Dynamic Programming Algorithms for Some Problems

Dr. Gur Saran Adhar

Reference clrs, Chapter 15

Approach: Dynamic Programming

Dynamic Programming: is a general technique which can be used to solve many optimization problems that exhibit **optimal substructure**. That is, an optimal solution to the problem contains within it optimal solution to sub-problems.
Solution of a large problem can be found by examining the solution of smaller sub-problems.

`Reference clrs, Chapter-15, Page-339`

List Of Some Example Problems

1.	Assembly Line Scheduling	clrs 324-
2.	Matrix Multiply	clrs 331-
3.	Longest Common Subsequence	clrs 350-
4.	0-1 Knap Sack	clrs 382-, and Udi Manber 108-
5.	Optimal Polygon Triangulation	
6.	Optimal Binary Search Tree	clrs 356
7.	Edit Distance	clrs 364 Udi Manber 156-

Approach: Dynamic Programming

Problem: Assembly Line Scheduling:
There are two assembly lines, each with n stations. Find the fastest way through the factory.

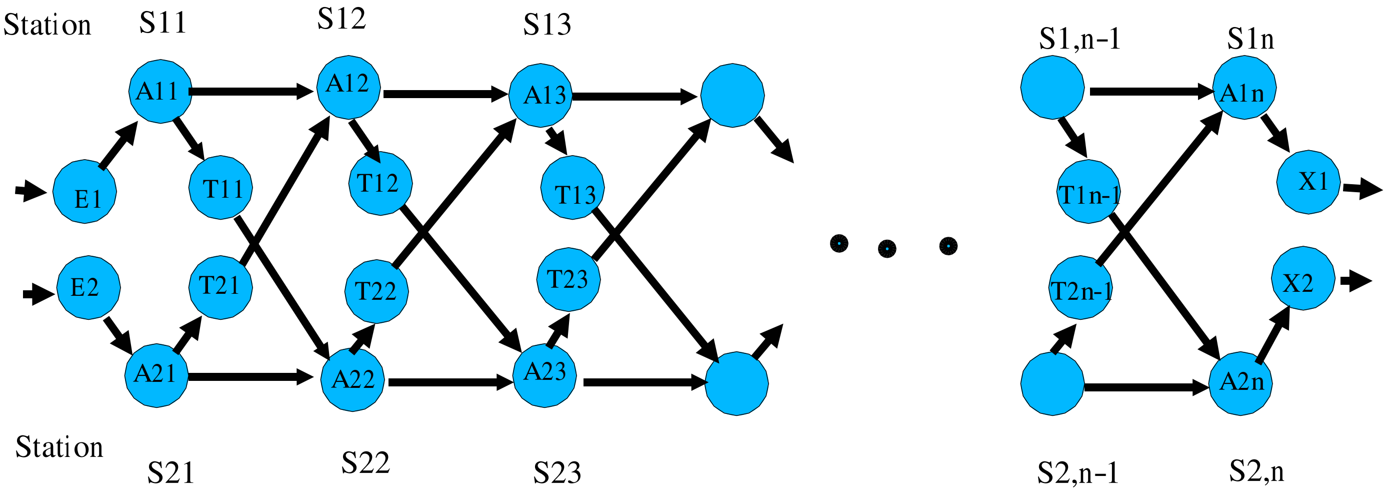
Notation:

$a_{i,j}$: assembly time at station $S_{i,j}$

$t_{i,j}$: transfer time from assembly line i to j after station $S_{i,j}$.

`Reference clrs page 324 Chapter-15`

Notation: Assembly Line Scheduling:



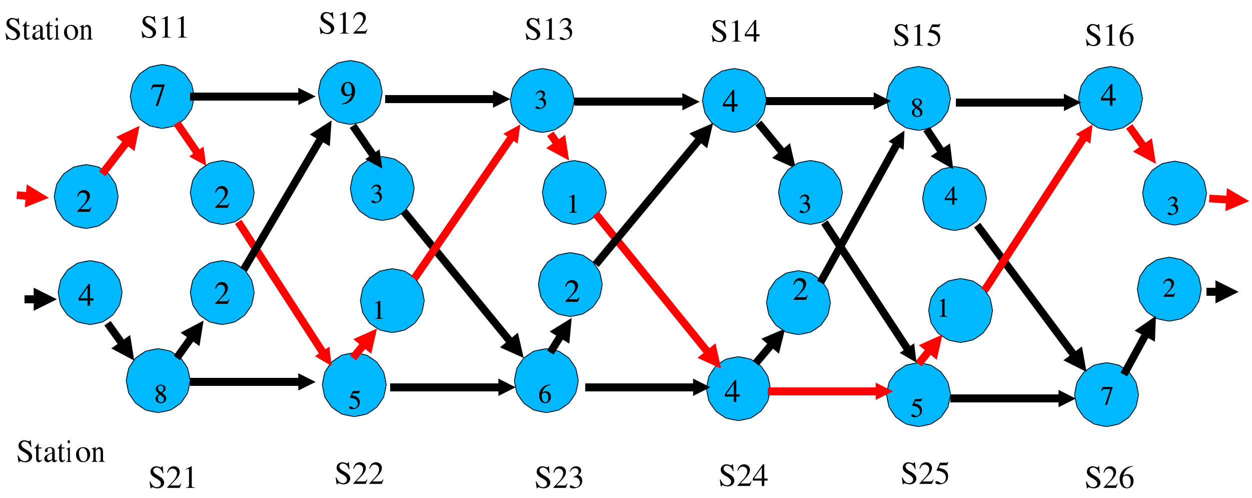
Recurrence Equation: Assembly Line Scheduling:

