
Subject: Analysis and Design of Computer Algorithms
Paper Code: MCA 403 **Author: Sh. Ganesh Kumar**
Lesson: Divide And Conquer Strategy **Vetter: Ms Jyoti**
Lesson No. : 01

STRUCTURE

1.1 INTRODUCTION

Given a function that has to compute on 'n' input the divide and conquer strategy suggest splitting k distinct subsets, $1 < K \leq n$, that yields K sub problems.

Each of these sub problems must be solved and the solutions are obtained. A method must be found to combine the sub solutions into a solution of a whole. If the sub problems are still relatively large, the divide and conquer strategy can be reapplied. The sub problems are of the same type as the original problem.

We can write a control abstraction that gives a flow of the control but whose primary operations are specified by other procedures. The meaning of these procedures is left undefined.

D and C (P) where P is the problem to b e solved. A Boolean valued function that determines if the input size is small enough that the answer can be computed without splitting. If it is so the function S is invoked else the problem P is divided into smaller sub problems.

The sub problems P_1, P_2,…..P_k can be solved by recursive applications of D and C. Combine is a function that determines the solution to P, using the solution the K sub problems.

Algorithm D and C (P)

{if small (p) the return S (P);

else

{divide P into smaller instances

P_1, P_2,…..P_k $K \geq I$

Apply D and C to each to these sub problems.

Return combine (D and C (P_1), D and C (P_2) ….D and C (P_K));

}

}

if the size of P is n and the size of K sub problems. are n_1, n_2…n_k respectively the computing time of D and C is described by the recurrence relation.

T(n) = { g(n) where n is the small

$T(n_1) + T(n_2)$…….+$t(n_k) + f(n)$

T(n) is the time for D and C is any input n.

g(n) is the time to compute the sum directly for small inputs.

f(n) is the time for dividing D and for combining the solutions for the sub problems.

When the sub problems are of the same type as the original problems. We describe the algorithm using recursion.

1.2 BINARY SEARCH

We assume that we have n ≥ 1 distinct integers which are stored in the increasing order. We are required to determine whether a given integer x is present in the list if x is present then determine a value J such that x = A[j]. If x is not in the list then j is to be set to O.

Let A = {n, i,…..a_e,x), where n is the number of element in the list (a_x,….a_e) is the list of element and x is the element searched for.

By making use of the divide and conquer that the set is stored, let A(mid) be the middle element.

There are three possibilities-

i) x < A (mid)

In this case x can only occur as A(1) to A(mid-1).

ii) x > A(mid)

In this case x can only occur as A(mid +1) to A(n)

iii) x = A(mid)

In this case set j to mid and return continue in this way by keeping two pointer, lower and upper, to indicate the range of elements.

