Dynamic Programming

chandakmb@gmail.com, hodcs@rknec.edu
www.mbchandak.com

Topics:

Longest Common Subsequence

Optimal Binary Search Tree

0/1 Knapsack problem

Chained Matrix Multiplication

Travelling Salesman Problem

String Editing

Characteristics

- **Developed by Richard Bellman in 1950.**
- **To provide accurate solution with the help of series of decisions.**
- **Define a small part of problem and find out the optimal solution to small part.**
- **Enlarge the small part to next version and again find optimal solution.**
- **Continue till problem is solved.**
- **Find the complete solution by collecting the solution of optimal problems in bottom up manner.**

Characteristics

- Types of problems and solution strategies.
- **1. Problem can be solved in stages.**
- **2. Each problem has number of states**
- **3. The decision at a stage updates the state at the stage into the state for next stage.**
- **4. Given the current state, the optimal decision for the remaining stages is independent of decisions made in previous states.**
- **5. There is recursive relationship between the value of decision at a stage and the value of optimum decisions at previous stages.**

Characteristics

- How memory requirement is reduce:
- How recursion overheads are converted into advantage
- - By storing the results of previous n-2 computations in computation of n-1 stage and will be used for computation of “n” stage.
- Not very fast, but very accurate
- It belongs to smart recursion class, in which recursion results and decisions are used for next level computation. Hence not only results but decisions are also generated.
- **Final collection of results: Bottom Up Approach.**

Longest Common Subsequence

- Given two strings of characters, LCS is a method to find a longest subsequence either continuous or non-continuous and is common in both the strings. The subsequence generation starts from comparison of first character of both the given strings.
- *The two strings may or may not be of equal length.*
- Let “A” and “B” be two strings of length “m” and “n” respectively, and let “i” and “j” be two pointers to handle the two strings.
- Let matrix c[m,n] be the storage matrix to store the results of comparison related with two strings. The C[m,n] matrix will be handled using pointer “i” and “j”.

If i=0 and j=0, then c[i,j] = 0

If a[1..i] and b[1..j] is compared and a[i] is not equal to b[j]. Then the comparison will be either

In between

a[1..i-1] and b[1..j] or a[1..i] and b[1..j-1]
Then

c[i,j] = max[c[i-1,j], c[I,j-1]]

If a[1..i] and b[1..j] is compared and a[i] is equal to b[j].

Then c[i,j] = c[i-1,j-1] + 1

Example
STRING A = G D V E G T A B = G V C E K S T

i/j	0	1G	2V	3C	4E	5K	6S	7T
0	0/H	0/H	0/H	0/H	0/H	0/H	0/H	0/H
1G	0/H	**1/D**	1/S	1/S	1/S	1/S	1/S	1/S
2D	0/H	1/U	1/U	1/U	1/U	1/U	1/U	1/U
3V	0/H	1/U	**2/D**	2/S	2/S	2/S	2/S	2/S
4E	0/H	1/U	2/U	2/U	**3/D**	3/S	3/S	3/S
5G	0/H	1/D	2/U	2/U	3/U	3/U	3/U	3/U
6T	0/H	1/U	2/U	2/U	3/U	3/U	3/U	**4/D**
7A	0/H	1/U	2/U	2/U	3/U	3/U	3/U	**4/U**

Example:2

STRING A = X M J Y A U Z B = M Z J A W X U

i/j	0	1X	2M	3J	4Y	5A	6U	7Z
0	0/H	0/H	0/H	0/H	0/H	0/H	0/H	0/H
1M	0/H	1/U	1/D	1/S	1/S	1/S	1/S	1/S
2Z	0/H	1/U	1/U	1/U	1/U	1/U	1/U	2/D
3J	0/H	1/U	1/U	2/D	2/S	2/S	2/S	2/S
4A	0/H	1/U	1/U	2/U	2/U	3/D	3/S	3/S
5W	0/H	1/U	1/U	2/U	2/U	3/U	3/U	3/U
6X	0/H	1/D	1/U	2/U	2/U	3/U	3/U	3/U
7U	0/H	1/U	1/U	2/U	2/U	3/U	4/D	4/S

Algorithm:

- Assumptions: Let the two strings be present in array “a” size “m” and array “b” size “n” handled using pointers “i“ and “j”.
- The resultant array will be c[m,n] : one instance c[i,j]. The array “c” will be structure: c[i,j].val and **c[i,j].dir = “u”, “s”, “d” and “h”**

```
Algorithm lcs (a,b: c)
{
   for i = 0 to m do
      for j = 0 to n do
        if a[i]=0 or b[j]=0
{
      c[i,j].val = 0;
      c[i,j].dir=‘h’
}
else
{
   if (a[i] ≠ b[j])
      c[i,j].val = max [c[i-1, j].val, c[i,j-1].val]
       if(c[i-1, j].val >= c[i,j-1].val)
         c[i,j].dir = ‘u’
       else
         c[i,j].dir=‘s’
   else
      c[i,j].val = c[i-1,j-1].val  + 1
      c[i,j].dir = ‘d’
}
```

