Huffman Coding

Each character is represented in 8 bits when characters are coded using standard codes such as ASCII. It can be seen that the characters coded using standard codes have fixed-length code word

representation. In this fixed-length coding system the total code length is more. For example, let we have six characters (a, b, c, d, e, f) and their frequency of occurrence in a message is {45, 13, 12, 16, 9, 5}. In fixed-length coding system we can use three characters to represent each code. Then the total code length of the message is (45+13+12+16+9+5) x 3 = 100 x 3 = 300.

Let us encode the characters with variable-length coding system. In this coding system, the character with higher frequency of occurrence is assigned fewer bits for representation while the characters having lower frequency of occurrence in assigned more bits for representation. The variable length code for the characters are shown in the following tableThe total code length in variable length coding system is $1 \times 45 + 3 \times 12 + 3 \times 16 \times 4 \times 9 + 4 \times 5 = 224$. Hence fixed length code requires 300 bits while variable code requires only 224 bits.

a	b	c	d	e	f
0	101	100	111	1101	1100

Prefix (Free) Codes

We have seen that using variable-length code word we minimize the overall encoded string length. But the question arises whether we can decode the string. If *a* is encoded 1 instead of 0 then the encoded string "111" can be decoded as "d" or "aaa". It can be seen that we get ambiguous string. The key point to remove this ambiguity is to use prefix codes. Prefix codes is the code in which there is no codeword that is a prefix of other codeword.

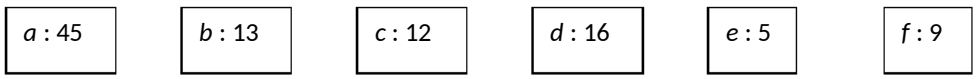
The representation of "decoding process" is binary tree whose leaves are characters. We interpret the binary codeword for a character as path from the root to that character, where

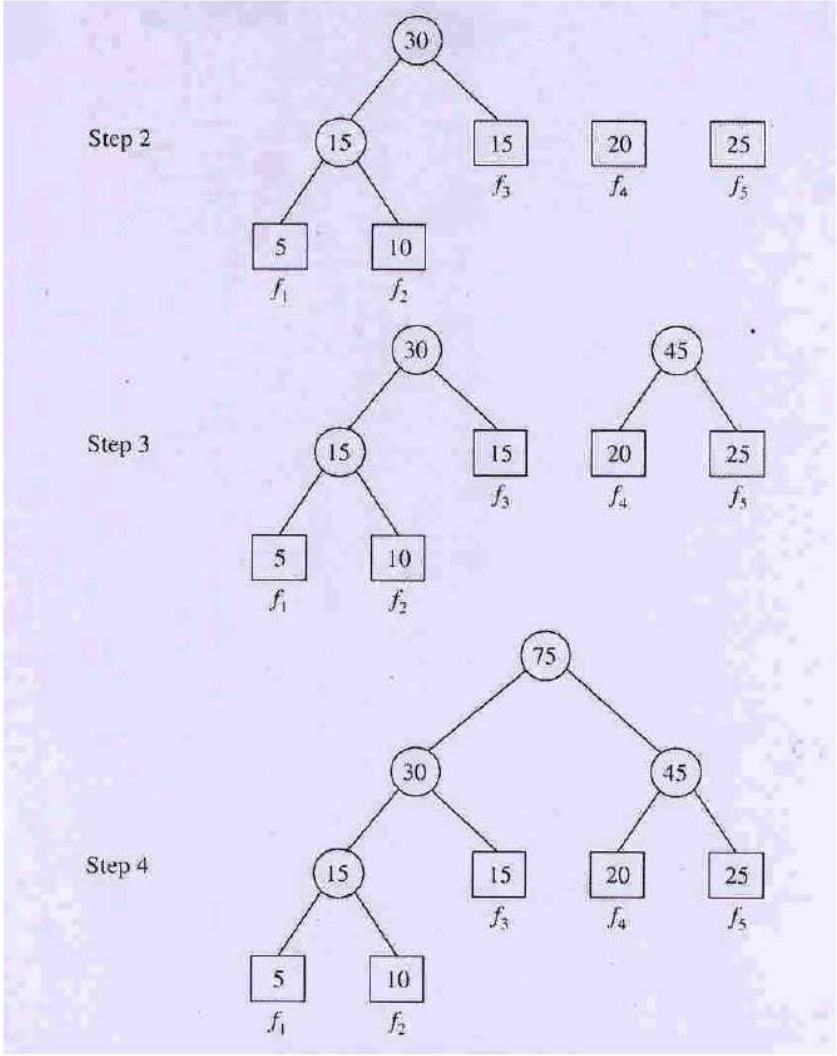
⇒ "0" means "go to the left child"
⇒ "1" means "go to the right child"

Greedy Algorithm for Huffman Code:

According to Huffman algorithm, a bottom up tree is built starting from the leaves. Initially, there are *n* singleton trees in the forest, as each tree is a leaf. The greedy strategy first finds two trees having minimum frequency of occurrences. Then these two trees are merged in a single tree where the frequency of this tree is the total sum of two merged trees. The whole process is repeated until there in only one tree in the forest.

Let us consider a set of characters *S*=<*a*, *b*, *c*, *d*, *e*, *f*> with the following frequency of occurrences *P* = < 45, 13, 12, 16, 5, 9 >. Initially, these six characters with their frequencies are considered six singleton trees in the forest. The step wise merging these trees to a single tree is shown in Fig. 6.3. The merging is done by selecting two trees with minimum frequencies till there is only one tree in the forest.





Step wise merging of the singleton trees.

Now the left branch is assigned a code “0” and right branch is assigned a code “1”. The decode tree after assigning the codes to the branches.

The binary codeword for a character is interpreted as path from the root to that character; Hence, the codes for the characters are as follows

a = 0

b = 101

c = 100

d = 111

e = 1100

f = 1101

Therefore, it is seen that no code is the prefix of other code. Suppose we have a code 01111001101. To decode the binary codeword for a character, we traverse the tree. The first character is 0 and the character at which the tree traversal terminates is *a*. Then, the next bit is 1 for which the tree is traversed right. Since it has not reached at the leaf node, the tree is next traversed right for the next bit 1. Similarly, the tree is traversed for all the bits of the code string. When the tree traversal terminates at a leaf node, the tree traversal again starts from the root for the next bit of the code string. The character string after decoding is "*adcf*".

AlgorithmHUFFMAN(*n*, *S*)

{

// *n* is the number of symbols and *S* in the set of characters, for each character *c*∈ *S*, the frequency of occurrence in *f*(*c*) //

Initialize the priority queue;

Q = *S* ; // Inilialize the priority Q with the frequencies of all the characters of set S//

for(*i* =1 ; *i*<= *n*-*i*, *i*++){

z = CREAT _NODE ( ); // create a node pointed by z; //

// Delete the character with minimum frequency from the Q and store in node *x*//

x = DELETE _MIN (Q);

// Delete the character with next minimum frequency from the Q and store in node *y*//

y = DELETE_MIN (Q);

z→*lef* = *x*; // Place x as the left child of z//

z→*right* = *y*; // Place y as the right child of z//

//The value of node *z* is the sum of values at node x and node y//

f(*z*) = *f*(*x*) + *f*(*y*);

//insert z into the priority Q//

INSERT (Q, *z*);

}

returnDELETE_MIN(Q)

}

Algorithm of Huffman coding.

Activity Selection Problem

Suppose we have a set of activities S = {a_1, a_2 .....a_n} that with to use a common resource. The objective is to schedule the activities in such a way that maximum number of activities can be performed. Each activity a_i has a start times$_i$ and a finish time f, where $0 \leq S_i < f_i < \infty$. The activities a_i and a_j are said to compatible if the intervals [s$_i$, f_i] and [S_j, f_j] do not overlap that means $s_i \geq f_j$ *or* $s_j \geq f_i$.

For example, let us consider the following set S of activities, which are sorted in monotonically increasing order of finish time.

i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	72	13	14

For this example, the subsets {a_3, a_9, a_{11}}, {a_1, a_4, a_8, a_{11}} and {a_2, a_4, a_9, a_{11}} consist of mutually compatible activities. We have two largest subsets of mutually compatible activities.

Now, we can devise greedy algorithm that works in a top down fashion. We assume that the n input activities are ordered by monotonically increasing finish time or it can be sorted into this order in O($nlog_2n$) time. The greedy algorithm for activity selection problem is given below.

AlgorithmACTIVITY SELECTION (S, f)

{

n = LENGTH (S) ; // n is the total number of activities //

A = {a_1} ; // A is the set of selected activities and initialized to a_1)//

i = 1; // i represents the recently selected activity //

for (j = 2 ; j< = n ; j++)

{

if ($s_j \geq f_i$) {

$A = A \cup \{a_m\}$;

$i = j$;

}

```
    }
    return A ;
}
```

Algorithm 3. Algorithm of activity selection problem.

The algorithm takes the start and finish times of the activities, represented as arrays *s* and *f* , length (s) gives the total number of activities. The set A stores the selected activities.Since the activities are ordered with respect to their finish times the set A is initialized to contain just the first activity a_1. The variable *i* stores the index of the recently selected activity. The for loop considers each activitya_j and adds to the set A if it is mutually compatible with the previously selected activities. To see whether activity a_j is compatible with every activity assumingly in A, it needs to check whether the short time of a_j is greater or equal to the finish time of the recently selected activity a_i.

Minimum Cost Spanning Tree

Let *G* = (*V*, *E*) be the graph where V is the set of vertices, E is the set of edges and |V|= *n*. Thespanning tree *G*′= (*V*, *E*′) is a sub graph of *G* in which all the vertices of graph *G* are connected with minimum number of edges. The minimum number of edges required to correct all the vertices of a graph *G* in *n* − 1. Spanning tree plays a very important role in designing efficient algorithms.