Algorithm

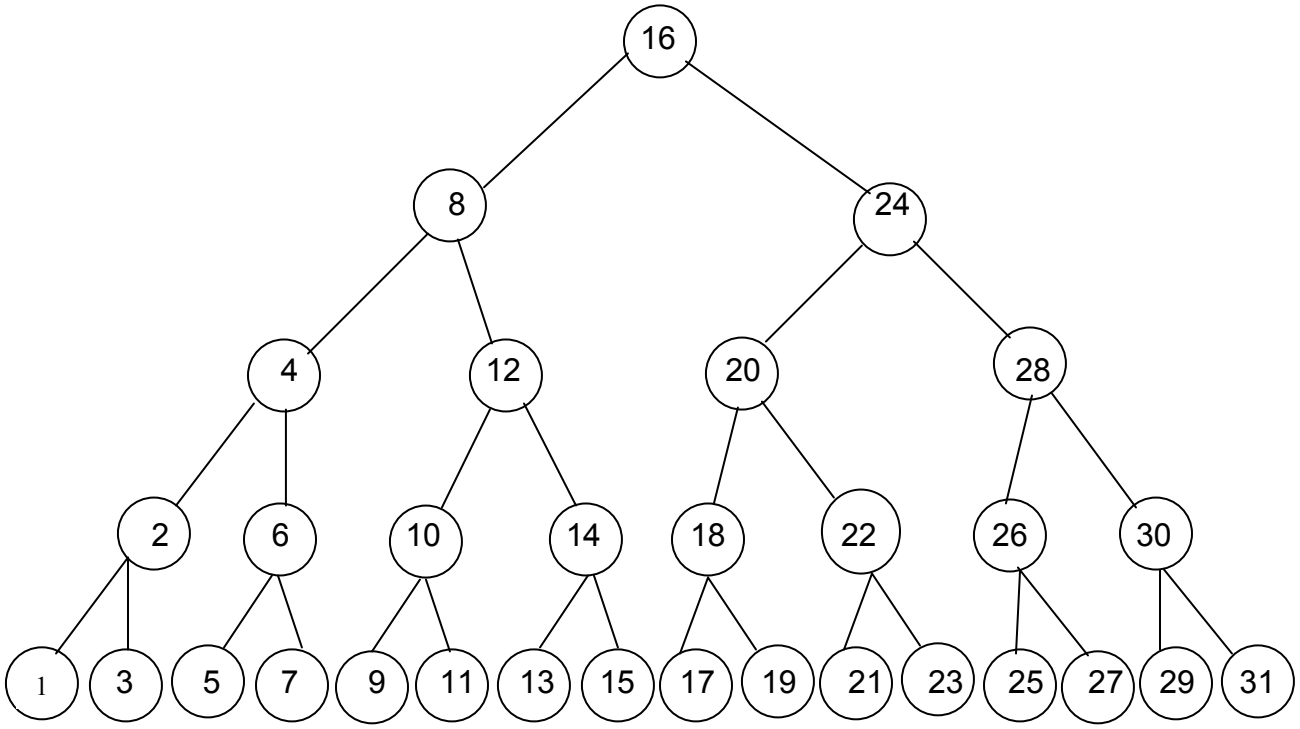
BINSRCH (A, n, x, j)

1- Lower ←1, i upper ← n
2- While lower ≤ upper do
3- Mid ← [lower + upper) 12]
4- Case
5- : x > A(mid): lower ← mid + 1
6- : x < A(mid): upper ← mid – 1
7- : else : j ← mid; return
8- end
9- end
10- J ← O

end

In the binary search method, it is always the key in the middle of the subfile currently being searched that is used for comparison. This splitting process can be described by drawing a binary decision tree.

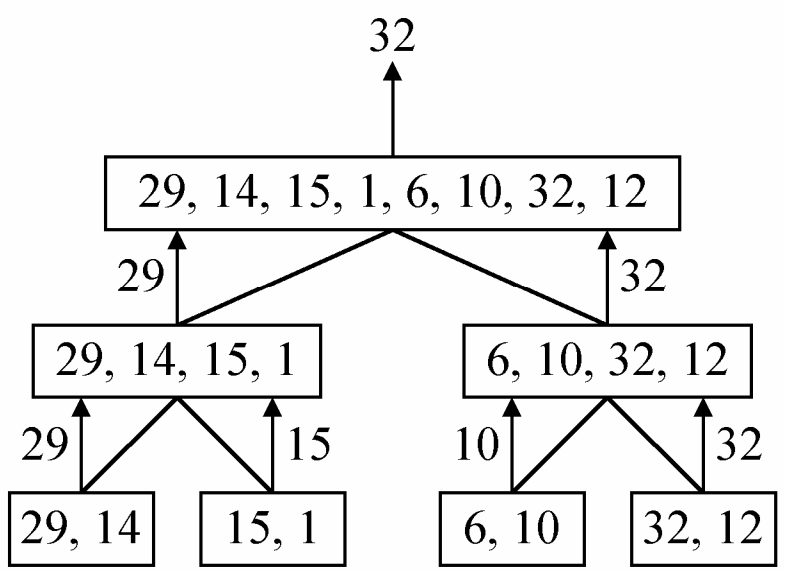
Suppose there are 31 records then the first key tested is K_{16} since [(1+31)/2] =16. If K is less than $K_{16,}$ since K_8 is tested, next [(1+15)/2] = 8 or if K is greater than K_{16} then K_{24} is tested. The binary tree is :



