CS 4310 Algorithms: V. Dynamic Programming

E. de Doncker[1]

[1] Dept. of Computer Science, WMU, U.S.A.

Feb. 15, 2024

Outline

Outline

1. Dynamic programming method
   - Optimal decision sequence
   - Principle of optimality
2. All pairs shortest path problem
3. 0/1 Knapsack problem
   - Forward method
   - Backward method
4. Traveling salesman problem (TSP)
5. String editing

Outline

1. Dynamic programming method
   - Optimal decision sequence
   - Principle of optimality
2. All pairs shortest path problem
3. 0/1 Knapsack problem
   - Forward method
   - Backward method
4. Traveling salesman problem (TSP)
5. String editing

Outline

1. Dynamic programming method
   - Optimal decision sequence
   - Principle of optimality
2. All pairs shortest path problem
3. 0/1 Knapsack problem
   - Forward method
   - Backward method
4. Traveling salesman problem (TSP)
5. String editing

Outline

1. Dynamic programming method
   - Optimal decision sequence
   - Principle of optimality
2. All pairs shortest path problem
3. 0/1 Knapsack problem
   - Forward method
   - Backward method
4. Traveling salesman problem (TSP)
5. String editing

Optimal decision sequence

This material is based on the text by Horowitz et al. [1].
Dynamic Programming is an algorithm paradigm that is suitable when the solution of a problem can be seen as a sequence of decisions.
Some such problems are covered under greedy methods:
– Greedy knapsack: The sequence of decisions determines which item to add next, and thus the consecutive values of the fractions x_i. An optimal sequence maximizes the total profit and satisfies the constraints.
– Optimal merge patterns: As a decision sequence, an optimal merge pattern determines which pair of files to merge next at each step, in order to minimize the weighted external path length of the 2-way (binary) merge tree.
– Huffman trees: The decision sequence determines an optimal merge pattern by deciding which subtrees to combine at each step, in order to minimize the weighted external path length and thus the average message or code length.
– Minimum spanning tree: The decision sequence determines which edge of the graph to add next in order to minimize the cost of the spanning tree.

Principle of optimality

– Shortest path problem: The decision sequence determines which vertex to go to next at each step in order to minimize the distance along the path. A greedy algorithm exists for the *Single source (all destinations) shortest path problem* (Dijkstra's algorithm).

For problems that can be solved by a greedy method, the decisions can be made one at a time without ever making a wrong choice, and leading to an optimal sequence; i.e., in a greedy algorithm, only one sequence is generated.

For problems where that is not possible, the other extreme is to construct all possible (feasible) sequences with their total cost or profit value, and choose the best one. This is an exhaustive search or total enumeration, generally prohibitively expensive.

A dynamic programming method may generate many sequences, but attempts to eliminate the construction of some sequences that cannot possibly be optimal, by relying on the:

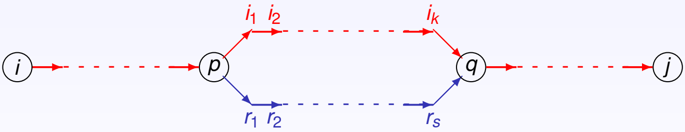
Principle of optimality: an optimal sequence cannot have suboptimal subsequences [1]

[1] *The term "suboptimal" is used for "below optimal" or not optimal.*

Principle of optimality

Example: The shortest path problem: involving a sequence of decisions as to which vertex to go to next at each step, in order to minimize the length (cost) of the path.

To show that the principle of optimality holds we consider a shortest path from vertex i to vertex j in a directed graph. Then a subpath p to q must be a shortest p to q path; otherwise another (blue) p to q path would exist, and its incorporation in the original i to j path would yield a shorter i to j path. This is a contradiction as the original i to j path was presumed a shortest path.



Note: In [1] (Example 5.5), p coincides with i, and q coincides with j; i.e., the paths $i_1, \ldots, i_k$ and $r_1, \ldots, r_s$ go between i and j.