Let us consider a graph shown in Fig 6.6(a). There are a number of possible spanning trees that is shown in Fig 6.6(b).

If we consider a weighted graph then all the spanning trees generated from the graph have different weights. The weight of the spanning tree is the sum of its edges weights. The spanning tree with minimum weight is called minimum spanning tree (MST). Fig. 6.7 shows a weighted graph and the minimum spanning tree.

A greedy method to obtain the minimum spanning tree would construct the tree edge by edge, where each edge is chosen accounting to some optimization criterion. An obvious criterion would be to choose an edge which adds a minimum weight to the total weight of the edges selected so far. There are two ways in which this criterion can be achieved.

1. The set of edges selected so far always forms a tree, the next edge to be added is such that not only it adds a minimum weight, but also forms a tree with the previous edges; it can be shown that the algorithm results in a minimum cost tree; this algorithm is called Prim's algorithm.
2. The edges are considered in non decreasing order of weight; the set T of edges at each stage is such that it is possible to complete T into a tree; thus T may not be a tree at all stages of the algorithm; this also results in a minimum cost tree; this algorithm is called Krusleat's algorithm.

Prim's Algorithm

This algorithm starts with a tree that has only one edge, the minimum weight edge. The edges (j, q) is added one by one such that node j is already included, node q is not included and weight $wt(j, q)$ is the minimum amongst all the edges (x, y) for which x is in the tree and $y$is not. In order to execute this algorithm efficiently, we have a node index $near(j)$ associated with each node j that is not yet included in the tree. If a node is included in the tree, $near(j) = 0$. The node $near(j)$ is selected into the tree such that $wt(j, near(j))$ in the minimum amongst all possible choices for $near(j)$.

AlgorithmPRIM (E, wt, n, T)

//E is the set of edges, $wt(n, n)$ is the weight adjacency matrix for G, n is the number of nodes and $T(n-1, 2)$ stores the spanning tree.

```
{
    (k, l) = edge with minimum wt.
    minwt = wt[k, l] ;
    T[1, 1] = k, T[1, 2] = l ;
    for(i = 1; i<=n; i++){
        if(wt[i, k] <wt [i, l] )
            near[i] = k;
        else
            near[i] = l;
    }
    near[k] = near[l] = 0;
    for(i = 2; i≤=n-1 ; i++)
    {
        letj be an index such that near[j] ≠ 0 and wt[j, near[j]] is minimum.
        T[i, 1] = j; T[i, 2] = near[j];
        minwt = minwt + wt[j, near[j]] ;
        near[j] = 0 ;
        for(k = 1; k<= n ; k++){
            if (near[k] ≠ 0 and wt[k, near[k]] >wt[k, j])
                near[k] = j;
        }
```

}

if(*minwt* == ∞)

print("No spanning tree");

return*minwt*;

}

Algorithm 4. Prim's algorithm for finding MST.

Fig 6. The weighted undirected graph to illustrate Prim's algorithm

Let us consider the weighted undirected graph shown in Fig.6.8 and the objective is to construct a minimum spanning tree. The step wise operation of Prim's algorithm is described as follows.

Step 1 The minimum weight edge is (2, 3) with weight 5. Hence, the edge (2, 3) is added to the tree. *near*(2) and *near*(3) are set 0.

Step 2 Find near of all the nodes that are not yet selected into the tree and its cost.

near(1) = 2 *weight* = 16

near(4) = 2 *weight* = 6

near (5) = - *weight* = ∞

near (6) = 2 *weight* = 11

The node 4 is selected and the edge (2, 4) is added to the tree because

weight(4, *near*(4)) is minimum. Then near(4) is set 0.

Step 3 *near*(1) = 2 *weight* = 16

near(5) = 4 *weight* = 18

near(6) = 2 *weight* = 11

As *weight*(6, *near*(6)) is minimum, the node 6 is selected and edge (2, 6) is added to the tree. So *near*(6) is set 0

Step 4 near (1) = 2 weight = 16

near (5) = 4 weight = 18

Next, the edge (2, 1) is added to the tree as *weight*(1, near(1)) is minimum. So *near*(1) is set 0.

Step 5 near (5) = 1 weight 12

The edge (1, 5) is added to the tree. The Fig. 6.9(a) to 6.9(e) show the step wise construction of MST by Prim's algorithm.

Fig. 6.9 Step wise construction of MST by Prim's algorithm

Time complexity of Prim's Algorithm

Prim's algorithm has three for loops. The first *for* loop finds the near of all nodes which require O(n) time. The second *for* loop is to find the remaining n-2 edges and the third *for* loop updates near of each node after adding a vertex to MST. Since the third for loop is within the second for loop, it requires O(n^2) time. Hence, the overall time complexity of Prim's algorithm is O(n^2).

Kruskal's Algorithm

This algorithm starts with a list of edges sorted in non decreasing order of weights. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle. Initially, each vertex is in its own tree in the forest. Then, the algorithm considers each edge ordered by increasing weights. If the edge (u, v) connects two different trees, then (u, v) is added to the set of edges of the MST and two trees connected by an edge (u, v) are merged in to a single tree. If an edge (u, v) connects two vertices in the same tree, then edge (u, v) is discarded.

The Krushal's algorithm for finding the MST is presented as follows. It starts with an empty set A, and selects at every stage the shortest edge that has not been chosen or rejected regardless of where this edge is situated in the graph. The pseudo code of Kruskal's algorithm is given in Algorithm .

The operations an disjoint sets used for Krushal's algorithm is as follows:

Make_set(v) : create a new set whose only member is pointed to v.

Find_set(v) : returns a pointer to the set containing v.

Union(u, v) : unites the dynamic sets that contain u and v into a new set that is union of these two sets.

Algorithm KRUSKAL (V, E, W)

// V is the set of vertices, E is the set of edges and W is the adjacency matrix to store the weights of the links. //

{

A = Φ ;

for (each vertex *u* in *V*)

Make_set(*u*)

Create a min heap from the weights of the links using procedure *heapify*.

for (each least weight edge (*u*, *v*) in *E*) // least weight edge is the root of the heap//

if (*Find_set*(*u*) ≠*Find_set*(*v*)){ // *u* and *v* are in two different sets //

A = A∪{*u*, *v*}

Union(*u*, *v*)

}

}

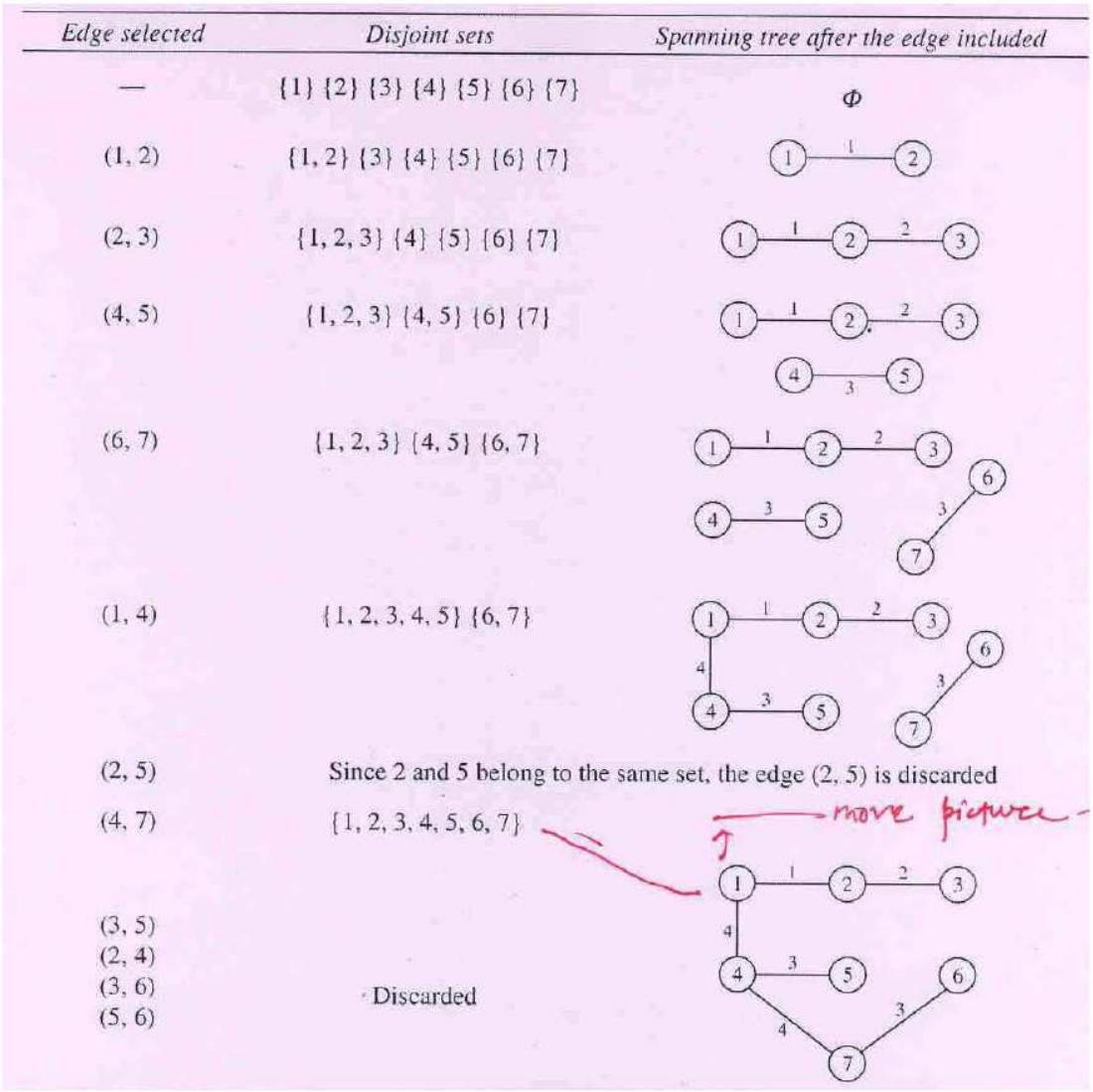
return A ;

}

Algorithm. 5 Kruskal's algorithm for finding MST.