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{if } j = 1 \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{if } j = 1 \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{if } j \geq 2 \end{cases}$$

`Reference clrs page 328 Chapter-15`

Example: Assembly Line Scheduling:



Approach: Dynamic Programming

Problem: Sequencing the Multiplication of Matrices:

To multiply n matrices

$$A = A_1 \times A_2 \times A_3 \times \ldots \times A_n$$

where each A_i has r_{i-1} rows and r_i columns, the problem is to determine the sequence in which the matrices should be multiplied so that the number of multiplications is minimum over all sequences.

`Note: we are not actually multiplying the matrices, the goal is to determine the order.`

`Reference clrs page 331 Chapter-16`

Approach: Dynamic Programming

Example: Sequencing the Multiplication of Matrices:

To multiply 3 matrices $< A_1, A_2, A_3 >$ of dimensions 10×100, 100×5 and 5×50 if they are multiplied according to parenthesization $(A_1, A_2), A_3)$ there are a total of 7500 multiplications; whereas if they are multiplied according to $(A_1(A_2, A_3))$ there are 75000 multiplications. Thus the multiplication according to first parenthesization is ten times faster.

`Conclusion: The order in which the matrices are multiplied can have a significant effect on the total number of multiplication operations required to find` A

`Reference clrs page 332 Chapter-16`

Approach: Dynamic Programming

Recurrence: Sequencing the Multiplication of Matrices:

Let $m[i,j]$ be the **minimum cost** (number of scalar multiplications) needed to compute the matrix $A_{i \ldots j}$. For the full problem the cost to compute $A_{1 \ldots n}$ would be $m[1,n]$. The recursive equation is :

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} & \text{if } i < j \end{cases}$$

The term $m[i,k]$ is the minimum cost of evaluating
$A_{i \ldots k} = A_i \times A_{i+1} \times \ldots A_k$ and
$m[k+1,j]$ is the cost of evaluating
$A_{k+1 \ldots j} = A_{k+1} \times A_{k+2} \times \ldots A_j$.
The third term is the cost of multiplying these two matrices.

`Reference clrs page 334 Chapter-16`

Approach: Dynamic Programming

Problem: Longest Common Subsequence:
Given two sequences :
$X = < x_1, x_2, \ldots, x_m >$ and
$Y = < y_1, y_2, \ldots, y_n >$
Find a maximum length common subsequence of X and Y.

```
note:  skipping is allowed when finding
common subsequence.  It is not a
"consecutive" subsequence.
```

```
Reference clrs page 350 Chapter-15
```

Approach: Dynamic Programming

Problem: Longest Common Subsequence:

Example: A strand of DNA of one organism may be:
$ACCGGTCGAGTGCGCGGAAGCCGGCCGAA$
and the DNA of another organism may be:
$GTCGTTCGGAATGCCGTTGCTCTGTAA$
and the goal of comparing two strands of DNAs is to determine how "similar" two DNAs are, as a measure of how closely related two organisms are.

`Reference clrs page 350 Chapter-15`

Approach: Dynamic Programming

Problem: Longest Common Subsequence:

Theorem:
Let $X =< x_1, x_2, \ldots, x_m >$ and
$Y =< y_1, y_2, \ldots, y_n >$
be two sequences, and let
$Z =< z_1, z_2, \ldots, z_k >$ be LCS of X and Y then:

1. if $x_m = y_n$ then $z_k = x_m = y_n$ **and** $Z_{k-1} = LCS(X_{m-1}, Y_{n-1})$
2. $x_m \neq y_n$ and $x_m \neq z_k$ **then** $Z = LCS(X_{m-1}, Y)$
3. $x_m \neq y_n$ and $y_n \neq z_k$ **then** $Z = LCS(X, Y_{n-1})$

`Reference clrs page 350 Chapter-15`

Approach: Dynamic Programming

Problem: Longest Common Subsequence:

Observation: To find LCS of X and Y we need to find the LCS of X and Y_{n-1} and also LCS of X_{m-1} and Y. But each of subproblems in turn have sub-subproblems of find LCS of X_{m-1} and Y_{n-1}.