All pairs shortest paths problem

$G = (V, E)$ is a directed graph with vertices labeled by $1, 2, \ldots, n$, and weight c_{ij} on edge (i, j) for all i, j (and $c_{ij} = \infty$ if the edge is not present), $1 \le i, j \le n$. It is assumed that G has no negative cycles.
The goal is to [find paths that] minimize the distance between any two vertices i and j.

The principle of optimality holds, leading to a dynamic programming method for the pairs shortest paths problem [1].

Starting with $A^0(i, j) = c_{ij}$, the dynamic programming algorithm constructs, at step k, the shortest paths that do not go through any vertices labeled higher than k:

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

Indeed, if shortest i to j path goes through k then $A^{k-1}(i, k) + A^{k-1}(k, j)$ is obtained, otherwise $A^{k-1}(i, j)$.

All pairs shortest paths - Algorithm (Floyd-Warshall)

Algorithm *Floyd-Warshall* (C, A, n) { // returns minimum path distances in A
// for input cost matrix C of graph G and size $n = |V|$

for ($i = 1; i \leq n; i{+}{+}$) // Initialize
  for ($j = 1; j \leq n; j{+}{+}$)
    $A[i][j] = C[i][j];$
for ($k = 1; k \leq n; k{+}{+}$)
  for ($i = 1; i \leq n; i{+}{+}$)
    for ($j = 1; j \leq n; j{+}{+}$)
      $A[i][j] = \min(A[i][j],\ A[i][k] + A[k][j]);$
}

Time complexity is $\mathcal{O}(n^3)$.

Compare to applying Dijkstra's algorithm (of $\mathcal{O}(n^2)$ for single source - all destinations) from each vertex to all other vertices: $\mathcal{O}(n \times n^2) = \mathcal{O}(n^3)$.

All pairs shortest paths - Example

Example: $n = 3$

$$C = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{pmatrix} = A^0 \qquad A^1 = \begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

For A^1, paths are allowed to go through vertex 1. Only the element $(3,2)$ changes, where $A^1(3,2) = \min\{\infty,\ A^0(3,1) + A^0(1,2)\} = \min\{\infty,\ 3+4\} = 7$.

$$A^2 = \begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix} \qquad A^3 = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$$

For A^2, paths are additionally allowed to go through vertex 2. Only the element $(1,3)$ changes, where $A^2(1,3) = \min\{11,\ A^1(1,2) + A^1(2,3)\} = \min\{11,\ 4+2\} = 6$.
For A^3, paths are additionally allowed to go through vertex 3. Only the element $(2,1)$ changes, where $A^3(2,1) = \min\{6,\ A^2(2,3) + A^2(3,1)\} = \min\{6,\ 2+3\} = 5$.

0/1 Knapsack problem

The 0/1 knapsack problem is as the greedy knapsack problem, except that the x_i are 0 or 1; thus an object is either not added or completely added to the knapsack.

Given n objects with profits p_i and weights w_i, and knapsack capacity M, the goal is to add objects to the knapsack in order to maximize the total profit, i.e.,

$$\textit{maximize} \sum_{i=1}^{n} p_i x_i \;\; \textit{subject to} \begin{cases} \sum_{i=1}^{n} w_i x_i \leq M \\ x_i = 0 \textit{ or } 1, \;\; 1 \leq i \leq n \end{cases}$$

To show that the principle of optimality holds, consider the problem $\mathrm{KNAP}(\ell, u, y)$ for the decision sequence in the range from ℓ to u, with remaining capacity y (see [1], Example 5.6):

$$\textit{maximize} \sum_{i=\ell}^{u} p_i x_i \;\; \textit{subject to} \begin{cases} \sum_{i=\ell}^{u} w_i x_i \leq y \\ x_i = 0 \textit{ or } 1, \;\; \ell \leq i \leq u \end{cases}$$

Using this notation, the entire knapsack problem is thus $\mathrm{KNAP}(1, n, M)$ for n objects and capacity M.