Let us consider the graph shown in Fig.6.10 to illustrate Kruskal's algorithm.

The step wise procedure to construct MST by following the procedure presented given below.



Step wise construction of MST by Kruskal's algorithm

Time complexity of Kruskal's Algorithm

The Kruskal's algorithm first creates n trees from n vertices which is done in O(n) time. Then, a heap is created in O(n) time using heapify procedure. The least weight edge is at the root of the heap. Hence, the edges are deleted one by one from the heap and either added to the MST or discarded if it forms a cycle. This deletion process requires O($nlog_2n$). Hence, the time complexity of Kruskal's algorithm is O($nlog_2n$).

Shortest Path Problem

Let us consider a number of cities connected with roads and a traveler wants to travel form his home city A to the destination B with a minimum cost. So the traveler will be interested to know the following:

- Is there a path from city A to city Bff
- If there is more than one path from A to B, which is the shortest or least cost pathff

Let us consider the graph $G = (V, E)$, a weighting function $w(e)$ for the edges in E and a source node v_0. The problem is to determine the shortest path from v_0 to all the remaining nodes of G. The solution to this problem is suggested by E.W. Dijkstra and the algorithm is popularly known as Dijustra's algorithm.

This algorithm finds the shortest paths one by one. If we have already constructed i shortest paths, then the next path to be constructed should be the next shortest path. Let S be the set of vertices to which the shortest paths have already been generated. For z not in S, let $dist[z]$ be the length of the shortest path starting form v_0, going through only those vertices that are in S and ending at z. Let u is the vertex in S to which the shortest path has already been found. If $dist[z] > dist[u] + w(u,z)$ then $dist[z]$ is updated to $dist[u] + w(u,z)$ and the predecessor of z is set to u. The Dijustra's algorithm is presented in Algorithm 6.6.

Algorithm Dijkstra (v_0, W, $dist$, n)

// v_0 is the source vertex, W is the adjacency matrix to store the weights of the links, $dist[k]$ is the array to store the shortest path to vertex k, n is the number of vertices//

```
{
    for (i = 1 ; i < = n ; i ++){
        S[i] = 0;                    // Initialize the set S to empty i.e. i is not inserted into the set//
        dist[i] = w(v0,i) ; //Initialize the distance to each node
    }
    S[vo] = 1;  dist[v0] =  0;
    for (j = 2; j<= n; j++) {
       choose a vertex u from those vertices not in S such that dist [u] is minimum.
        S[u] = 1;
        for (each z adjacent to u with S[z] = 0) {
            if(dist[z] >dist [u] + w[u, z])
                dist[z] = dist [u] + w[u, z];
        }
    }
}
```

Algorithm 6.6. Dijkstra's algorithm for finding shortest path.

Let us consider the graph. The objective is to find the shortest path from source vertex 0 to the all remaining nodes.

Iteration	Set of nodes to which the shortest path is found	Vertex selected	Distance				
			0	1	2	3	4
Initial	{0}	-	0	10	∞	5	∞
1	{0, 3}	3	0	8	14	5	7
2	{0, 3,4}	4	0	8	13	5	7
3	{0, 3, 4, 1}	1	0	8	9	5	7
4	{0, 3, 4, 1, 2}	2	0	8	9	5	7

The time complexity of the algorithm is $O(n^2)$.

Dynamic Programming

Introduction

The Dynamic Programming (DP) is the most powerful design technique for solving optimization problems. It was invented by mathematician named Richard Bellman inn 1950s. The DP in closely related to divide and conquer techniques, where the problem is divided into smaller sub-problems and each sub-problem is solved recursively. The DP differs from divide and conquer in a way that instead of solving sub-problems recursively, it solves each of the sub-problems only once and stores the solution to the sub-problems in a table. The solution to the main problem is obtained by the solutions of these sub- problems.

The steps of Dynamic Programming technique are:

- **Dividing the problem into sub-problems**: The main problem is divided into smaller sub- problems. The solution of the main problem is expressed in terms of the solution for the smaller sub-problems.
- **Storing the sub solutions in a table**: The solution for each sub-problem is stored in a table so that it can be referred many times whenever required.

- **Bottom-up computation**: The DP technique starts with the smallest problem instance and develops the solution to sub instances of longer size and finally obtains the solution of the original problem instance.

The strategy can be used when the process of obtaining a solution of a problem can be viewed as a sequence of decisions. The problems of this type can be solved by taking an optimal sequence of decisions. An optimal sequence of decisions is found by taking one decision at a time and never making an erroneous decision. In Dynamic Programming, an optimal sequence of decisions is arrived at by using the principle of optimality. *The principle of optimality states that whatever be the initial state and decision, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting form the frst decision*.

A fundamental difference between the greedy strategy and dynamic programming is that in the greedy strategy only one decision sequence is generated, wherever in the dynamic programming, a number of them may be generated. Dynamic programming technique guarantees the optimal solution for a problem whereas greedy method never gives such guarantee.

Matrix chain Multiplication

Let, we have three matrices A_1, A_2 and A_3, with order (10 x 100), (100 x 5) and (5 x 50) respectively. Then the three matrices can be multiplied in two ways.

(i) First, multiplying A_2 and A_3, then multiplying A_1 with the resultant matrix i.e. $A_1(A_2 A_3)$.
(ii) First, multiplying A_1 and A_2, and then multiplying the resultant matrix with A_3 i.e. $(A_1A_2) A_3$.

The number of scalar multiplications required in case 1 is 100 * 5 * 50 + 10 * 100 * 50 = 25000 + 50,000 = 75,000 and the number of scalar multiplications required in case 2 is 10 * 100 * 5 + 10 * 5 * 50 = 5000 + 2500 = 7500

To find the best possible way to calculate the product, we could simply parenthesize the expression in every possible fashion and count each time how many scalar multiplications are required. Thus the matrix chain multiplication problem can be stated as "*fnd the optimal parenthesisation of a chain of matrices to be multiplied such that the number of scalar multiplications is minimized*".

Dynamic Programming Approach for Matrix Chain Multiplication

Let us consider a chain of n matrices A_1, A_2.....A_n, where the matrix A_i has dimensions $P[i\text{-}1]$ x $P[i]$. Let the parenthesisation at k results two sub chains A_1.....A_k and A_{k+1}.....A_n. These two sub chains must each be optimal for A_1.....A_n to be optimal. The cost of matrix chain (A_1....A_n) is calculated as $cost(A_1.....A_k)$ + $cost(A_{k+1}.....A_n)$ + cost of multiplying two resultant matrices together i.e.

$cost(A_1.....A_n) = cost(A_1.....A_k) + cost(A_{k+1}.....A_n)$ + cost of multiplying two resultant matrices together.

Here, the cost represents the number of scalar multiplications. The sub chain (A_1....A_k) has a dimension $P[0]$ x $P[k]$ and the sub chain (A_{k+1}.....A_n) has a dimension $P[k]$ x $P[n]$. The number of scalar multiplications required to multiply two resultant matrices is $P[0]$ x $P[k]$ x $P[n]$.

Let $m[i, j]$ be the minimum number of scalar multiplications required to multiply the matrix chain (A_i.....A_j). Then

(i) $m[i, j] = 0$ if $i = j$

(ii) $m[i, j]$ = minimum number of scalar multiplications required to multiply (A_i....A_k) + minimum number of scalar multiplications required to multiply (A_{k+1}....A_n) + cost of multiplying two resultant matrices i.e.

$$m[i, j] = m[i, k] + m[k, j] + P[i-1] \times P[k] \times P[j]$$

However, we don't know the value of k, for which $m[i, j]$ is minimum. Therefore, we have to try all $j - i$ possibilities.

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

Therefore, the minimum number of scalar multiplications required to multiply n matrices $A_1 A_2$.....A_n is

$$m[1, n] = \min_{1 \le k < n} \{ m[1, k] + m[k, n] + P[0] \times P[k] \times P[n] \}$$

The dynamic programming approach for matrix chain multiplication is presented in Algorithm 7.2.

Algorithm MATRIX-CHAIN-MULTIPLICATION (P)

// P is an array of length n+1 i.e. from $P[0]$ to $P[n]$. It is assumed that the matrix A_i has the dimension $P[i-1] \times P[i]$.

```
{
    for(i = 1; i<=n; i++)
        m[i, i] = 0;
    for(l = 2; l<=n; l++){
        for(i = 1; i<=n-(l-1); i++){
            j = i + (l-1);
            m[i, j] = ∞;
        for(k = i; k<=j-1; k++)
                q = m[i, k] + m[k+1, j] + P[i-1] P[k] P[j] ;
                if (q<m [i, j]){
```

```
                    m[i, j] = q;
                    s[i, j] = k;
                }
            }
        }
    }
    return m and s.
}
```

Algorithm 7.2 Matrix Chain multiplication algorithm.

Now let us discuss the procedure and pseudo code of the matrix chain multiplication. Suppose, we are given the number of matrices in the chain is n i.e. $A_1, A_2 \ldots\ldots A_n$ and the dimension of matrix A_i is $P[i-1] \times P[i]$. The input to the matrix-chain-order algorithm is a sequence $P[n+1] = \{P[0], P[1], \ldots\ldots P[n]\}$. The algorithm first computes $m[i, i] = 0$ for $i = 1, 2, \ldots\ldots n$ in lines 2-3. Then, the algorithm computes $m[i, j]$ for $j - i = 1$ in the first step to the calculation of $m[i, j]$ for $j - i = n - 1$ in the last step. In lines 3 – 11, the value of $m[i, j]$ is calculated for $j - i = 1$ to $j - i = n - 1$ recursively. At each step of the calculation of $m[i, j]$, a calculation on $m[i, k]$ and $m[k+1, j]$ for $i \leq k < j$, are required, which are already calculated in the previous steps.

