Columbia University
W4231: Analysis of Algorithms
Professor Luca Trevisan

Handout 12
November 9, 1999

Notes on Dynamic Programming

See CLR Section 16.1, 16.2 and 16.3 for an introduction to dynamic programming and two examples. Here we only discuss three problems that are not covered in the book

1 Subset sum

Description of the problem. Given n items of "size" $l_1, \ldots, l_n$ (positive integers) and a bound B (non-negative integer), decide whether there is a subset $S \subseteq \{1, \ldots, n\}$ of the items such that their total size equals B, i.e. $\sum_{i \in S} l_i = B$.

The problem has the following recursive solution:

- Base Case: $n = 0$. The answer is `no`, unless $B = 0$.
- General Case. The answer is `yes` iff
  - either there is a subset $S \subseteq \{1, \ldots, n-1\}$ such that $\sum_{i \in S} l_i = B$ (in which case S is the solution),
  - or there is a subset $S \subseteq \{1, \ldots, n-1\}$ such that $\sum_{i \in S} l_i = B - l_n$ (in which case $S \cup \{n\}$ is the solution).

A divide-and-conquer algorithm based on this recursive solution has a running time given by the recurrence

$$T(n) = 2T(n-1) + O(1) \ , \ T(1) = O(1)$$

which gives $T(n) = \Omega(2^n)$.

However, there are only Bn problems that one needs to solve. Namely, it is enough to compute for every $1 \leq k \leq n$ and every $1 \leq B' \leq B$ whether there is a subset $S \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in S} l_i = B'$.

A dynamic programming algorithm computes a boolean matrix $M[\cdot, \cdot]$ having the property that $M[k, B'] = True$ iff there is a subset $S \subseteq \{1, \ldots, k\}$ such that $\sum_{i \in S} l_i = B'$. (We save some memory and some computations by not including a 0-th row and a 0-th column in the matrix.)

The algorithm computes the matrix iteratively. Initially, the first row of the matrix is initialized by setting $M[1, l_1] = True$ and $M[1, B'] = 0$ for $B' \neq l_1$. Then the matrix is filled for k going from 2 to n and for B' going from 1 to B. The entry $M[k, B']$ is computed as the OR of the entries $M[k-1, B']$ and $M[k-1, B' - l_k]$. If the entry $M[k-1, B' - l_k]$ does not exist (because $B' \leq l_k$) then we simply set $M[k, B'] = M[k-1, B']$.

Each entry is computed in constant time, therefore the matrix is filled in $O(nB)$ time. Once the matrix is filled, the answer to the problem is contained in $M[n, B]$.

2 Knapsack

Description of the problem. Given n items of "volume" $v_1, \ldots, v_n$ and "cost" $c_1, \ldots, c_n$, and a volume bound B; Find a subset $S \subseteq \{1, \ldots, n\}$ of the items of total volume at most B (i.e. $\sum_{i \in S} v_i \leq B$) such that the total cost $cost(S) = \sum_{i \in S} c_i$ is maximized.

The problem has the following recursive solution.

- Base Case $n = 1$.
  - If $v_1 > B$, then the only possible solution is the empty set, of cost zero.
  - If $v_1 \leq B$, then the optimum solution is $\{1\}$, of cost c_1.
- General Case. An optimum solution is given by the better of the following alternatives:
  - An optimum solution S for the instance containing the first $n-1$ items and with volume bound B.
  - $S \cup \{n\}$ where S is an optimum solution for the instance containing the first $n-1$ items and with volume bound $B - v_k$.

We can construct a $n \times (B+1)$ matrix $M[\cdot, \cdot]$ such that for every $1 \leq k \leq n$ and $0 \leq B' \leq B$, $M[k, B']$ contains the cost of an optimum solution for the instance that uses only a subset of the first k elements (of volume $v_1, \ldots, v_k$ and cost $c_1, \ldots, c_k$) and with volume bound B'. Matrix M has the following recursive definition.

- $M[1, B'] = c_1$ if $B' \geq v_1$; $M[1, B'] = 0$ otherwise;
- For every $k, B' \geq 1$,

$$M[k, B'] = \max\{M[k-1, B'] \ , \ M[k-1, B' - v_k] + c_k\} \ .$$

An iterative algorithm can fill the matrix by computing each entry in constant time. At the end, the cost of an optimum solution is reported in $M[n, B]$. The time needed by an algorithm to fill the table is $O(nB)$.