Principle of optimality

The principle of optimality holds for the 0/1 Knapsack problem [1].
If $y_1, y_2, \ldots, y_n$ is an optimal decision sequence assigning 0 or 1 to the corresponding $x_1, x_2, \ldots, x_n$, we show that $y_2, \ldots, y_n$ is an optimal subsequence:
For $y_1, y_2, \ldots, y_n$, either $y_1 = 0$ (first object left out) or $y_1 = 1$ (first object in) the knapsack.
If $y_1 = 0$, then $y_2, \ldots, y_n$ must be an optimal sequence for $\textsc{Knap}(2, n, M)$, otherwise $y_1, y_2, \ldots, y_n$ would not be an optimal sequence for the original problem.
If $y_1 = 1$, then $y_2, \ldots, y_n$ must be an optimal sequence for $\textsc{Knap}(2, n, M - w_1)$ (with the weigth of the first object subtracted from the original capacity M).
Otherwise, there must be another subsequence $z_2, \ldots, z_n$, which is feasible $(w_1 + \sum_{i=2}^{n} w_i z_i \leq M)$ and with higher profit, $\sum_{i=2}^{n} p_i z_i > \sum_{i=2}^{n} p_i y_i$;
but then the total profit of the new sequence would exceed that of the original sequence $(p_1 + \sum_{i=2}^{n} p_i z_i > p_1 + \sum_{i=2}^{n} p_i y_i)$. The original sequence was, however, assumed optimal, so the existence of $z_2, \ldots, z_n$ leads to a contradiction.

Forward method

The first decision, $x_1 = 0$ or 1, is represented by

$$g_0(M) = \max\{\, g_1(M),\, g_1(M - w_1) + p_1 \,\}$$

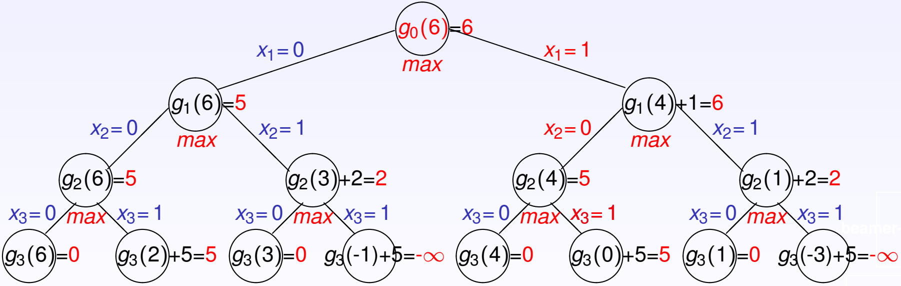
(see [1] Example 5.8), where $g_0(M)$ denotes the profit resulting from the decision; $g_1(M)$ is the profit incurred by leaving out the first object, and $g_1(M - w_1) + p_1$ is the profit when the first object is put in the knapsack.
We can traverse the computation as a tree; the value $g_0(M)$ is associated (as the profit value) with the root of the tree, and the g_1 values are at depth 1. The branches from the root are labeled with $x_1 = 0$ and $x_1 = 1$.
This process is continued, so that we can use the following recurrence,

at depth $i < n$: $\begin{cases} g_i(y) = \max\{\, g_{i+1}(y),\, g_{i+1}(y - w_{i+1}) + p_{i+1} \,\} & \text{for } y \geq 0 \\ g_i(y) = -\infty & \text{for } y \geq 0 \end{cases}$

and at depth n: $\begin{cases} g_n(y) = 0 & \text{for } y \geq 0 \\ g_n(y) = -\infty & \text{for } y < 0 \end{cases}$ Nodes with value $-\infty$ will not be selected in the sequence, so their subtree will be cut off. The tree is developed top-down, followed by the profit calculation bottom-up.

Forward method - Example

Example: $n = 3$ objects, capacity $M = 6$,
$(p_1, p_2, p_3) = (1, 2, 5)$, $(w_1, w_2, w_3) = (2, 3, 4)$
The forward method first makes the decision on x_1, then x_2, then x_3 (i.e., looks forward on the decision sequence).