To find the optimal placement of parenthesis for matrix chain multiplication $A_i, A_{i+1}, \ldots\ldots A_j$, we should test the value of $i \leq k < j$ for which $m[i, j]$ is minimum. Then the matrix chain can be divided from $(A_1 \ldots\ldots A_k)$ and $(A_{k+1} \ldots\ldots A_j)$.

Let us consider matrices $A_1, A_2 \ldots\ldots A_5$ to illustrate MATRIX-CHAIN-MULTIPLICATIONalgorithm. The matrix chain order $P = \{P_0, P_1, P_2, P_3, P_4, P_5\} = \{5, 10, 3, 12, 5, 50\}$. The objective is to find the minimum number of scalar multiplications required to multiply the 5 matrices and also find the optimal sequence of multiplications.

The solution can be obtained by using a bottom up approach that means first we should calculate m_{ii} for $1 \leq i \leq 5$. Then $m_{ij}$is calculated for $j - i = 1$ to $j - i = 4$. We can fill the table shown in Fig. 7.4 to find the solution.

Fig. 7.4 Table to store the partial solutions of the matrix chain multiplication problem

The value of m_{ii} for $1 \leq i \leq 5$ can be filled as 0 that means the elements in the first row can be assigned 0. Then

For j – i = 1

$m_{12} = P_0 P_1 P_2 = 5 \times 10 \times 3 = 150$

$m_{23} = P_1 P_2 P_3 = 10 \times 3 \times 12 = 360$

$m_{34} = P_2 P_3 P_4 = 3 \times 12 \times 5 = 180$

$m_{45} = P_3 P_4 P_5 = 12 \times 5 \times 50 = 3000$

For j – i = 2

m_{13} = min {$m_{11} + m_{23} + P_0 P_1 P_3, m_{12} + m_{33} + P_0 P_2 P_3$}

= min {0 + 360 + 5 * 10 * 12, 150 + 0 + 5*3*12}

= min {360 + 600, 150 + 180} = min {960, 330} =

330 m_{24} = min {$m_{22} + m_{34} + P_1 P_2 P_4, m_{23} + m_{44} + P_1 P_3 P_4$}

= min {0 + 180 + 10*3*5, 360 + 0 +10*12*5}

= min {180 + 150, 360 + 600} = min {330, 960} =

330 m_{35} = min {$m_{33} + m_{45} + P_2 P_3 P_5, m_{34} + m_{55} + P_2 P_4 P_5$}

= min {0 + 3000 + 3*12*50, 180 + 0 + 3*5*50}

= min {3000 + 1800 + 180 + 750} = min {4800, 930} = 930

For j – i = 3

m_{14} = min {$m_{11} + m_{24} + P_0 P_1 P_4, m_{12} + m_{34} + P_0 P_2 P_4, m_{13} + m_{44} + P_0 P_3 P_4$}

= min {0 + 330 + 5*10*5, 150 + 180 + 5*3*5, 330+0+5*12*5}

= min {330 + 250, 150 + 180 + 75, 330 +300}

= min {580, 405, 630} = 405

m_{25} = min {$m_{22} + m_{35} + P_1 P_2 P_5, m_{23} + m_{45} + P_1 P_3 P_5, m_{24} + m_{55} + P_1 P_4 P_5$}

= min {0 + 930 +10*3*50, 360+3000+10*12*50, 330+0+10*5*50}

= min {930 + 1500, 360 +3000+6000, 330+2500}

= min {2430, 9360, 2830} = 2430

For j - i = 4

m_{15} = min{$m_{11} + m_{25} + P_0 P_1 P_5, m_{12} + m_{35} + P_0 P_2 P_5, m_{13} + m_{45} + P_0 P_3 P_5, m_{14} + m_{55} + P_0 P_4 P_5$ }

= min{0+2430+5*10*50, 150+930+5*3*50, 330+3000+5*12*50,

405+0+5*5*50}

= min {2430+2500, 150+930+750, 330+3000+3000, 405+1250}

= min {4930, 1830, 6330, 1655} = 1655

Hence, minimum number of scalar multiplications required to multiply the given five matrices in 1655.

To find the optimal parenthesization of $A_1 \ldots\ldots\ldots A_5$, we find the value of k is 4 for which m_{15} is minimum. So the matrices can be splitted to $(A_1 \ldots A_4)(A_5)$. Similarly, $(A_1 \ldots A_4)$ can be splitted to $(A_1 A_2)(A_3 A_4)$ because for $k = 2$, m_{14} is minimum. No further splitting is required as the subchains $(A_1 A_2)$ and $(A_3 A_4)$ has length 1. So the optimal paranthesization of $A_1 \quad A_6$ in $((A_1 A_2)(A_3 A_4))(A_5)$.

Time complexity of multiplying a chain of *n* matrices

Let $T(n)$ be the time complexity of multiplying a chain of n matrices.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \Theta(1) + \sum_{k=1}^{n-1} \left[T(k) + T(n-k) + \Theta(1) \right] & \text{if } n > 1 \end{cases}$$

$$\Rightarrow T(n) = \Theta(1) + \sum_{k=1} \left[T(k) + T(n-k) + \Theta(1) \right] \quad \text{if } n > 1$$

$$= \Theta(1) + \Theta(n-1) + \sum_{k=1}^{n} \left[T(k) + T(n-k) \right]$$

$$\Rightarrow T(n) = \Theta(n) + 2\left[T(1) + T(2) + \ldots + T(n-1) \right] \ldots (7.1)$$

Replacing n by n-1, we get

$$T(n-1) = \Theta(n-1) + 2\left[T(1) + T(2) + \ldots + T(n-2) \right] \ldots (7.2)$$

Subtracting equation 7.2 from equation 7.1, we have

$$T(n) - T(n-1) = \Theta(n) - \Theta(n-1) + 2T(n-1)$$

$$\Rightarrow T(n) = \Theta(1) + 3T(n-1)$$

$$= \Theta(1) + 3\left[\Theta(1) + 3T(n-2) \right] = \Theta(1) + 3\Theta(1) + 3^2 T(n-2)$$

$$= \Theta(1)\left[1 + 3 + 3^2 + \ldots + 3^{n-2} \right] + 3^{n-1} T(1)$$

$$= \Theta(1)\left[1 + 3 + 3^2 + \ldots + 3^{n-1} \right]$$

$$= \frac{3^n - 1}{2} = O(3^n)$$

Longest Common Subsequence

The longest common subsequence (LCS) problem can be formulated as follows "Given two sequences $X = \langle x_1, x_2 \ldots\ldots\ldots x_n \rangle$ and $Y = \langle y_1, y_2 \ldots\ldots\ldots y_n \rangle$ and the objective is to find the LCS $Z = \langle z_1, z_2 \ldots\ldots\ldots z_n \rangle$ that is common to x and y".

Given two sequences X and Y, we say Z is a common sub sequence of X and Y if Z is a subsequence of both X and Y. For example, X = ‹ A, B, C, B, D, A, B› and Y = ‹ B, D, C, A, B, A›, the sequence ‹B, C, A› is a common subsequence. Similarly, there are many common subsequences in the two sequences X and Y. However, in the longest common subsequence problem, we wish to find a maximum length common subsequence of X and Y, that is ‹ B, C, B, A› or ‹ B, D, A, B›. This section shows that the LCS problem can be solved efficiently using dynamic programming.

4.7.1 Dynamic programming for LCS problem

Theorem.4.1.(Optimal Structure of an LCS)

Let $X = \langle x_1, x_2 \ldots\ldots\ldots x_n \rangle$ and $Y = \langle y_1, y_2 \ldots\ldots\ldots y_n \rangle$ be sequences and let $Z = \langle z_1, z_2 \ldots\ldots\ldots z_n \rangle$ be any LCS of X and Y.

Case 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an *LCS* of X_{m-1} and Y_{n-1}. **Case 2.** If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an *LCS* of X_{m-1} and Y. **Case 3.** If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Proof The proof of the theorem is presented below for all three cases.

Case 1. If $x_m = y_n$ and we assume that $z_k \neq x_m$ or $z_k \neq y_n$ then $x_m = y_n$ can be added to Z at any index after k violating the assumption that Z_k is the longest common subsequence. Hence $z_k = x_m = y_n$. If we do not consider Z_{k-1} as *LCS* of X_{m-1} and Y_{n-1}, then there may exist another subsequence W whose length is more than k-1. Hence, after adding $x_m = y_n$ to the subsequence W increases the size of subsequence more than k, which again violates our assumption.

Hence, $Z_k = x_m = y_n$ and Z_{k-1} must be an LCS of X_{m-1} and Y_{n-1}.

Case 2.If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y_n. If there were a common subsequence W of $X_{m-1}$and Y with length greater then k than W would also be an LCS of X_m and Y_n violating our assumption that Z_k is an LCS of X_m and Y_n.

Case 3. The proof is symmetric to case-2.

Thus the LCS problem has an optimal structure.

Overlapping Sub-problems

From theorem 4.1, it is observed that either one or two cases are to be examined to find an LCS of $X_m$and Y_n. If $x_m = y_n$, then we must find an LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$, then we must find an LCS of X_{m-1} and Y_n and LCS of X_m and Y_{n-1}. The LCS of X and Y is the longer of these two LCSs.

Let us define $c[m, n]$ to be the length of an LCS of the sequences X_m and Y_n. The optimal structure of the LCS problem gives the recursive formula

$$c[m,n] = \begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ c[m-1, n-1] + 1 & \text{if } x_m = y_n \\ \max\{c[m-1, n], c[m, n-1]\} & \text{if } x_m \neq y_n \end{cases} \quad \text{.....(7.1)}$$

Generalizing equation 7.1, we can formulate

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{if } x_i \neq y_j \end{cases} \quad \text{.....(7.2)}$$

Computing the length of an LCS

Based on equation (7.1), we could write an exponential recursive algorithm but there are only $m*n$ distinct problems. Hence, for the solution of $m*n$ distinct subproblems, we use dynamic programming to compute the solution using bottom up approach.