Conclusion: When solutions of subproblems share solution of sub-subproblems don't re-compute them just store them away.

`Reference clrs page 350 Chapter-15`

Approach: Dynamic Programming

Problem: Longest Common Subsequence:

Let $c[i,j]$ be the **length** of LCS of sequences X_i and Y_j then the following recursive formulae hold.

$$c[i,j] = \begin{cases} 0 \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i-1, j], c[i, j-1]) & \text{if } i, j, > 0 \text{ and } x_i \neq y_j \end{cases}$$

`Reference clrs page 350 Chapter-15`

Approach: Dynamic Programming

Problem: KnapSack:
Given an integer K (say the size of the knapsack), and n items of different sizes such that the i^{th} item has an integer size $S[i]$, find a subset of the items whose sizes sum to exactly K, or determine that no such subset exists.

```
Note:  in a 0-1 knap sack either the item is
picked or not picked.  There is no fractional
amount of item which can be picked.

Reference Udi Manber page 108-100 for 0-1 knapsa
Reference clrs page 382 Chapter-15 for fractiona
knapsack
```

Algorithm Knap_Sack(S, K)
Input: S (an array of size n storing the sizes of the items
K (the size of the KnapSack)
Output: P (a two dimensional array such that
$P[i, k].exist = true$ if there exists
a solution to the knapsack problem with
first i items and a knapsack of size k
$P[i, k].belong = true$ if the i^{th}
element belongs to that solution)

Reference Udi Manber page 108-110

Algorithm Knap_Sack(S, K)

begin
  $P[0,0].exist = true$;
  for $k = 1$ to K do
    $P[0,k].exist = false$;
    {comment –there is no need to initialize
    $P[i,0]$ for $i \geq 1$ it will be computed from $P[0,0]$}
  for $i = 1$ to n **do** {comment –for each item}
    for $k = 0$ to K **do** {comment–for each incremental size}
      $P[i,k].exist = false$; {comment –the default value}
        if $P[i-1,k].exist$ **then**
          $P[i,k].belong = false$;
          $P[i,k].exist = true$;
//there is no solution with any selection from fi
        else if $k - S[i] \geq 0$ **then**
          if $P[i-1, k-S[i]].exist$ **then**
            $P[i,k].exist = true$;
            $P[i,k].belong = true$;
end.

Reference Udi Manber page 108-110

Approach: Dynamic Programming

Problem: Optimal Polygon Triangulation: Given a **Convex** polygon $P =< v_0, v_1, \ldots, v_{n-1} >$ and a weight function w defined on triangles formed by sides and chords of P. The Optimal Polygon Triangulation problem is to find a triangulation that minimizes the sum of the weights in the triangulation.