Solution: $(x_1, x_2, x_3) = (1, 0, 1)$ with total profit = $g_0(6) = 6$

Backward method

The decision sequence is constructed by looking backward [1] (Example 5.13), i.e., first the decision is made on $x_n = 0$ or 1, then x_{n-1}, etc. Thus the computation tree is assigned

$$f_n(M) = \max\{f_{n-1}(M),\ f_{n-1}(M - w_n) + p_n\}$$

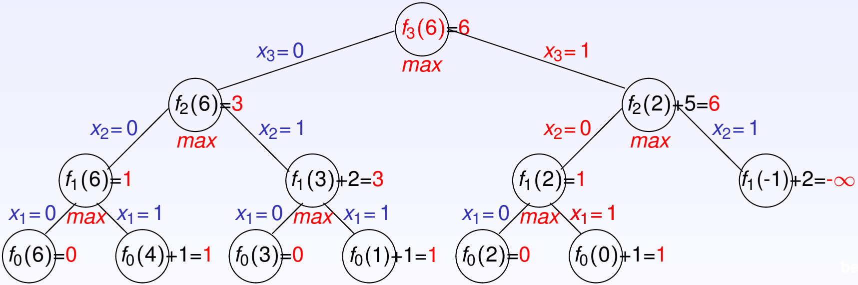
at the root, and the process is continued according to the following recurrence:

at depth $i < n$: $\begin{cases} f_i(y) = \max\{f_{i-1}(y),\ f_{i-1}(y - w_i) + p_i\} & \text{for } y \geq 0 \\ f_i(y) = -\infty & \text{for } y \geq 0 \end{cases}$

and at depth n: $\begin{cases} f_0(y) = 0 & \text{for } y \geq 0 \\ f_0(y) = -\infty & \text{for } y < 0 \end{cases}$

Backward method - Example

Example: $n = 3$ objects, capacity $M = 6$,
$(p_1, p_2, p_3) = (1, 2, 5)$, $(w_1, w_2, w_3) = (2, 3, 4)$



Solution: $(x_1, x_2, x_3) = (1, 0, 1)$ with total profit = $f_3(6) = 6$

0/1 knapsack dynamic programming method - complexity

The forward and backward methods may generate 2^n sequences, corresponding to all the leaf nodes of the full binary tree. This also corresponds to the total number of n-tuples $(x_1, x_2, \ldots, x_n)$ of $x_i = 0$ or 1, which would have to be generated by a total enumeration.

However, the dynamic programming strategy may avoid to generate large subtrees.

Traveling salesman problem (TSP)

Consider a directed graph $G = (V, E)$, with vertices labeled by $1, 2, \ldots, n$, and weight $c_{ij} > 0$ on edge (i, j) for all i, j (and $c_{ij} = \infty$ if the edge is not present).
A tour is a simple cycle (containing no repeated vertices except the first and the last) that visits all vertices of the graph. The cost of a tour is the sum of the weights on its edges. The goal of TSP is to construct a tour of minimum cost.

Without loss of generality we consider a tour from vertex 1 and returning to vertex 1. The tour starts with an edge $(1, k)$, followed by a simple path from k to 1 that visits all other vertices.
It can be seen that the principle of optimality holds.

Indeed, if the tour is optimal, then the path from k back to 1 has to be a shortest path that goes through all vertices in $V - \{1, k\}$ exactly once [1].

TSP - dynamic programming algorithm

Let $g(i, S)$ denote the cost of a simple path from vertex i, visiting all vertices in the set S and going back to vertex 1.
From the principle of optimality, the dynamic programming solution is given by the cost in going from vertex 1, through all vertices in $V - \{1\}$ and back to 1 as follows:

$$g(1, V - \{1\}) = \min_{2 \leq k \leq n}\{\, c_{1k} + g(k, V - \{1, k\})\,\}$$

The g-function in the right hand side will be obtained using the generalized expression for a vertex $i \notin S$:

$$g(i, S) = \min_{j \in S}\{\, c_{ij} + g(j, S - \{j\})\,\}$$

TSP - Example

Example:

Graph $G = (V, E)$ with cost matrix $C = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$

The optimal tour cost is:

$g(1, \{2,3,4\}) = \min\{\, c_{12} + g(2, \{3,4\}),\ c_{13} + g(3, \{2,4\}),\ c_{14} + g(4, \{2,3\}) \,\}$

$\quad g(2, \{3,4\}) = \min\{\, c_{23} + g(3, \{4\}),\ c_{24} + g(4, \{3\}) \,\}$