The algorithm LCS_length (X, Y) takes two sequences $X = \langle x_1, x_2 \ldots\ldots\ldots x_n \rangle$ and $Y = \langle y_1, y_2 \ldots\ldots\ldots\ldots y_n \rangle$ as inputs and find $c[m, n]$ as the maximum length of the subsequence in X and Y. It stores $c[i, j]$ and $b[i, j]$ in tables $c[m, n]$ and $b[m, n]$ respectively, which simplifies the construction of optimal solution.

Algorithm LCS_LENGTH (X, Y)

```
{
    m=length [X]
    n=length [Y]
    for( i =1; i<=m; i++)
        c[i,0] = 0;
    for(j=0; j<n; j++)
        c[0, j]= 0;
    for(i=1; i< m; i++){
        for(j = 1; j <= n; j++){
            if(x[i] = = y[j]) {
                c[i, j] = 1 + c[i-1, j-1];
                b[i,j] = ' ';
            }
```

```
        else{
            if(c[ i-1, j] ≥ c[i, j-1] )
                c[i, j] = c[i-1, j];
                b[i,j] = '↑';
            else
                c[i,j] = c[i, j-1];
                b[i, j] = '←'
        }
    return c and b ;
}
```

Algorithm 7.3 Algorithm for finding Longest common subsequence .

Constructing an LCS

The algorithm LCS_LENGTH returns *c* and *b* tables. The *b* table can be used to construct the LCS of *X* and *Y* quickly.

AlgorithmPRINT_LCS (*b*, *X*, *i*, *j*)

```
{
    if (i == 0 | | j == 0)
        return;
    if (b[i, j] = = '    {
        PRINT_LCS (b, X, i-1, j-1)
        Print x_i
    }
    else if (b[i, j] = = '↑')
        PRINT_LCS (b, X, i-1, j)
    else
        PRINT_LCS (b, X, i, j-1)
}
```

Algorithm 7.4 Algorithm to print the Longest common subsequence .

Let us consider two sequences X = ‹C, R, O, S, S› and Y = ‹R, O, A, D, S› and the objective is to find the LCS and its length. The c and b table are computed by using the algorithm LCS_LENGTH for X and Y that is shown in Fig.7.5. The longest common subsequence of X and Y is ‹R, O, S› and the length of LCS is 3.

Reliability Design Problem

In this section, we present the dynamic programming approach to solve a problem with multiplicative constraints. Let us consider the example of a computer return in which a set of nodes are connected with each other. Let r_i be the reliability of a node i.e. the probability at which the node forwards the packets correctly in r_i. Then the reliability of the path connecting from one node s to another node d is $\prod_{i=1}^{k} r_i$ where k is the number of intermediate node. Similarly, we can also consider a system with n devices connected in series, where the reliability of device i is r_i. The reliability of the system is $\prod_{i=1}^{k} r_i$. For example if there are 5 devices connected in series and the reliability of each device is 0.99 then the reliability of the system is $0.99 \times 0.99 \times 0.99 \times 0.99 \times 0.99$=0.951. Hence, it is desirable to connect multiple copies of the same devices in parallel through the use of switching circuits. The switching circuits determine the devices in any group functions properly. Then they make use of one such device at each stage.

Let m_i be the number of copies of device D_i in stage i. Then the probability that all m_i have malfunction i.e. $(1-r_i)^m$ .Hence, the reliability of stage i becomes $1-(1-r_i)^m$. Thus, if r_i=0.99 and m_i =2, the reliability of stage i is 0.9999. However, in practical situations it becomes less because the switching circuits are not fully reliable. Let us assume that the reliability of stage i in $\phi_i(m_i)$, $i \le n$. Thus the reliability of the system is $\prod_{i=1}^{k} \phi_i(m_i)$.

Fig. 7.9

The reliability design problem is to use multiple copies of the devices at each stage to increase reliability. However, this is to be done under a cost constraint. Let c_i be the cost of each unit of device D_i and let c be the cost constraint. Then the objective is to maximize the reliability under the condition that the total cost of the system $m_i c_i$ will be less than c.

Mathematically, we can write

maximize $\prod_{1 \le i \le n} \phi_i(m_i)$

subject to $\sum_{1 \le i \le n} c_i m_i \le c$

$m_i \ge 1$ *and* $1 \le i \le n$.

We can assume that each c_i>0 and so each m_i must be in the range $1 \le m_i \le u_i$, where

$$u_i = \left\lfloor \frac{c + \sum_{j=1 \text{ and } j \ne i}^{n} c_j}{c_i} \right\rfloor .$$

The dynamic programming approach finds the optimal solution $m_1,m_2...m_n$. An optimal sequence of decision i.e. a decision for each m_i can result an optimal solution.

Let $f_n(c)$ be the maximum reliability of the system i.e. maximum $\prod_{i=1}^{n} \phi_i(m_i)$, subject to the constraint $\sum c_i m_i \le c$ $1 \le m_i \le$ $1 \le i \le n$. Let we take a decision on the value of m_n from {1,2... u_n}. Then and $1 \le i \le n$ x_i ,

the value of the remaining m_n $1 \le i \le n$ can be chosen in such a way that $\phi(m_i$ for $)$

$1 \le i \le n \le n$ -1 can be maximized under the cost constraint c-$c_n m_n$. Thus the principle of optimality holds and we can write

$$f_n(c) = \max_{1 \le m_n \le u_n} \{ \phi(m_j) f_{j-1}(x - m_j c_j) \} \quad (8.1)$$

We can generalized the equation 8.1 and we can write

$$f_j(x) = \max_{1 \le m_j \le u_j} \{ \phi(m_j) f_{j-1}(x - m_j c_j) \} \quad (8.2)$$

It is clear that $f_0(x)$=1 for all x, $0 \le x \le c$. Let S^i consists of tuples of the form (f,x) where $f = f_i(x)$. There is atmost one tuple for each different x that remits from a sequence of decisions on $m_1,m_2...m_n$. If there are two tuples (f_1,x_1) and (f_2,x_2) such that $f_1 \ge f_2$ and $x_1 \le x_2$ then (f_2,x_2) is said to be dominated tuple and discarded from S^i.

Let us design a three stage system with devices D_1,D_2 and D_3. The costs are Rs 30,Rs 15 and Rs 20 respectively. The cost constraint of the system is Rs 105. The reliability of the devices are 0.9, 0.8 and 0.5 respectively. If stage i has m_i devices in parallel then $\phi(m_i) = (1 - (1 - r)^{m_i})$. We can write c_1=30, c_2 =15,c_3=20, c=105, r_1=0.9, r_2=0.8 and r_3=0.5. We can calculate the value of u_i, for $1 \le i \le 3$

$$x_1 = \left\lfloor \frac{105 - (15 + 20)}{30} \right\rfloor = \left\lfloor \frac{70}{30} \right\rfloor = 2$$

$$x_2 = \left\lfloor \frac{105 - (30 + 20)}{15} \right\rfloor = \left\lfloor \frac{55}{15} \right\rfloor = 3$$

$$x_3 = \left\lfloor \frac{105 - (30 + 15)}{20} \right\rfloor = \left\lfloor \frac{60}{20} \right\rfloor = 3$$

Then we start with S^0={(1,0)}. We can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together.

S^1_1 ={(0.9,30)} S^1_2 ={(0.99,60)} S^1 ={(0.9,30),(0.99,60)}

Considering 1 device at stage q, we can write S^2_1 as follows

S^2_1 ={(0.9 × 0.8,30+15),(0.99 × 0.8,60+15)}

={(0.72,45),(0.792,75)}

Considering 2 devices of D_2 in stage 2, we can compute the reliability at stage 2

$\phi_2(m_2)$=1-(1-0.8)^2=0.96 cost at stage 2=2 ×

15=30 Hence, we can write

S^2_2 ={(0.9 × 0.96,30+30),(0.99 × 0.96,60+30)}

={(0.864,60),(0.9504,90)}

The tuple (0.9504, 90) is removed as it left only Rs 15 and the maximum cost of the third stage is 20.

Now, we can consider 3 devices of D_2 in stage 2 and compute the reliability at stage 2 is

$\phi_2(m_2)$=1-(1-0.8)^3=1-0.008=0.992.

Hence, we can write

S^2_3 ={(0.9 × 0.992,30+45),(0.99 × 0.992,60+45)}

={(0.8928,75),(0.98208,105)}

The tuple (0.98208,105) is discarded as there is no cost left for stage 3. Combining S^2_1, S^2_2 and S^2_3, we get

S^2 ={(0.72,45),(0.792,75),(0.864,60),(0.8928,75)}

The tuple (0.792,75) is discarded from S^2 as it is dominated by

(0.864,60). Now, we can compute S^3 assuming 1 device at stage 3.

S_1^3 ={(0.72 × 0.5,45+20),(0.864 × 0.5,60+20),(0.8928 × 0.5,75+20)}

={(0.36,65),(0.432,80),(0.4464,95)}

If there are 2 devices at stage 3, then

$\phi_3(m_3)$=(1-(1-0.5)^2)=0.75

We can write S_2^3 as follows

S_2^3 ={(0.72 × 0.75,45+40),(0.864 × 0.75,60+40),(0.8928 × 0.75,75+40)}

={(0.54,85),(0.648,100)} (tuple(0.8928 × 0.75,115) is discarded as cost

constraint is 105).