In order to *construct* an optimum solution, we can also build a Boolean matrix $C[\cdot, \cdot]$ of the same size $n \times (B+1)$.

For every $1 \leq k \leq n$ and $0 \leq B' \leq nB$, $C[k, B'] = True$ iff there is an optimum solution that packs a subset of the first k items in volume B' so that item k is included in the solution.

Using matrix C we can work backwards to reconstruct the elements present in an optimum solution.

As an example, consider an instance with 9 items and volume bound 15. The costs and the volumes of the items are as follows:

Item	1	2	3	4	5	6	7	8	9
Cost	2	3	3	4	4	5	7	8	8
Volume	3	5	7	4	3	9	2	11	5

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k = 9$	0	0	7	7	7	11	11	15	15	15	19	19	19	21	23	23
$k = 8$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k = 7$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k = 6$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k = 5$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k = 4$	0	0	0	2	4	4	4	6	6	7	7	7	9	9	9	9
$k = 3$	0	0	0	2	2	3	3	3	5	5	5	5	6	6	6	8
$k = 2$	0	0	0	2	2	3	3	3	5	5	5	5	5	5	5	5
$k = 1$	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k = 9$	0	0	0	0	0	0	0	1	1	0	1	1	1	1	1	1
$k = 8$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k = 7$	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$k = 6$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k = 5$	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1
$k = 4$	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
$k = 3$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
$k = 2$	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
$k = 1$	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

The optimum solution is as follow:

Optimum: 23
Item 9 Cost 8 Volume 5
Item 7 Cost 7 Volume 2
Item 5 Cost 4 Volume 3
Item 4 Cost 4 Volume 4

3 Edit distance

A *string* is a sequence $a_1 a_2 \cdots a_n$ of characters. The length of a string is the number of characters in the sequence. As a convention, we call λ the *empty* string consisting of zero characters.

Given two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ we want to compute what is the minimum number of "errors" (edit operations) needed to transform x into y.

Possible operations are:

- insert a character.
  $insert(x, i, a) = x_1 x_2 \cdots x_i a x_{i+1} \cdots x_n$.
- delete a character.
  $delete(x, i) = x_i x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$.
- modify a character.
  $modify(x, i, a) = x_1 x_2 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

The problem admits a recursive characterization.

To transform $x_1 \cdots x_n$ into $y_1 \cdots y_m$ we have three choices:

Put y_m at the end: we first use an insert operation to transform x into $x_1 \cdots x_n y_m$ and then we recursively find the best way of transforming $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.

Delete x_n: we first use a delete operation to transform x into $x_1 \cdots x_{n-1}$ and then we recursively find the best way of transforming $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.

Change x_n into y_m: if x_n and y_m are different, we first use a modify operation to transform x into $x_1 \cdots x_{n-1} y_m$ and then we recursively find the best way of transforming $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$ (if $x_n = y_m$ we need not do the modify operation at the beginning).

A dynamic programming solution can be obtained as follows. Given strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$, we define an $(n+1) \times (m+1)$ matrix $M[\cdot, \cdot]$. For every $0 \le i \le n$ and $0 \le j \le m$, $M[i, j]$ is the minimum number of operations to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$. Matrix $M[\cdot, \cdot]$ can be computed iteratively based on the following relations:

- $M[0, j] = j$ because the only way to transform the empty string into $y_1 \cdots y_j$ is to add the j characters $y_1, \ldots, y_j$.
- $M[i, 0] = i$ for similar reasons.
- For $i, j \ge 1$,

$$
\begin{aligned}
M[i,j] \quad = \min\{ \quad & M[i-1, j] + 1, \\
& M[i, j-1], \\
& M[i-1, j-1] + change(x_i, y_j)\}
\end{aligned}
$$

  where $change(x_i, y_j) = 1$ if $x_i \ne y_j$ and $change(x_i, y_j) = 0$ otherwise.

So the problem of computing the edit distance between two strings can be computed in $O(nm)$ time.