1.3 Finding Maximum and Minimum

The Divide-and-Conquer Strategy

e.g. find the maximum of a set S of n numbers



time complexity:

$$T(n)=\begin{cases} 2T(n/2)+1 & , n>2 \\ 1 & , n\leq 2 \end{cases}$$

assume $n = 2^k$

$T(n) = 2T(n/2)+1$

$= 2(2T(n/4)+1)+1$

$= 4T(n/4)+2+1$

:

$=2^{k-1}T(2)+2^{k-2}+\ldots+4+2+1$

$=2^{k-1}+2^{k-2}+\ldots+4+2+1$

$=2^k-1 = n-1$

A general divide-and-conquer algorithm:

Step 1: If the problem size is small, solve this problem directly; otherwise, split the original problem into 2 sub-problems with equal sizes.

Step 2: Recursively solve these 2 sub-problems by applying this algorithm.

Step 3: Merge the solutions of the 2 sub-problems into a solution of the original problem.

time complexity:

$$T(n)=\begin{cases} 2T(n/2)+S(n)+M(n) & , n \geq c \\ b & , n < c \end{cases}$$

where S(n): time for splitting

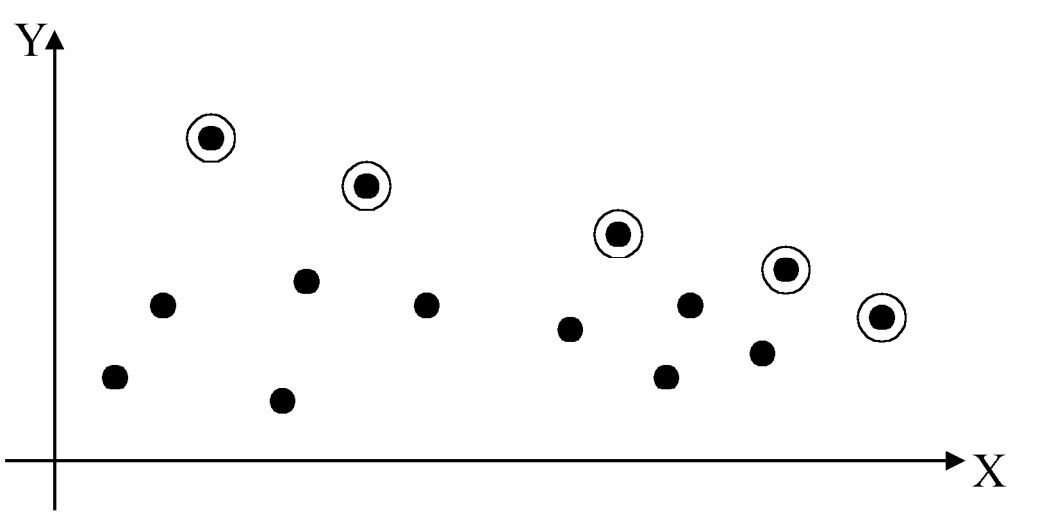
M(n): time for merging

b: a constant

c: a constant.

- **2-D maxima finding problem**

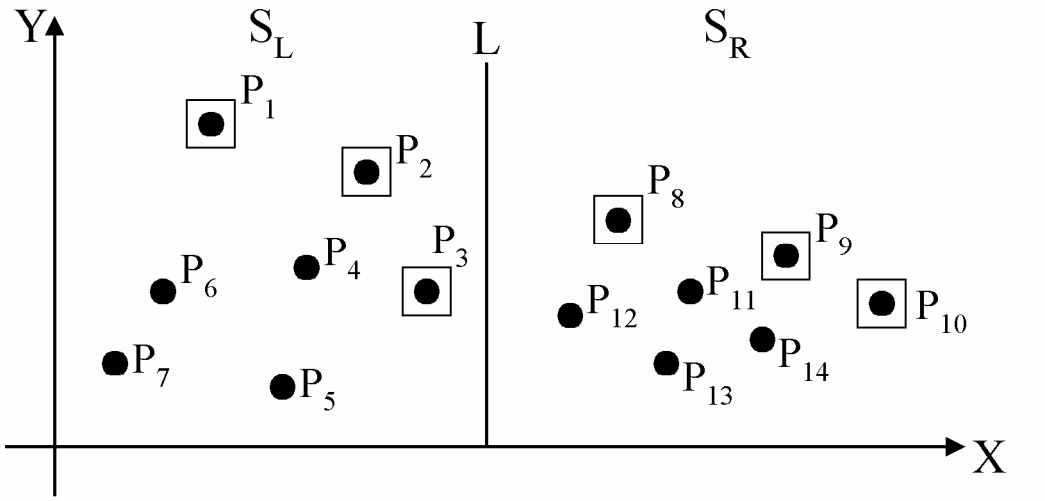
Def: A point (x_1, y_1) dominates (x_2, y_2) if $x_1 > x_2$ and $y_1 > y_2$. A point is called a maxima if no other point dominates it.