If there are 3 devices at stage 3 then

$\phi_3(m_3)$=(1-(1-0.5)^3)=1-0.125=0.875

Hence, we can write S_3^3 ={(0.72 × 0.875,45+60)}={(0.63,105)}

Combining S_1^3 , S_2^3 and S_3^3 we can write S^3 discarding the dominant tuples ass given below

S^3={(0.36,65),(0.432,80),(0.54,85),(0.648,100)}

The best design has the reliability 0.648 and a cost of 100. Now, we can track back to find the number of devices at each stage. The tuple(0.648,100) is taken from S_2^3 that is with 2 devices at stage 2. Thus m_3 =2.

The tuple (0.648,100) was derived from the tuple (0.864,60) taken from S^2_2 and computed with considering 2 devices at stage 2. Thus m_2=2. The tuple (0.864,60) is derived from the tuple (0.9,30) taken from S^1_1 computed with 1 device at stage 1. Thus m_1 =1.

Bellman Ford Algorithm

In the previous chapter, it is observed that the Dijkstra's algorithm finds the shortest path from one node to another node in a graph *G* = (*V*, *E*) where the weights of the links are positive. However, if there are some negative weight edges then dijkstra's algorithm may fail to find the shortest path from one node to another. Hence, Bellman-ford algorithm solves the single source shortest paths problem in general case. Let us consider the graph *G*= (*V*, *E*) shown in Fig.7.10. Let us assume that node 1 is the source, node 2 and node 3 are destinations. Then by using Dijkstra's algorithm, we can compute the shortest path to node 2 and node 3 as 5 and 7 respectively whereas it is not actually the case. The shortest path from 1 to 3 is 123 and the path length is 2. This can be computed by Bellman-ford algorithm.

Before applying Bellman-ford algorithm, we assume that the negative weight edges are permitted but there should not be any negative weight cycle. This is necessary to answer that the shortest paths consists of a finite number of edges. In Fig.7.11 , the path from 1 to 3 is 121212...123 and the path

length is $-\infty$. When there are no negative weight cycles, the shortest path between any two nodes in n-node graph contains $n-1$ edges.

Let $dist^l[u]$ be the length of the shortest path from source node v to the node u under the constraint that the shortest path contains at most l edges. The $dist^1[u]$=cost[v,u], for $1 \le u \le n$. As we discussed earlier, if there is no negative weight cycle then the shortest path contains atmost$n-1$ edges. Hence, $dist^{n-1}[u]$ in the length of the shortest path from v to u. Our goal is to compute $dist^{n-1}[u]$ and this can be done by using Dynamic programming approach. We can make the following observations

(i) If the shortest path from v to u with at most k, $k>1$ edges has no more than $k-1$ edges, then $dist^k[u] = dist^{k-1}[u]$.

(ii) If the shortest path from v to u with at most k, $k>1$ edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by edge<j, u>. The path from v to j has $k-1$ edges and its length is $dist^{k-1}[j]$. All vertices i such that <i, u>$\in E$ are the candidates of j. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i]+cost[i,u]$ is the correct value for j.

$$Dist^k[u] = \min_{(i,u)\in E}\left\{ dist^{k-1}[u], \min_{(i,u)\in E}\left\{ dist^{k-1}[i] + cost(i,u) \right\} \right\}$$

The Bellman ford algorithm is presented in Algorithm 7.6.

AlgorithmBELLMAN FORD(v,*cost*,*dist*,n)

//v is the source, *cost* is the adjacency matrix representing the cost of edges, *dist* stores the distance to all nodes, n is the number of nodes//

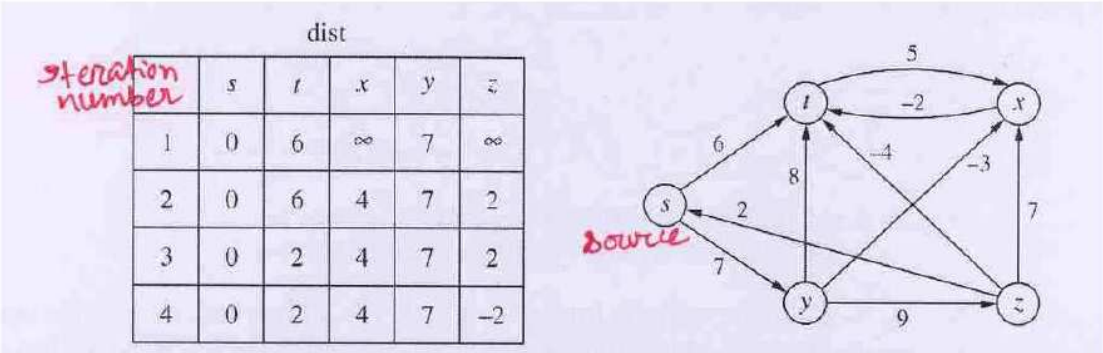
```
{
1   for(i=1;i<=n; i++)
2   dist[i]=cost[v][i];
3   for(k=2; k<= n-1; k++){
4       for each u such that k ≠ v and (j, u)∈E
5           if(dist[u]>dist[j]+cost[j][u])
6               dist[u]=dist[j]+cost[j][u];
7   }//end for k
}
```

Algorithm 7.6 Bellman FordAlgorithm.

The Bellman ford algorithm starts with computing the path from source to each node i in v. Since there are n nodes in a graph, the path to any node can contain atmost$n-1$ nodes. Then for each node u, remaining $n-2$ nodes are required to be examined. If <j, u>$\in E$ then the condition $dist[u]$ is compared

with *dist*[*j*] + cost[*j*][*u*]. If *dist*[*u*] is greater than *dist*[*j*]+*dist*[*j*][*u*] then *dist*[*u*] is updated to *dist*[*j*]+*cost*[*j*][*u*].

Let us consider the example network shown in Fig. 7.12.

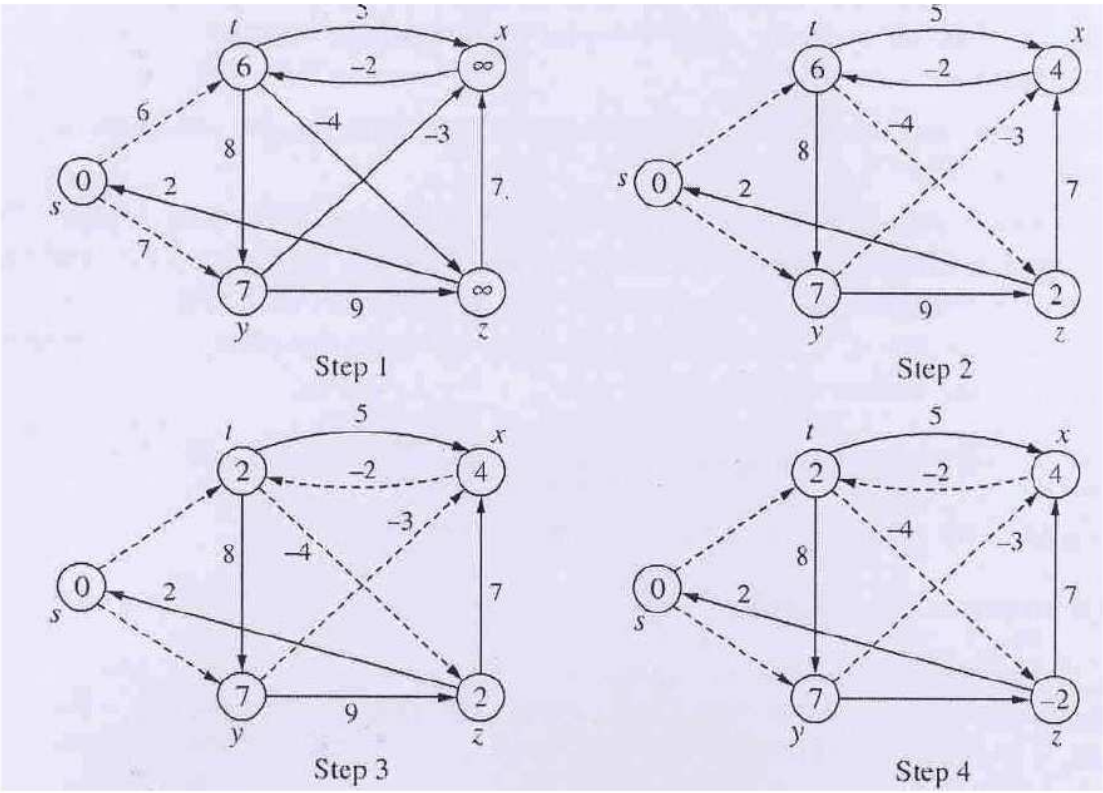


dist

Iteration number	s	t	x	y	z
1	0	6	∞	7	∞
2	0	6	4	7	2
3	0	2	4	7	2
4	0	2	4	7	–2

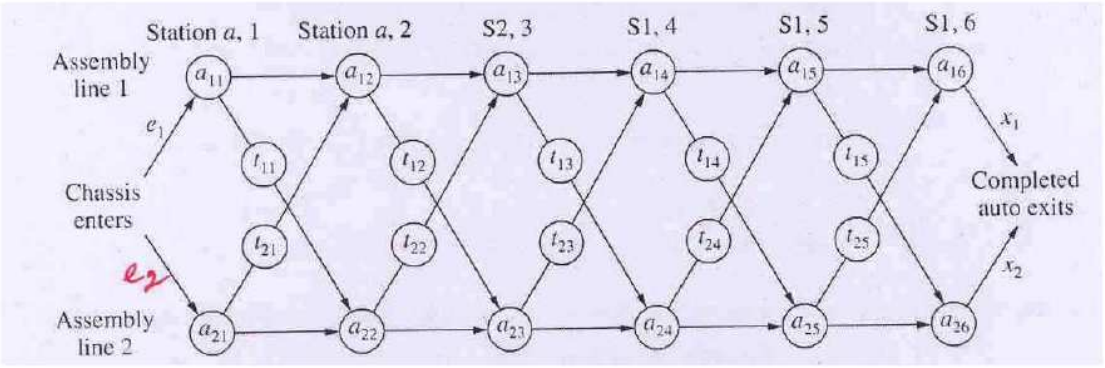
The example graph and its adjacency matric

The time complexity of Bellman ford algorithm is O(*ne*). The lines 1-2 takes O(n) time. If the matrix is stored in an adjacency list then lines 4-6 takes O(e) time. Hence, the lines 3-7 takes O(*ne*) time. Therefore, the several time of Bellman ford algorithm is O(ne).