$\qquad g(3, \{4\}) = c_{34} + g(4, \emptyset) = c_{34} + c_{41} = 12 + 8 = 20$

$\qquad g(4, \{3\}) = c_{43} + g(3, \emptyset) = c_{43} + c_{31} = 9 + 6 = 15$

$\Rightarrow g(2, \{3,4\}) = \min\{\, 9 + 20,\ 10 + 15 \,\} = 25$

$\quad g(3, \{2,4\}) = \min\{\, c_{32} + g(2, \{4\}),\ c_{34} + g(4, \{2\}) \,\}$

$\qquad g(2, \{4\}) = c_{24} + g(4, \emptyset) = c_{24} + c_{41} = 10 + 8 = 18$

$\qquad g(4, \{2\}) = c_{42} + g(2, \emptyset) = c_{42} + c_{21} = 8 + 5 = 13$

$\Rightarrow g(3, \{2,4\}) = \min\{\, 13 + 18,\ 12 + 13 \,\} = 25$

TSP - Example

$$g(4,\{2,3\}) = \min\{\, c_{42} + g(2,\{3\}),\ c_{43} + g(3,\{2\})\,\}$$
$$g(2,\{3\}) = c_{23} + g(3,\emptyset) = c_{23} + c_{31} = 9 + 6 = 15$$
$$g(3,\{2\}) = c_{32} + g(2,\emptyset) = c_{32} + c_{21} = 13 + 5 = 18$$
$$\Rightarrow g(4,\{2,3\}) = \min\{\, 8 + 15,\ 9 + 18\,\} = 23$$

$\Rightarrow$ Cost for shortest tour: $g(1,\{2,3,4\}) = \min\{\, 10 + 25,\ 15 + 25,\ 20 + 23\,\} = 35$

Path:
The minimum for $g(1,\{2,3,4\})$ is obtained via $10 + 25 = c_{12} + g(2,\{3,4\})$, thus the path goes from 1 to 2.
The minimum for $g(2,\{3,4\})$ is obtained via $10 + 15 = c_{24} + g(4,\{3\})$, thus the path goes from 2 to 4.
$g(4,\{3\})$ is obtained via $c_{43} + g(3,\emptyset)$, thus the next vertex is 3.

Therefore the tour is: $1 \to 2 \to 4 \to 3 \to 1$.

TSP - dynamic programming algorithm complexity

Complexity for $n = |V|$ (= number of vertices in G)

Let $N =$ number of $g(i, S)$ values computed in order to obtain $g(1, V - \{1\})$:

Number of distinct sets S of size k not including i and 1: $\binom{n-2}{k}$

and $n - 1$ possible values of i (excluding 1)

$\Rightarrow N \leq (n-1) \sum_{k=0}^{n-2} \binom{n-2}{k} = (n-1)\, 2^{n-2} \Rightarrow N = \mathcal{O}(n\, 2^n)$

Note: Show that $\sum_{k=0}^{n-2} \binom{n-2}{k} = 2^{n-2}$:

Binomial expansion: $(a+b)^\eta = \sum_{k=0}^{\eta} \binom{\eta}{k} a^k b^{\eta-k}$

Setting $a = b = 1$: $2^\eta = \sum_{k=0}^{\eta} \binom{\eta}{k}$, now set $\eta = n - 2$.

Since each $g(i, S)$ requires taking a minimum of $\mathcal{O}(n)$ values,
the overall time complexity is $\mathcal{O}(n\, N) = \mathcal{O}(n^2\, 2^n)$. – *Compare to a total enumeration of all possible tours of* $(n-1)$ *vertices* $(2, 3, \ldots, n)$*:* $(n-1)!$ *permutations.*

The space complexity (for storage of all $g(i, S)$) is $\mathcal{O}(N) = \mathcal{O}(n\, 2^n)$.