Algorithm:

- Printing LCS: Assumptions: Let “A” be the given array.
- Let “C” be the array containing the details of LCS

```
Algorithm print_lcs (a,c,i,j)
{
   if (i=0 or j=0)
      return;
  if(c[i,j] = ‘d’
  {
    print_lcs(a,c,i-1,j-1)
    print(a[i])
  }
  else
  {
     if(c[i,j] = ‘u’
     print_lcs(a,c,i-1,j)
    else
     print_lcs(a,c,i,j-1)
  }
```

Optimal Binary Search Tree (OBST)

- Drawback of BST: The first value in the sequence is always used as ROOT node.
- If the data is present in sorted order, BST will be in the form of linked list. This will increase search time and complexity equation will be not valid.
- For example if dictionary word are to be stored in BST, it will be in the form of linked list:

Apple, Bat, Cat, Dog, Elephant, Fish, Gun, Horse, Ink, Jug, King, Lion

Since all the start characters are Greater than "A", all the values will be on the Right side of the Apple.

Example of word list

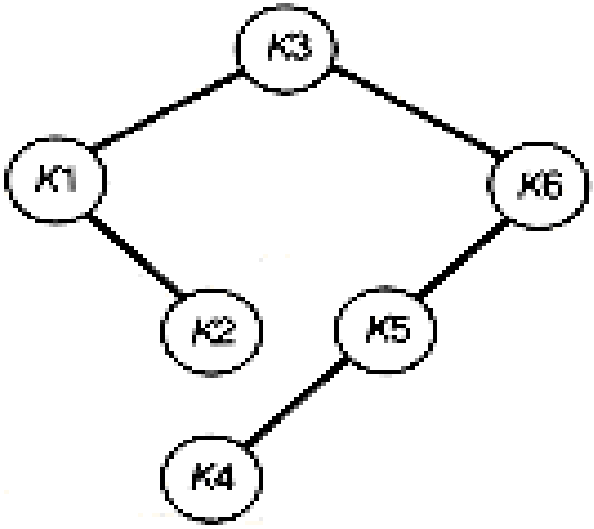
Optimal Binary Search Tree

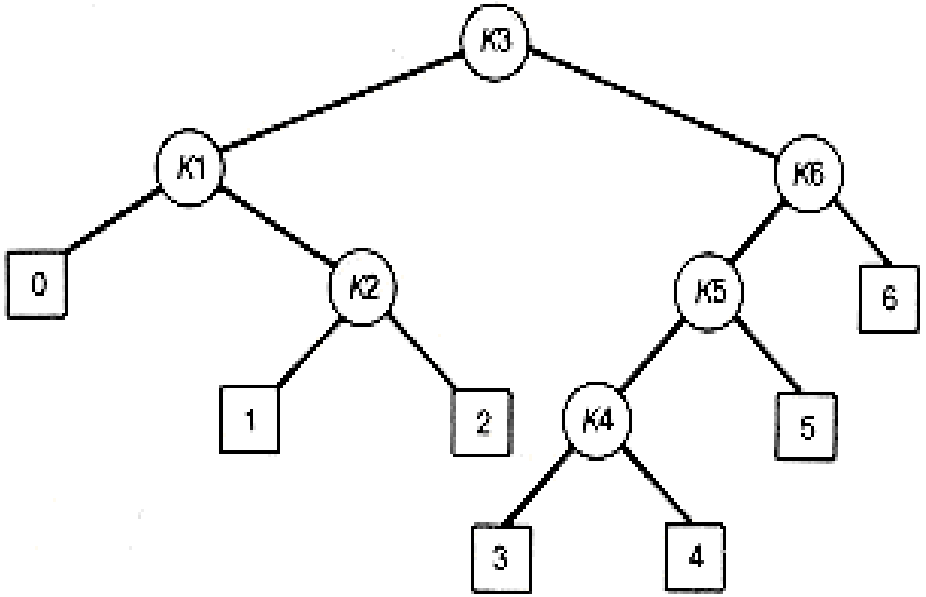
- **Need:** Mechanism to decide out of the available values, which value should be used as ROOT, so that the tree is balanced and COST of searching is minimum.
- **Requirement**: For selection, the probability of data in the domain text is required.
- **There are two values**: Successful probability value and Unsuccessful probability value.
- **Application**: OBST is applied to generate the tree for the set of values (keys) with defined successful and unsuccessful search probabilities.

