Dynamic Programming

Overview

Dynamic programming is the paradigm in which, given a problem, a directed graph of subproblems is defined. These subprograms are then solved in topological order. The inputs for any subproblem may include the outputs of preceding subproblems. The solution to the orginal problem is then computed from the solutions to the subproblems.

The hardest part of doing dynamic programming is realizing that your problem should be worked that way. I have seen very smart students fail to see this. The second hardest part is identifying the subproblems. Thirdly, you must write a program to compute a solution to each subproblem, given the solutions to prior subproblems.

The subproblems may fall into different classes, requiring different algorithms rather than a uniform algorithm for all. We can see that in some of the problems below.

Paragraph Breaking

Given a sequence of n words, way, you need to decide on line breaks to optimize the resulting paragraph. Let w_i be the ith word, and x_i its length. i^{th} word. Your are given a messure of the "goodness" of any paragraph. Your job is to find the "best" possible paragraph given that sequence of words. We frequently measure this goodness in reverse: we define the "penalty" for each possible line length and try to find the paragraph with minimum penalty. The output of your problem is an increasing sequence $(i_1, \ldots i_k$ such that the k^{th} line of the best paragraph ends at the i_k^{th} word.

The time complexity of any paragraph breaking algorithm is $\Omega(n)$, since the algorithm must look at each word at least once.

Trivial greedy algorithm. Suppose there is a maximum length L of any line, and the penalty of any line is L minus the length of that line, except for the last line, which has no penalty. The trivial algorithm is to end each line at the last possible position. This algorithm takes $\Theta(n)$ time. Does it create the best paragraph?

General penalty. The greedy algorithm, which you use when typing, generally results in a ragged right margin. For more sophisticated situations, such as books and published articles, a line can be compressed or stretched. The penalty function is designed to make the right-hand margin look smoother. Let L be the ideal total length of the words on a line, and M the maximum allowed. The penalty is zero if the length of the line is L, and is larger if that length deviates from M in either direction, except for the last line. The penalty of the paragraph is then the sum of the penalties of the lines. There are n subproblems. For each $1 \leq i \leq n$, define $P[i]$ to be the paragraph consisting of the words $w_i \ldots w_n$. The subprograms are worked in reverse order, starting with $P[n]$ and ending with $P[1]$, the solution to the original problem. The time complexity depends on the penalty function.

Editing Distance. Given two strings u and v, of lengths n and m, respectively, what is the cost of the changes that need to be made to u to make it v? We call that the editing distance from u to v. An edit consists of a sequence of small changes, each of which has a given cost, and the goal is to minimize that cost. Levenshtein distance is only one example. Let $u[1 \ldots i]$ prefix of u consisting of the first i symbols; similarly $v[1 \ldots j]$. If the editing cost function is simple, such as for Levenshtein distance, we define the subproblem $S[i, j]$ for all $0 \leq i \leq n$, $0 \leq j \leq m$ to be the cost of changing $u[1 \ldots i]$ to $v[1 \ldots j]$. If the cost function is reasonably simple, such as in the calse of Levenshtein distance, we let $S[0, 0] = 0$, then compute $S[i, j]$ from the values of $S[i-1, j-1]$, $S[i-1, j]$, and $S[i, j-1]$.

Shortest or Longest Path. Given a directed acyclic graph G, we have two closely related problems.

1. Given a "source" vertex s of G and a "target" vertex t, find the shortest directed path through G from s to t.
2. Find the longest directed path through G.

Either of these problems can be enhanced by assigning a weight to each arc, measuring the cost of a path instead of simply its number of arcs.

For either problem, let $S[i]$ be the minimum length of any path from the source to i for problem (a), or the maximum length of any path which ends at j for problem (b). The time complexity is $O(m)$, Where m is the number of arcs of G, provided we visit the vertices in topological order.

Remark. The Levenshtein edit distance problem reduces to (a).

Remark. The longest increasing subsequence problem reduces to (b).

Longest Monotone Subsequence. Given a sequence of numbers $\sigma = x_1, x_2, \ldots x_n$, find a longest strictly monotone increasing subsequence of σ. For example, $\sigma = 3, 2, 1, 6, 5, 4, 9, 8, 7$ The greatest length of any monotone increasing sequence of σ is 3, and there are many subsequences of that length, such as $1, 6, 8$.

Reduction of the LMS problem to the maximum length path problem. Let $G = (V, E)$ be the directed graph where $V = \{1, 2, 3, \ldots n\}$ and $E = \{i, j) : i < j \text{ and } x[i] < x[j]\}$

The longest increasing subsequence problem can be solved in $O(n^2)$ time, using a fairly simple algorithm. But there is a more sophisticated algorithm that takes $O(n \log n)$ time.

Maximum Contiguous Subsequence. Let $x_1, \ldots x_n$ be a sequence of numbers, both positive and negative. The problem is to select a contiguous subsequence of maximum sum, that is, to choose $1 \leq k \leq \ell \leq n$ to maximize $\sum_{i=k}^{\ell}$. The dynamic program has two kinds of subproblems: $A[i]$ = the maximum sum of any contiguous subsequence of σ whose last term is x_i. $B[i]$ = the maximum sum of any contiguous subsequence of $\sigma[1 \ldots i]$

```
A[i] = x[i]+max(0,A[i-1]);
B[i] = max(A[i],B[i-1]);
```

The solution is $B[n]$.

Coin Row Problems There are two different dynamic programs for the standard coin-row problem. Let $x[1], x[2], \ldots x[n]$ be the values of the coins, where $x[i] > 0$ for all i. Recall that the problem is to select a set of coins of maximum value, subject to the condition that no two consecurive coins can be selected.

1. Let $T[i]$ be the maximum value of a subset of the first i coins, subject to the condition that no two consecurive coins can be selected.

```
int findmax()
 {
  T[0] = 0;
  T[1] = x[1];
  for(int i = 2; i <= n; i++)
   T[i] = max(T[i-1],T[i-2]_x[i]);
  return T[n];
 }
```

2. Alternatively, let $T[i]$ be the maximum value of any set which contains the i^{th} coin, subject to the condition that no two consecurive coins can be selected.

```
int findmax()
 {
  T[0] = 0; // even though there is no zeroth coin
  T[1] = x[1];
  T[2] = x[2];
  for(iint i = 3; i <= n; i++)
   T[i] = T[i] + max(T[i-2],T[i-3])
  return max(T[n-1],T[n]);
 }
```