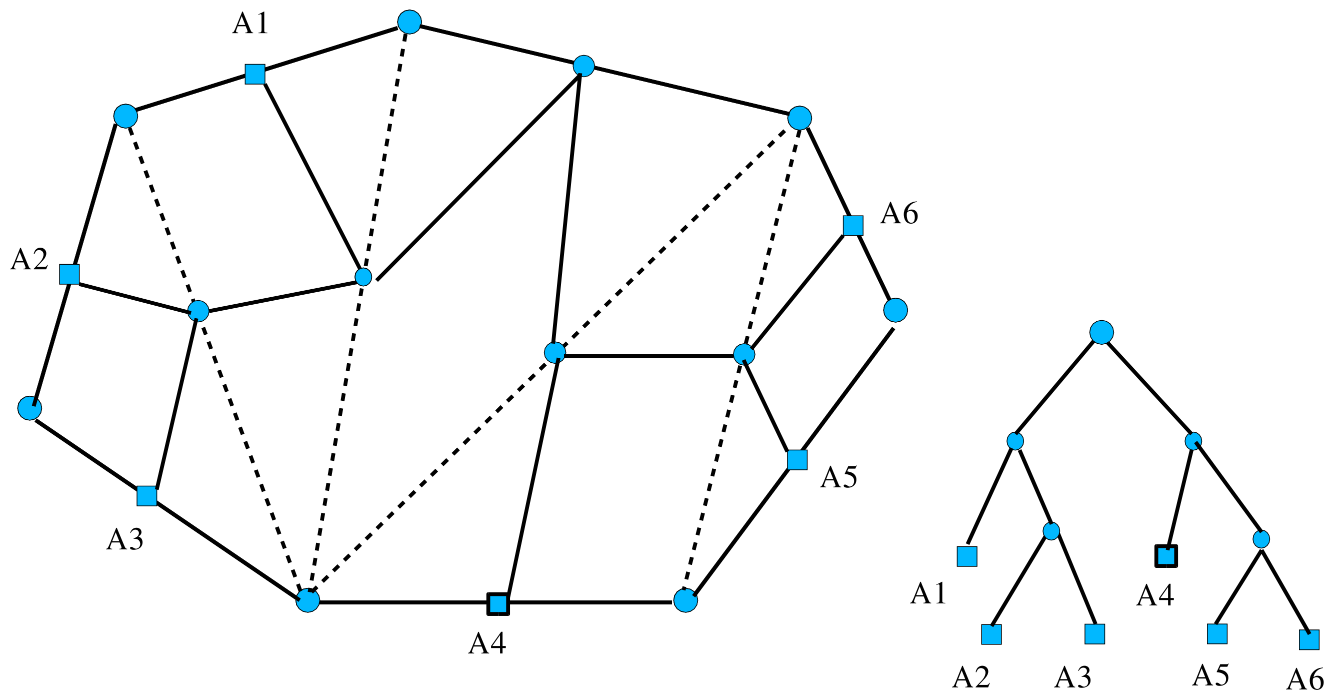
Approach: Dynamic Programming

Problem: Optimal Polygon Triangulation:
One weight function w on triangles is:

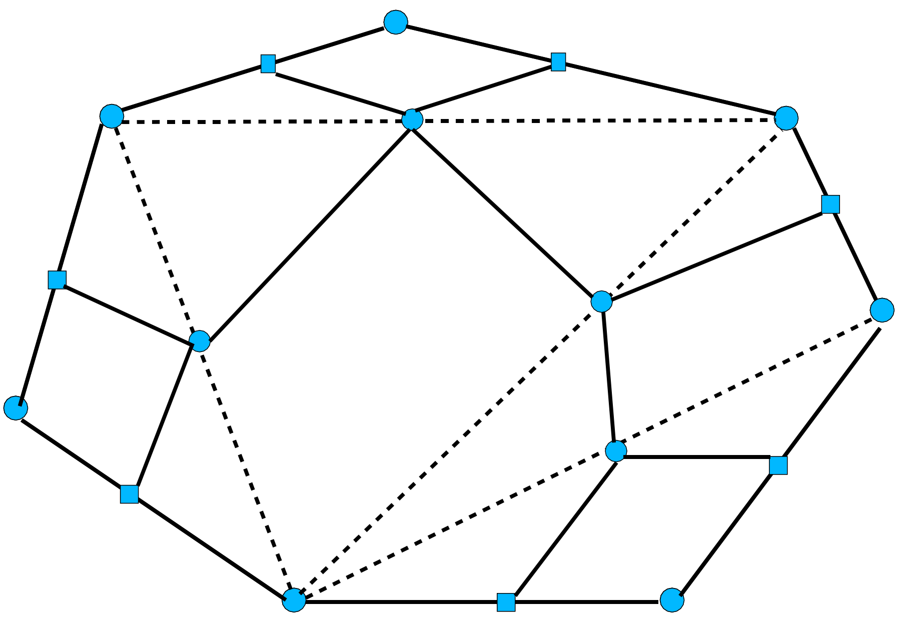
$$w(\triangle v_i v_j v_k) = |v_i v_j| + |v_j v_k| + |v_i v_k|$$

where $|v_i v_j|$ denotes the Euclidean distance from v_i to v_j.

Example: Optimal Triangulation:



(((A1(A2,A3))(A4(A5,A6)))



Approach: Dynamic Programming

Problem: Minimum Edit Distance:
Given two strings
$A = \{a_1, a_2, \ldots a_n\}$, and
$B = \{b_1, b_2, \ldots b_m\}$
and the cost of transformation operations such as copy, insert, delete, and replace, the **edit distance** from A to B is the cost of transformation sequence that transforms A to B.
The 'Minimum Edit Distance' problem is to find the edit distance with the least cost.