Methodology

- Requirements:
- Root node selection
- Node for left sub-tree and right sub-tree
- Final cost of tree should be minimum
- **Method to derive the COST of Tree**
- **For the present discussion: an arbitrary node is selected as root and tree is constructed.**
- **The cost of tree is calculated by:**
  - **Level of node.**
  - **Probability of success.**
  - **Probability of un-success.**

Optimal Binary Search Tree





Optimal Binary Search Tree

i	0	1	2	3	4
pi		0.12	0.10	0.09	0.20
qi	0.10	0.08	0.05	0.11	0.15
sum	0.10	0.20	0.15	0.20	0.35

$$w[i,j] = \begin{Bmatrix} q_{i-1}\ if\ j = i-1 \\ w[i,j-1] + p_j + q_j \end{Bmatrix}$$

$$e[i,j] = \begin{Bmatrix} q_{i-1}\ if\ j = i-1 \\ min[i \le r \le j]\ e[i,r-1] + e[r+1,j] + w[i,j] \end{Bmatrix}$$

W	**0**	**1**	**2**	**3**	**4**
1	0.10	0.30	0.45	0.65	1.0
2	#	0.08	0.23	0.43	0.78
3	#	#	0.05	0.25	0.60
4	#	#	#	0.11	0.46
5	#	#	#	#	0.15

E	**0**	**1**	**2**	**3**	**4**
1	0.10	0.48/1	0.91/1	1.54/2	2.63/3
2	#	0.08	0.36/2	0.90/3	1.83/4
3	#	#	0.05	0.41/3	1.16/4
4	#	#	#	0.11	0.72/4
5	#	#	#	#	0.15

i	0	1	2	3	4
pi		0.12	0.10	0.09	0.20
qi	0.10	0.08	0.05	0.11	0.15
sum	0.10	0.20	0.15	0.20	0.35

Example 2: OBST

i	0	1	2	3	4	5
pi		0.05	0.20	0.05	0.10	0.15
qi	0.10	0.10	0.05	0.05	0.05	0.10
sum	0.10	0.15	0.25	0.10	0.15	0.25

$$w[i,j] = \begin{Bmatrix} q_{i-1}\ if\ j = i-1 \\ w[i,j-1] + p_j + q_j \end{Bmatrix}$$

$$e[i,j] = \begin{Bmatrix} q_{i-1}\ if\ j = i-1 \\ min[i \le r \le j]\ e[i,r-1] + e[r+1,j] + w[i,j] \end{Bmatrix}$$

Travelling Salesman Problem: (Self Study topic)

- Principle: To find out cycle in a given graph by visiting all the vertices once and reaching back to source vertex.
- Constraint: A vertex or an edge can be only visited only once.
- The total length of cycle should be minimum.
- Applied on complete graph (a matrix with only diagonal element as zero).

Formulation

- Formula used:
- g(i,S) = min {Cij + g(j – ***(S - {j})***)
- j€S
- ***In this formula:***
- “i" represents source vertex
- “j” represents the next vertex used as intermediate.
- “S” represents the vertices other than “i" and if out of these vertices “j” vertex is used as next intermediate, then remaining vertices left will be ***S - {j}.***

Example

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

1			*	1
1		**		1
1	***			1
1				1

Example

- The path will consist of FIVE vertices: in which FIRST and LAST vertex will be SAME (SOURCE)

- Step: 1: To perform computation with zero intermediates an reach to Source, i.e., for position one place before the SOURCE

- Possible vertices which can be used are 2, 3, 4
- g(2, *) = C21 = 5
- g(3,*) = C31 = 6
- g(4,*) = C41 = 8

Example:

- **Step 2:**
- Vertex 2 at pre-to-pre-final position

g(2, {3}) = C23 + g(3,*) = 9+6=15

g(2, {4}) = C24 + g(4,*) = 10 + 8 = 18

- Vertex 3 at pre-to-pre-final position

g(3, {2}} = {C32 + g(3, *) } = 13 + 5 = 18

g(3, {4}} = {C34 + g(4, *) } = 12 + 8 = 20

- Vertex 4 at pre-to-pre-final position

g(4, {2}} = {C42 + g(2, *) } = 8 + 5 = 13

g(4, {3}} = {C43 + g(3, *) } = 9 + 6 = 15

Example:

- **Step 3: This step will allow two intermediates**

g (2, {3,4}) = $\min\{C23 + g\{3,4\}\ ,\ C24 + g\{4,3\}) = 25$

g(3, {2,4}) = $\min\{C32 + g\{2,4\}\ ,\ C34 + g\{4,2\}) = 25$

g (4, {2,3}) = $\min\{C42 + g\{2,3\}\ ,\ C43 + g\{3,2\})$ = 23

Step 4: The path from the source vertex

g (1, {2,3,4} = min{C12 + g{2,{3,4}}, C13 + g{3, {2,4}}, C14 + g{4, {2,3}}}

= min {10+25, 15+25, 30+23} = 35

Final Path:

1	2	4	3	1

String Editing: Self Study topic

- **Principle: Dynamic programming based approach to transform one string into another.**
- **The editing is performed using sequence of edit operations.**
- **Three Edit operations permitted: INSERT, DELETE, UPDATE**
- **With each operation cost is associated: INSERT=1, DELETE=1, UPDATE=2**
- **The cost of transformation is cost of all operations. It is required to convert string : A into string : B in minimum cost.**

String Editing

- **Let X be the string to be modified, so there will be no change in the String Y.**
- **Let D(xi) be cost of deleting symbol xi from X**
- **Let I(yj) be cost of inserting symbol yj into string X**
- **Let C(xi, yj) be cost of changing symbol xi of X to yj**
- **Example: X = a a b a b and Y = b a b b**
- **One simple solution is delete all symbols of X and insert all symbols of Y. Total Cost = 5 + 4 = 9**
- **Cost Effective solution: Delete x1, x2 and insert y4, total cost = 2+1 = 3**

String Editing: Implementation

- Formulation: Let “i“ and “j” be two pointers, it is required to generate **cost matrix.**
- Case 1: If i=0 and j=0 → cost[i,j] = 0
- Case 2: if i=0 and j>0 → cost[0,j] = cost[0,j-1] + Ins(yj)
- Case 3: if i>0 and j=0 → cost[i,0] = cost[i-1, 0] + Del(xi)
- Case 4: i>0 and j>0 → There are three possibilities

String Editing: Implementation

- Case 4.1: Transform x1, x2, x3, xi-1 into y1, y2, y3, yj and then delete xi. The corresponding cost:

$$Cost[i,j]=Cost(i-1, j) + D(xi)$$

- Case 4.2: Transform x1, x2, x3, xi into y1, y2,y3,yj-1 using minimum edit sequence and then insert yj.
  - $Cost[i,j]=Cost(i, j-1) + I(yj)$
- Case 4.3: Transform x1, x2, x3, xi-1 into y1, y2, y3, yj-1 and then change symbol xi to yj the cost associated will be
  - $Cost[i,j]=Cost(i-1, j-1) + U(xi,yj)$

Recurrence Equation of String Editing

- cost (i,j) = $\left\{\begin{array}{c} 0\ if\ i=j=0 \\ cost(i-1,0)+D(xi)\ if\ i>0, j=0 \\ cost(0,j-1)+I(yi)\ if\ i=0, j>0 \\ cost'(i,j)\ if\ i>0, j>0 \end{array}\right\}$

- $cost'(i,j) = min\left\{\begin{array}{c} cost(i-1,j)+D(xi) \\ cost(i,j-1)+I(xi) \\ cost(i-1,j-1)+U(xi,yj) \end{array}\right\}$

String Editing Example:

- X = a a b a b (m=5)
- Y = b a b b (n=4)
- Compulsory: X on Row side and Y on Column side
- Cost matrix dimensions: cost[0..m, 0..n]

	0	1 (b)	2 (a)	3 (b)	4 (b)
0	0	1	2	3	4
1 (a)	1	2/IDC	1/C	2/I	3/I
2 (a)	2	3/IDC	2/DC	3/IDC	4/IDC
3 (b)	3	2/C	3/ID	2/C	3/IC
4 (a)	4	3/D	2/C	3/ID	4/IDC
5 (b)	5				

Example: String Editing

	0	1 (b)	2 (a)	3 (b)	4 (b)
0	0	1(I)	2(I)	3(I)	4(I)
1 (a)	1(D)	2(I/D)	1(C)	2(I)	3(I)
2 (a)	2(D)	3(I/D/U)	2(C)	3(I/D/C)	4(I/D/C)
3 (b)	3(D)	2(C)	3(I/D)	2(C)	3(I/C)
4 (a)	4(D)	3(D)	2(C)	3(I/D)	4(I/D/C)
5 (b)	5(D)	4(D/C)	3(D)	2(C)	3(C)

Example: 2

- X = a a b a a b a b a a
- Y = b a b a a b a b