Straightforward method:

compare every pair of points

time complexity: $O(n^2)$.



The maximal of S_L and S_R

Algorithm 5.1 A Divide-and-Conquer Approach to Find Maximal Points in the Plane

Input: A set of n planar points.

Output: The maximal points of S.

Step 1. If S contains only one point, return it as the maxima. Otherwise, find a line L perpendicular to the X-axis which separates the set of points into two subsets S_L and S_R, each of which consisting of n/2 points.

Step 2. Recursively find the maximal points of S_L and S_R.

Step 3. Project the maximal points of S_L and S_R onto L and sort these points according to their y-values. Conduct a linear scan on the projections and discard each of the maximal points of S_L if its y-value is less than the y-value of some maximal point of S_R.

time complexity:

$$T(n)=\begin{cases} 2T(n/2)+O(n)+O(n \log n) & , n > 1 \\ 1 & , n = 1 \end{cases}$$

Assume $n = 2^k$

$$T(n) = O(n \log n) + O(n \log^2 n)$$
$$= O(n \log^2 n)$$

Improvement:

(1)Step 3: Find the largest y-value of S_R.

time complexity:

$$T(n)=\begin{cases} 2T(n/2)+O(n)+O(n) & , n > 1 \\ 1 & , n = 1 \end{cases}$$

$\Rightarrow T(n) = O(n \log n)$

(2)The sorting of y-values need be done only once(only one presorting).

No sorting is needed in Step3.

time complexity:

$$O(n \log n) + T(n) = O(n \log n)$$

where

$$T(n)=\begin{cases} 2T(n/2)+O(n)+O(n) & , n > 1 \\ 1 & , n = 1 \end{cases}$$

1.4 MERGE SORT

Using the divide and conquer strategy, sorting algorithm called the merge sort, we split a given set of numbers into two equal sized sets and a combining operation is the merging of two sorted sets into one.

Lets consider a sequence of n elements (a [1] to a [n/2]) and (a [n/2] to a[n]. Each set has to be individually sorted and the resulting sorted sequences are merged which produce a single sorted sequence of n elements.

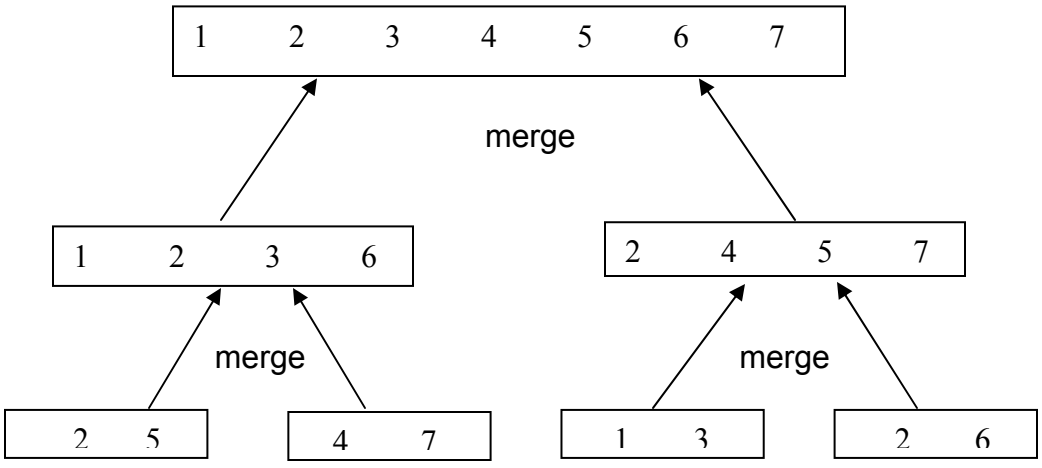
The merge sort algorithms uses recursion and a function merge, which merges two sorted sets.