Stepwise working procedure of Bellman Ford Algorithm

Assembly Line Scheduling



Assembly lines for automobile manufacturing factory

An automobile company has two assembly lines as shown in Fig.7.13.The automobile chassis enters an assembly line and goes through all stations of the assembly line and complete auto exists at the end of the assembly line. Each assembly line has n stations and the j^{th} station of i^{th} line is denoted as S_{ij}. The time required at station S_{ij} is a_{ij} and the time required to travel from one station to the next station is negligible. Normally, the chassis enters in a line goes through all the stations of the same line. However, in overload situations, the factory manager has the flexibility to switch partially completed auto from a station of one line to the next station of other line. The time required to transfer the partially completed auto from station S_{ij} to the other line is t_{ij}; where i=1,2 and j=1,2...n-1. Now the objective to choose some station from line 1 and some station from line 2 so that the total time to manufacture an auto can be minimized.

If there are many stations in the assembly lines the brute force search takes much time to determine the stations through which the auto can be assembled. There are 2^n possible ways to chose stations from the assembly line. Thus determining the fastest way for assembling the auto takes O(2^n) time, which is infeasible when n is large. This problem can be efficiently solved by dynamic programming technique.

The first step of dynamic programming technique is to characterize the structure of an optimal solution. Since, there are 2 assembly lines with stations in each line, the computation of time to move to the 1st station of any assembly line is straight forward. However, there are two choices for j=2,3... n in each assembly line. First, the chassis may come from station $S_{1,j-1}$ and then directly move to $S_{1,j}$ since the time to move from one station to the next station in the same assembly line is negligible. The second choice is the chassis can come from station $S_{2,j-1}$ and then been transferred to station $S_{1,j}$ with a transfer time $t_{2,j-1}$. Let the fastest way through station $S_{1,j}$ is through station $S_{1,j-1}$. Then there must be a fastest way through from the starting point through station $S_{1,j-1}$. Similarly, if there is a fastest way through station $S_{2,j-1}$ then the chassis must have taken a fastest way from the starting point through station $S_{2,j-1}$.

Thus the optimal solution to the problem can be found by solving optimal solution of the sub-problems that in the fastest way to either $S_{1,j-1}$ or $S_{2,j-1}$. This is referred as optimal structure of assembly line scheduling problem.

If we find the fastest way to solve assembly line scheduling problem through station j-1 on either line 1 or line 2. Thus the fastest way through station $S_{1,j}$ is either

- the fastest way to $S_{1,j-1}$ and then directly through station $S_{1,j}$.
- the fastest way to $S_{2,j-1}$ and a transfer from line 2 to line 1 and then through station $S_{1,j}$.

Similarly, the fastest way through station $S_{2,j}$ is

- the fastest way to $S_{2,j-1}$ and then directly through station $S_{2,j}$.
- the fastest way to $S_{1,j-1}$, a transfer time from station 1.

Let $f_i[j]$ be the fastest possible time to get a chassis from the starting point through station $S_{i,j}$ of the assembly line i. Then chassis directly goes to the first station of each line.

$$f_1[1] = e_1 + a_{1,1}$$

$$f_2[1] = e_2 + a_{2,1}$$

Now, we can compute $f_i[j]$, $2<j\le n$ and $1\le i\le 2$ as

$$f_1[j]=\min\{f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}\}$$

$$f_2[j]=\min\{f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}\}$$

If the chassis goes all the way through station n either line 1 or line 2 and then exits, we have

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

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

Let the fastest time to get the chassis all the way through the factory is denoted by f^*. Then

$$f^* = \min\{f_1[n] + x_1, f_2[n] + x_2\}$$

Now, we can compute the stations through which the chassis must move to deliver the end product at minimum time. This can be calculated with a backward approach. Let l^* denote the line whose station n is used in a fastest way through the entire process. If $f_1[n] + x_1 < f_2[n] + x_2$ then $l^*=1$ else $l^*=2$. Then let us denote $l_i[j]$ be the line number 1 or 2 whose station j-1 is used in a fastest way through station $S_{i,j}$. $l_i[1]$ is not required to be calculated since there is no station proceeding to station $S_{i,1}$.

If $f_1[j-1] + a_{1,j} \le f_2[j-1]+t_{2,j-1} + a_{1,j}$ then $l_1[j]=1$.Otherwise, $l_1[j]=2$. Similarly, if $f_2[j-1]+ a_{2,j} \le f_1[j-1]+t_{1,j-1} + a_{2,j}$ then $l_2[j]=2$. Otherwise, $l_2[j]=1$. The algorithm for assembly line scheduling problem is presented in Algorithm7.7.

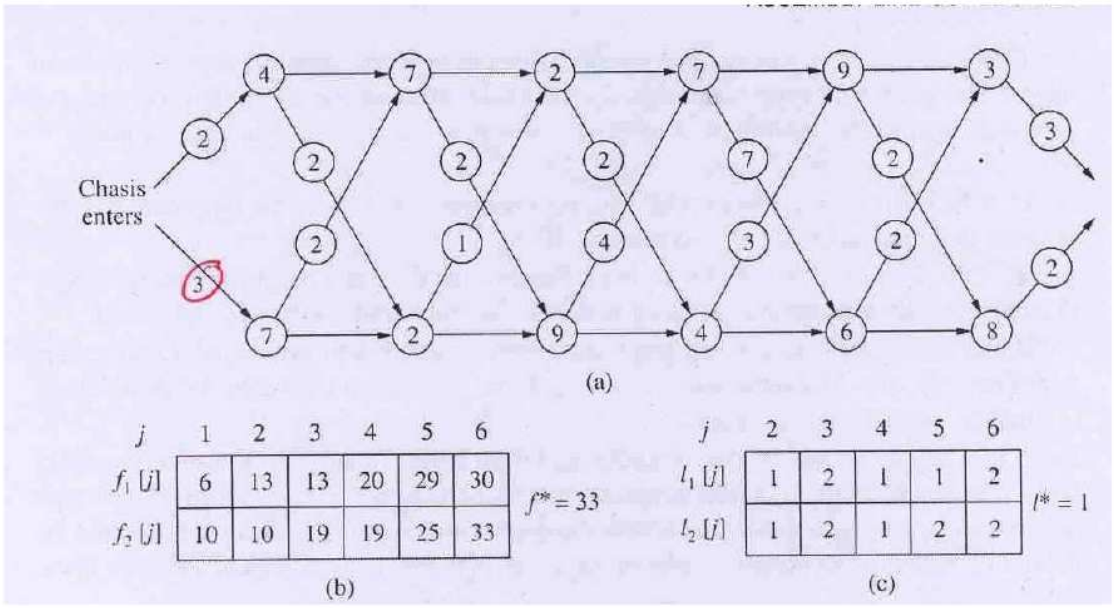


Illustration of assembly line scheduling procedure.

$f_1[1] = e_1 + a_{1,1} = 2 + 4 = 6 \quad f_2[1] = e_2 + a_{2,1} = 3 + 7 = 10$

$f_1[2] = \min\{f_1[1] + a_{1,2}, f_2[1] + t_{2,1} + a_{1,2}\}$

$= \min\{6 + 7, 10 + 2 + 7\} = 13$

$f_2[2] = \min\{f_2[1] + a_{2,2}, f_1[1] + t_{1,1} + a_{2,2}\}$

$= \min\{10 + 2, 6 + 2 + 2\} = 10$

The time required at all the stations of both theassembly lines are shown above. The minimum time required to assemble an auto is 33. The assembly lines through which the complete auto is assembled is shown above.

MODULE - III

- Lecture 21 - Data Structure for Disjoint Sets
- Lecture 22 - Disjoint Set Operations, Linked list Representation
- Lecture 23 - Disjoint Forests
- Lecture 24 - Graph Algorithm - BFS and DFS
- Lecture 25 - Minimum Spanning Trees
- Lecture 26 - Kruskal algorithm
- Lecture 27 - Prim's Algorithm
- Lecture 28 - Single Source Shortest paths
- Lecture 29 - Bellmen Ford Algorithm
- Lecture 30 - Dijkstra's Algorithm

Lecture – 21 Disjoint Set Data Structure

In computing, a **disjoint-set data structure**, also called a **union–find data structure** or **merge–find set**, is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets.

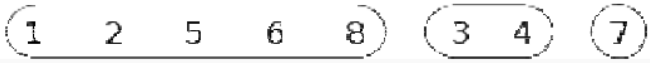
It supports the following useful operations:

- *Find*: Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.
- *Union*: Join two subsets into a single subset.
- *Make Set*, which makes a set containing only a given element (a singleton), is generally trivial. With these three operations, many practical partitioning problems can be solved.

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find*(x) returns the representative of the set that *x* belongs to, and *Union* takes two set representatives as its arguments.

Example :

Make Set creates 8 singletons.



After some operations of *Union*, some sets are grouped together.

Applications :

- partitioning of a set
- Boost Graph Library to implement its Incremental Connected Components functionality.
- implementing Kruskal's algorithm to find the minimum spanning tree of a graph.

•Determine the connected components of an undirected graph.