`Reference Udi Manber page 155-158`

`Reference clrs page 364 Chapter-15`

Approach: Dynamic Programming

Example: Minimum Edit Distance:
In the example following transformation-operations are used:

- **Copy**:
- **Replace**:
- **Delete**:
- **Insert**:
- **Twiddle**:
- **Kill**:

`Reference Udi Manber page 155-158`

`Reference clrs page 364 Chapter-15`

Approach: Dynamic Programming

Example: Minimum Edit Distance:

A source string `algorithm` is transformed to a target string `altruistic` by following sequence of transformations:

operation	x	z
initial string	algorithm	_
copy	algorithm	a_
copy	algorithm	al_
replace by t	algorithm	alt_
delete	algorithm	alt_
copy	algorithm	altr_
insert u	algorithm	altru_
insert i	algorithm	altrui_
insert s	algorithm	altruis_
twiddle	algorithm	altruisti_
insert c	algorithm	altruistic_
kill	algorithm_	altruistic

The cost of transformation is

$(3.cost(copy)) + cost(replace) + cost(delete) + (4.cost(insert)) + cost(twiddle) + cost(kill)$

Approach: Dynamic Programming

We consider all the different possibilities of constructing the minimum change from A to B with the aid of best changes of smaller sequences involving A and B.

$A(i)$ **denotes the prefix string** $a_1, a_2, \ldots, a_i$ **and**
$B(j)$ **denotes the prefix string** $b_1, b_2, \ldots, b_j$**.**

$C(i, j)$ **denotes the minimum cost of changing** $A(i)$ **to B**(j)**.**

and Denote

$$m[i,j] = \begin{cases} 0 & \textbf{if } a_i = bj \\ 1 & \textbf{if } a_i \neq b_j \end{cases}$$

Three transformations (plus one do nothing) are considered.

delete:
if a_n is deleted in the minimum change from $A(n)$ to $B(m)$, then the best change would be from $A(n-1)$ to $B(m)$ plus one more deletion. That is:

$$C(n, m) = C(n-1, m) + deletion\_cost$$

insert:
if the minimum change from $A(n)$ to $B(m)$ involves insertion of a character to match b_m, then we have

$$C(n, m) = C(n, m-1) + insertion\_cost$$

That is, we find the minimum change from $A(n)$ to $B(m-1)$ and insert a character equal to b_m.

replace:
if a_n is replacing b_m, then we first need to find the minimum change from $A(n-1)$ to $B(m-1)$ and and then to add 1 if $a_n \neq b_m$. That is

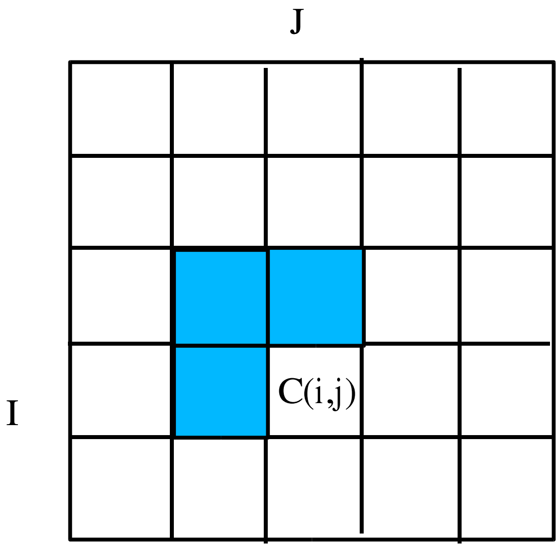
$$C(n, m) = C(n-1, m-1) + replacement\_cost$$

do nothing:
if $a_n = b_m$, then $C(n, m) = C(n-1, m-1)$

In short (assuming insertion cost, deletion cost, replacement cost is equal to 1),

$$C(n,m) = \min \begin{cases} C(n-1,m) + cost\_of\_deletion & \text{deleting } a_n \\ C(n,m-1) + cost\_of\_insertion & \text{inserting for} \\ C(n-1,m-1) + cost\_of\_replacement & \text{replacing } a_n \\ C(n-1,m-1) & \text{do\_nothing } a \end{cases}$$

The dependencies of C(i,j)



Approach: Dynamic Programming

Algorithm Minimum_Edit_Distance(A, n, B, m)
Input: A (a string of size n; and
B (a string of size m)
Output: C (the cost matrix)
begin

```
    for i = 0 to n do C[i, 0] = i;
    for j = 1 to m do C[0, j] = j;
    for i = 1 to n do
        for j = 1 to m do
            x = C[i-1, j] + cost_of_deletion;
            y = C[i, j-1] + cost_of_insertion;
            if a_i = b_j then
                z = C[i-1, j-1];
            else
                z = C[i-1, j-1] + cost_of_replacement;
            C[i, j] = min(x, y, z);
```

end.

Reference Udi Manber page 158