Algorithm:-

```
merge sort (low, high)
{
if (low < high) then
{
mid: = [(low +high)/2]
mergesort (low, mid);
mergesort (mid +1, high);
mergesort (low, mid, high) ;
}
}
Algorithm merge (low, mid, high)
{
I: = low; j: = low; k: = mid + 1;
While ((n ≤ mid) and (k ≤ high)) do
{
if (a[n] ≤ a[k]) then
{
b[ j ] : = a[n];
```

```
n : = n + 1;
}
else
{
b [ j ] : =a[ k ];
k: = k + 1;
}
j : = j + 1;
}
if (i > mid) then
for L : =  k to high do
{
b[ j ] : = a[L];
j : = j + 1;
}
else
for L: = i to mid do
{
b [ j ]: = a [L];
j: = j + 1;
}
for L: = low to high do
a[L]: = b[L];
}
```

Sorted Sequence



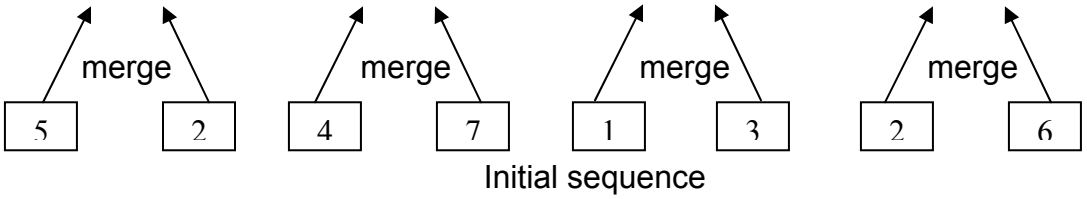


Figure : The operation of merge sort on the array A = (5, 2, 4, 7, 1, 3, 2, 6).

The length of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

The time for merging operation is proportional to n. The computing time for merge sort is :

$T(n) = \{a$ n = 1, 'a' is constant

$2T(n/2) + Cn$; n>1, C is constant.

When n is a power of 2 is $n = 2^k$ using

Successive substitutions:

$T(n) = 2\{2T(n/4) + (n/2)\} + Cn$

$= 4T(n/4) + 2Cn$

$= 2^2 T(n/2^2) + 2Cn$

$= 2^K T(n/2^k) + K Cn$ $[2^K = n]$

$= 2^K T(1) + K Cn$

$= 2^K a + K Cn$

$= na + Cn \log n$ $[K = \log_2 n]$

Hence merge sort is o [n log n].

1.5 QUICK SORT

Quick sort, like merge sort, is based on the divide and conquer paradigm. Here is the three step divide and conquer process for sorting a subarray A[P…..r].

Divide : Partition (rearrange) the array A[P…..r] into two (possibly empty) subarrays A[P….q-1] and A[q+1…..r] such that each element of A[P….q-1] is less than or equal to A[q], which is less than or equal to each element of A[q+1….r]. Compute the index q as part of this partitioning procedure.

Conquer : Sort the two subarray A[P….q-1] and A[q+1….r] by recursive calls to quicksort.

Combine : The subarrays are sorted in place. There is no needed to combine them; the entire array A[P….r] is now sorted.

Algorithm :

Quicksort (A, P, r)

1- if P < r

2- then q ← partition (A, P, r)

3- Quicksort (A, P, q-1)

4- Quicksort (A, q+1, r

To sort an entire array A, the initial call is quicksort (A, q, length [A]).