CONNECTED-COMPONENTS(G)

1. **for** each vertex $v \in V[G]$
2. **do** MAKE-SET(v)
3. **for** each edge $(u,v) \in E[G]$
4. **doif** FIND-SET(u) $\neq$ FIND-SET(v)
5. **then** UNION(u,v)

SAME-COMPONENT(u,v)

1. **if** FIND-SET(u)=FIND-SET(v)
2. **thenreturn** TRUE
3. **elsereturn** FALSE

Lecture 22 - Disjoint Set Operations, Linked list Representation

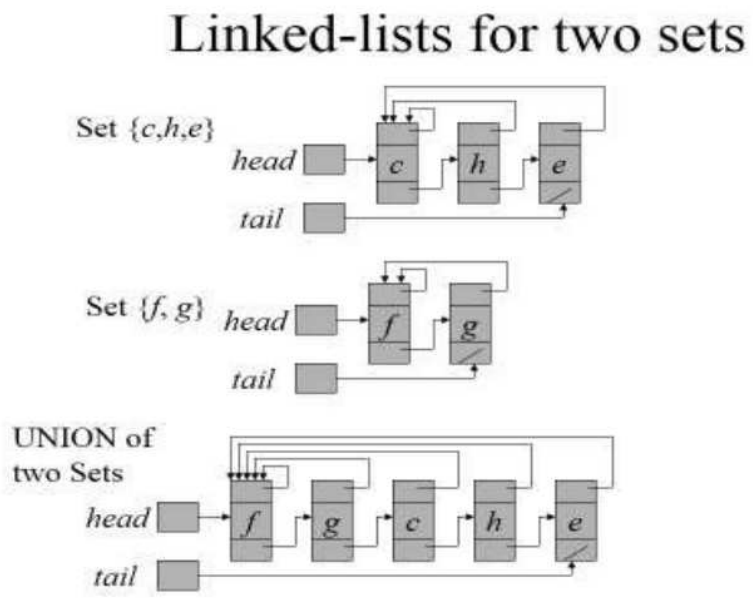
- A disjoint-set is a collection ©={S_1, S_2,…, S_k} of distinct dynamic sets.
- Each set is identified by a member of the set, called *representative*.

Disjoint set operations

- MAKE-SET(x): create a new set with only x. assume x is not already in some other set.
- UNION(x,y): combine the two sets containing x and y into one new set. A new representative is selected.
- FIND-SET(x): return the representative of the set containing x.

Linked list Representation

- Each set as a linked-list, with head and tail, and each node contains value, next node pointer and back-to-representative pointer.
- Example:
- MAKE-SET costs $O(1)$: just create a single element list.
- FIND-SET costs $O(1)$: just return back-to-representative pointer.

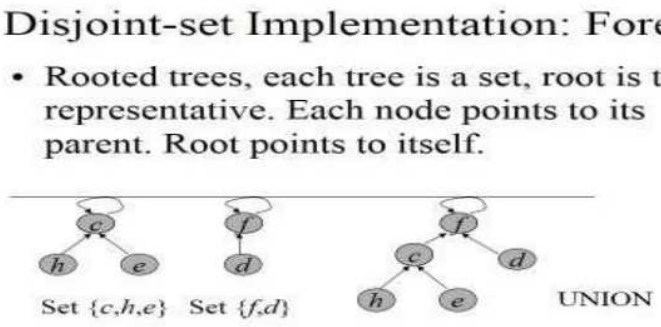


UNION Implementation

- A simple implementation: UNION(x,y) just appends x to the end of y, updates all back-to-representative pointers in x to the head of y.
- Each UNION takes time linear in the x's length.
- Suppose n MAKE-SET(x_i) operations (O(1) each) followed by n-1 UNION
  - UNION(x_1, x_2), O(1),
  - UNION(x_2, x_3), O(2),
  - …..
  - UNION(x_{n-1}, x_n), O(n-1)
- The UNIONs cost 1+2+…+n-1=$\Theta(n^2)$

So 2n-1 operations cost $\Theta(n^2)$, average $\Theta(n)$ each

Lecture 23 - Disjoint Forests

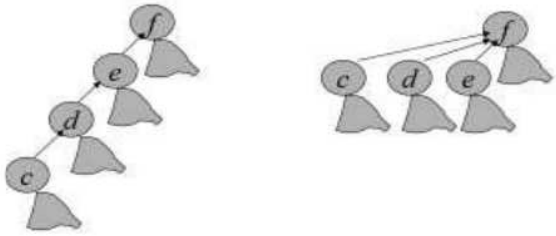


- Three operations
  - MAKE-SET(x): create a tree containing x. O(1)
  - FIND-SET(x): follow the chain of parent pointers until to the root. O(height of x's tree)
  - UNION(x,y): let the root of one tree point to the root of the other. O(1)
- It is possible that n-1 UNIONs results in a tree of height n-1. (just a linear chain of n nodes).
- So n FIND-SET operations will cost $O(n^2)$.

Union by Rank & Path Compression

- **Union by Rank:** Each node is associated with a rank, which is the upper bound on the height of the node (i.e., the height of subtree rooted at the node), then when UNION, let the root with smaller rank point to the root with larger rank.
- **Path Compression:** used in FIND-SET(x) operation, make each node in the path from x to the root directly point to the root. Thus reduce the tree height.

Path Compression



Algorithm for Disjoint-Set Forest

```
MAKE-SET(x)
1.  p[x]←x
2.  rank[x]←0
```

```
UNION(x,y)
1. LINK(FIND-SET(x),FIND-SET(y))
```

```
LINK(x,y)
1.  if rank[x]>rank[y]
2.  then p[y] ←x
3.  else p[x] ←y
4.       if rank[x]=rank[y]
5.       then rank[y]++
```

```
FIND-SET(x)
1.  if x≠ p[x]
2.     then p[x] ←FIND-SET(p[x])
3.  return p[x]
```

Worst case running time for m MAKE-SET, UNION, FIND-SET operations is: $O(m\alpha(n))$ where $\alpha(n)\leq 4$. So nearly linear in m.

Lecture 24 - Graph Algorithm - BFS and DFS

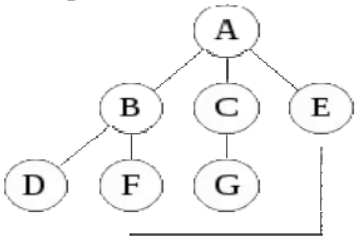
In graph theory, **breadth-first search** (**BFS**) is a strategy for searching in a graph when search is limited to essentially two operations:
(a) visit and inspect a node of a graph;
(b) gain access to visit the nodes that neighbor the currently visited node.

- The BFS begins at a root node and inspects all the neighboring nodes.
- Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on.
- Compare BFS with the equivalent, but more memory-efficient.

Historical Background

- BFS was invented in the late 1950s by E. F. Moore, who used to find the shortest path out of a maze,
- discovered independently by C. Y. Lee as a wire routing algorithm (published 1961).

Example



A BFS search will visit the nodes in the following order: A, B, C, E, D, F, G

BFS Algorithm

The algorithm uses a queue data structure to store intermediate results as it traverses the graph, as follows:

1. Enqueue the root node
2. Dequeue a node and examine it
   - If the element sought is found in this node, quit the search and return a result.
   - Otherwise enqueue any successors (the direct child nodes) that have not yet been discovered.
3. If the queue is empty, every node on the graph has been examined – quit the search and return "not found".
4. If the queue is not empty, repeat from Step 2.

Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes *u* and *v* (with path length measured by number of edges)
- Testing a graph for bipartiteness
- (Reverse) Cuthill–McKee mesh numbering
- Ford–Fulkerson method for computing the maximum flow in a flow network
- Serialization/Deserialization of a binary tree vs serialization in sorted order, allows the tree to be re-constructed in an efficient manner.

Pseudo Code

Input: A graph *G* and a root *v* of G

```
1  procedure BFS(G,v) is
2      create a queue Q
3      create a set V
4      add v to V
5      enqueuev onto Q
6      whileQ is not empty loop
7          t ← Q.dequeue()
8          ift is what we are looking for then
9              returnt
10         end if
11         for all edges e in G.adjacentEdges(t) loop
12             u ← G.adjacentVertex(t,e)
13             ifu is not in Vthen
14                 add u to V
15                 enqueueu onto Q
16             end if
17         end loop
18     end loop
19     return none
20 end BFS
```

Time and space complexity

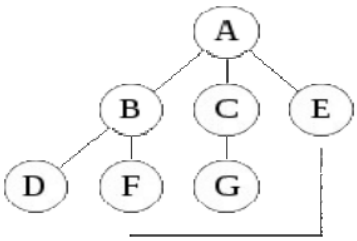
The time complexity can be expressed as $O(|V| + |E|)$[3] since every vertex and every edge will be explored in the worst case. Note: $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$ depending on how sparse the input graph is.

When the number of vertices in the graph is known ahead of time, and additional data structures are used to determine which vertices have already been added to the queue, the space complexity can be expressed as $O(|V|)$ where $|V|$ is the cardinality of the set of vertices. If the graph is represented by an Adjacency list it occupies $\Theta(|V| + |E|)$[4] space in memory, while an Adjacency matrix representation occupies $\Theta(|V|^2)$.

Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Historical Background
A version of depth-first search was investigated in the 19th century by French mathematician **Charles Pierre Trémaux**

Example



A DFS search will visit the nodes in the following order: A, B, D, F, E, C, G

Pseudo Code

Input: A graph *G* and a vertex *v* of G

Output: All vertices reachable from *v* labeled as discovered

A recursive implementation of DFS

```
1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

A non-recursive implementation of DFS

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      whileS is not empty
5            v ← S.pop()
6            ifv is not labeled as discovered:
7                label v as discovered
8                for all edges from v to winG.adjacentEdges(v) do
9                    S.push(w)
```

Applications

- Finding connected components.
- Topological sorting.
- Finding the bridges of a graph.
- Generating words in order to plot the Limit Set of a Group.
- Finding strongly connected components.
- Planarity testing
- Solving puzzles with only one solution, such as mazes. (DFS can be adapted to find all solutions to a maze by only including nodes on the current path in the visited set.)
- Maze generation may use a randomized depth-first search.
- Finding bi-connectivity in graphs.