String editing [1] (Section 5.6)

Given strings $X = x_1 x_2 \dots x_n$ and $Y = y_1 y_2 \dots y_m$, composed of symbols from a given alphabet, the objective is to transform X into Y using a minimum cost sequence of edit operations.
The edit operations and associated costs are:
– delete symbol x_i from X, at cost $D(x_i)$;
– insert symbol y_j into X, at cost $I(y_j)$;
– change symbol x_i from X into y_j, at cost $C(x_i)$.
The cost of a sequence of edit operations is the sum of the costs for all individual symbols.

Example: Let $X = x_1 x_2 x_3 x_4 x_5 = aabab$, $Y = y_1 y_2 y_3 y_4 = babb$, and the costs for deletion, insertion and change are 1, 1, and 2, respectively. Different sequences for transforming X into Y are:
(i) delete all symbols of X, then insert all symbols of Y one by one (at total cost $5 + 4 = 9$);
(ii) delete x_1 and x_2 yielding $X = bab$ and insert $y_4 = b$ at the end (at cost $1 + 1 + 1 = 3$);
(iii) change x_1 to y_1 yielding $X = babab$ and then delete $x_4 = a$ (at cost $2 + 1 = 3$).

String editing [1] (Section 5.6)

The principle of optimality holds for string editing. Indeed, if S is a minimum cost sequence that transforms X into Y, then the subsequence X' of X from the second edit operation constitutes again a minimum cost sequence for transforming X' into Y.

Algorithm derivation

Let $cost(i,j)$ be the minimum cost of any edit sequence to transform $x_1 x_2 \ldots x_i$ into $y_1 y_2 \ldots y_j$ for $1 \leq i \leq n,\ 1 \leq j \leq m$.
Deriving the $cost(i,j)$ values will yield the minimum cost at $cost(n,m)$.
For $X = Y = \lambda$ (= empty string), $cost(0,0) = 0$;
if $j = 0$ and $i > 0$, X is transformed into Y by a sequence of deletions, and $cost(i,0) = cost(i-1,0) + D(x_i)$;
if $j > 0$ and $i = 0$, X is transformed into Y by a sequence of insertions, and $cost(0,j) = cost(0,j-1) + I(y_j)$;

if $i \neq 0$ and $j \neq 0$, $cost(i,j)$ is obtained as the minimum obtained via a symbol deletion, insertion or change.

String editing [1] (Section 5.6)

The algorithm generates the $n \times m$ array *cost* using the following recurrence:

$$cost(i,j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1,0) + D(x_i) & j = 0, i > 0 \\ cost(0,j-1) + I(y_j) & i = 0, j > 0 \\ \min\{\, cost(i-1,j) + D(x_i),\ cost(i-1,j-1) + C(x_i,y_j), & \\ \qquad cost(i,j-1) + I(y_j)\,\} & i,j > 0 \end{cases}$$

The time complexity is $\mathcal{O}(mn)$.

String editing [1] (Section 5.6)

Example result:

			b	a	b	b
	i/j	0	1	2	3	4
	0	0	1	2	3	4
a	1	1	2	1	2	3
a	2	2	3	2	3	4
b	3	3	2	3	2	3
a	4	4	3	2	3	4
b	5	5	4	3	2	3

The array is generated starting with $cost(0,0)$, then the first row and first column, then consecutive rows (each from left to right).

$$\begin{aligned} cost(1,1) &= \min\{\, cost(0,1) + D(x_1),\ cost(0,0) + C(x_1,y_1),\ cost(1,0) + I(y_1)\,\} \\ &= \min\{\, 2,2,2 \,\} = 2 \\ cost(1,2) &= \min\{\, cost(0,2) + D(x_1),\ cost(0,1) + C(x_1,y_2),\ cost(1,1) + I(y_2)\,\} \\ &= \min\{\, 3,1,3 \,\} = 1 \\ &\ldots \end{aligned}$$

String editing [1] (Section 5.6)

The minimum cost is $cost(n, m) = cost(5, 4) = 3$.
The red path in the array corresponds to *(ii)* listed in the Example before.

The blue path plus the first and last elements (at $(0, 0)$ and $(5, 4)$) correspond to *(iii)* of the Example.

Bibliography

S. Sahni E. Horowitz and S. Rajasekeran.
Computer Algorithms/C++.
Computer Science Press, 2nd edition, 1998.
ISBN 0-7167-8315-0.