Partitioning the array

Partition (A, P, r)

1- x ← A [r]

2- i ← P-1

3- for j ← P to r – 1

4- do if A[j] ≤ x

5- then i ← i + 1

6- enchage A[i] ↔ A[j]

7- exchange A[i + 1] ↔ A[r]

8- return i + 1

1.6 STRASSEN'S MATRIX MULTIPLICATION

Let A and B be two n x n matrices. The product matrix C = AB is also an n x n matrix. Whose i and j^{th} elements is formed by taking [i, j] the elements in the i^{th} column of B and multiplying them to give c [i j] = ∑ A [i k] B [k,j], 1 ≤ k < n for all i and j between 1 and n. To compute c[i, j] using this formula we require n^3 multiplications.

The divide and conquer strategy suggest another way to compute the product of two n x n matrices. We will assume that n is a power of 2. in case n is

not a power of 2 then enough rows and columns of zeros may be added to both A and B so that the resulting dimensions are a power of 2.

Imagine that A and B are each partitioned into 4^2 sub matrices each having dimension n/2 x n/2 then the product AB can be computed by using the above formula for the product of 2 x 2 matrices.

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$

$C_{12} = A_{11} B_{12} + A_{12} B_{22}$

$C_{21} = A_{21} B_{11} + A_{22} A_{21}$

$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

if n = 2 then the above formula is computed using a multiplication operation for the elements of A and B.

For n > 2 the elements of C can be computed using matrix multiplication and addition operations applied to matrices of size n/2 x n/2. Since n is a recursively computed by the same algorithm for n x n case.

Strassen formulated a method to reduce the number of matrix multiplication so as to reduce the complexity of the algorithm. This method uses only 7 multiplications and 18 addition or subtractions. The method involves first computing 7 n/2 x n/2 matrices p, q, r, s, t, u and v as below :

$P = (A_{11} + A_{22}) (B_{11} + B_{22})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} – B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} - A_{11}) (B_{11} + B_{12})$

$V = (A_{12} – A_{22}) (B_{21} + B_{22})$

C_{11} = P+S-T+V

C_{12} = R + T

C_{21} = P+R-Q+U

The resulting time complexity is

$$T(n) = \begin{cases} bn \leq 2 \\ 7T(n/2) + an2n > 2 \end{cases}$$

Where a and b are constants.

Limitations of Strassen's Algorithm

From a practical point of view , Strassen's algorithm is often not the method of choice for matrix multiplication, for the following four reason:

1. The constant factor hidden in the running time of Strassen's algorithm is larger than the constant factor in the native $\theta(n^3)$ method.
2. When the matrices are sparse, methods tailored for sparse matrices are faster.
3. Strassen's algorithm is not quite as numerically stable as the native method.
4. The sub matrices formed at the levels of recursion consume space.

1.7 SUMMARY

The unit discusses various issues in respect of the technique viz., Divede and Conquer for designing and analyzing algorithms for solving problems. First , the general plan of the Divide and conquer technique is explained . the issue of whether at some stage to solve a problem directly or whether to further subdivide it, is discussed in terms of the relative efficiencies in the two alternative cases.

The technique is illustrated with examples of its applications to solving problems of Binary Search, Sorting , of finding maximum, of finding minimum of given data and matrix multiplication. Under sorting , the well- known techniques viz., Merge-sort and quick-sort are discussed in detail.

1.8 KEYWORDS

Divide: the problem into a number of sub problems.

Conquer: the sub problems by solving them recursively. If the sub problem sizes are small enough , however , just solve the sub problems in a straightforward manner.

Combine: the solutions to the sub problems into the solution for the original problem.

