0 \leq j \leq C$. We can compute the solution from R as follows: $M = \max_{j \leq C}\{j | R(d,j) = True\}$.

2.) Define a recurrence: Initialize: $R(0,j) = False$, $j > 0$; $R(i,0) = True$, $i = 0, 1, ..., d$.

Before writing a recurrence for $i, j \geq 1$, consider the following: Suppose we do not include object i. Then, obviously, $R(i,j) = True$ iff $R(i-1,j) = True$. Suppose object i is included. Then it must be the case that $R(i,j) = True$ iff $R(i-1, j-w_i) = True$, $j - w_i \geq 0$ (i.e., there is a subset S of objects $\{1, ..., i-1\}$ s.t. the sum of weights $K(S)$ is exactly $j - w_i$, $j \geq w_i$).

We obtain the following recurrence for $i, j \geq 1$: $R(i,j) = True$ iff $R(i-1,j) = True$ or ($j \geq w_i$ and $R(i-1, j-w_i) = True$).

3.) Write an algorithm:

We will use the same space saving trick as before by using one-dimensional array $S(j)$ for $R(i,j)$.

$S(0) \longleftarrow True$ (*Initialize array for* $i = j = 0$)
for $j : 1..C$ do
    $S(j) \longleftarrow False$ (*Initialize array for* $i = 0$ *and* $j = 1..C$)
end for
for $i : 1..d$
    for decreasing $j : C$ downto 1
        if ($j \geq w_i$ and $S(j - w_i) = True$) then
            $S(j) \longleftarrow True$
        end if
    end for
    (*at this point* $S(j) = R(i,j)$ *for all* j)
end for

Note that overwriting does not create a problem here, because the values of j are considered in decreasing order $C, C-1, ..., 1$. Thus the array position $S(j - w_i)$ has not yet been updated when the reference is made.

Exercise: Find an input for which the program would make an error if the inner loop "for decreasing $j : C..1$" were changed to "for $j : 1..C$".

The assertion $S(j) = R(i,j)$ can be proved by induction on the number of times the inner loop body is executed. This assertion implies that upon termination of the algorithm, $S(j) = R(d,j)$ for all j. From this, the solution M can be computed as indicated above.

The time complexity of the above algorithm is $O(dC)$. Is this polynomial time (in the length of an input instance $d, w_1, ..., w_d, C$, where $w_i \leq C$) ? Use binary notation for the input.

The length of the input is $n = O(d \log C)$. Is the time of the execution $O(dC)$ polynomial in the length of the input $O(d \log C)$? NO! For example, the binary representation of $C = 2^{100}$ has 101 bits. Thus, if we have an input of say 100 numbers of 100 bits each, then execution will take more than 2^{100} steps, which in impossibly long. (In fact no one knows how to solve this problem feasibly for such inputs.) Notice however that this algorithm is polytime if C is written in unary notation, i.e., $\underbrace{1.....1}_{C}$. In this case, large values of C (such as 2^{100}) cannot be written.