1.9 REVIEW QUESTIONS

1. A sorting method is said to be stable if at the end of the method, identical elements occur in the same order as in the original unsorted. Is merge sort a stable sorting method?
2. Apply the Quick sort Algorithm to sort the numbers.
   65, 70, 75, 80, 85, 60, 55, 50, 45.
3. Explain the method used by Strassen to reduce the complexity for matrix multiplication.
4. What is the worst case complexity of Quick sort and when does it occur?
5. Apply the divide and conquer strategy to select a particular number x from a array of 10 numbers.
6. **Multiply the following two matrices using Strassen's algorithm**

$$\begin{bmatrix} 5 & 6 \\ -5 & 3 \end{bmatrix} \text{ and } \begin{bmatrix} -7 & 6 \\ 5 & 9 \end{bmatrix}$$

1.10 FURTHER READINGS

1. Horowitz E and Sahni S., "Fundamental of Computer Algorithm" Galgotia Publications.
2. Aho A. V. Hoperoft, J.E. and Ullman, J.D., "Design and Analysis of Algorithm" Addison Wesley.
3. D. Harel., " Algorithmics : The Spirit of Computing" Addison Wesley.

Subject: Analysis and Design of Computer Algorithms
Paper Code: MCA 403 **Author: Sh. Ganesh Kumar**
Lesson: Greedy Method **Vetter: Jyoti**
Lesson No. : 02

STRUCTURE

2.1 INTRODUCTION

The Greedy method is a designed technique that can be applied to a wide variety of problems. Most of these problems have **n** inputs and require us to obtain a subset that satisfies these constraints. Any subset that satisfies these constraints is called a 'feasible solution'. Feasible solutions either maximize or minimize a given 'objective' function. 'A feasible solution that does this is called an optimal solution'.

The Greedy method devices an algorithm that works in stages considering one input at a time. At each stage a decision is made regarding whether a particular input is in an optimal solution. That is done by considering the inputs in an order determined by

some selection procedure. If adding that input leads to an infeasible solution then that input is not added to the partial solution. Else its added.

```
Algorithm Greedy (a, n)
{
        Solution : = Ø ;
        for  I : = 1 to n do
        {
                x : = select (a) ;
                if  feasible (solution, x) then
                        solution : = Union (solution, x) ;
        }
        Return solution;
        }
```

The function select, selects an input from a [ ] and removes it. The selected input value is assigned to x.

Feasible is a Boolean valued function that determines whether x can be included into the solution vector. The function union combines x with solution and update the objective function.

Problems that call for the selection of an optimal subset use the version of the Greedy technique called as the subset paradigm.

Problems that make decision by considering the inputs in some order and where each decision is made using an optimization criteria that can be computed using decisions already made, use the version of Greedy method called as the ordering paradigm.

Example 2.1: Let us suppose that we have to go from city A to city E through either city B or city C or city D with costs of reaching between pairs of cities as shown below :

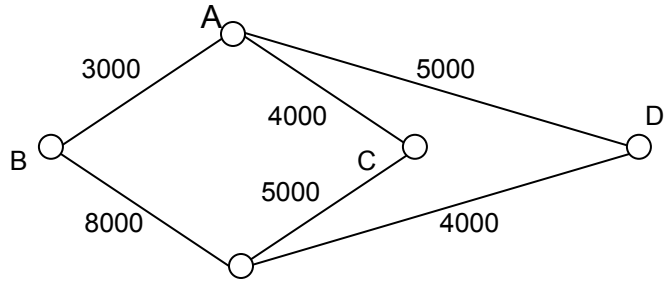


Figure 2.1.1

Then the Greedy technique suggests that we take the route from A to B, the cost of which Rs. 3,000 is the minimum among the tree costs, (viz., Rs. 3,000, Rs. 4,000 and Rs. 5,000) of available routes.

However, at B there is only one route available to reach E. Thus, Greedy algorithms suggest the route from A to B to E, which costs Rs.11000. But, the route from A to C to E, costs only Rs. 9000. Also the route from A to D to E costs also Rs.9000. Thus locally better solution, at some stage, suggested by Greedy technique yields overall (or globally) costly solution.

2.2 OPTIMAL STORAGE ON TAPES

There are n programs that are to be sorted on a computer tape of length I. Associated with each program i is a length li where $1 \le i \le n$.

Programs can be stored on the tape if and only if the sum of the lengths of the programs is at the most I. if the programs are stored in the order

$I = i_1\ i_2\ i_3 \ldots\ldots i_n$ in the time tj needed to retrieve the program ij is proportional to $\sum_{i \le k \le j} l_{ik}$ if all programs are retrieved $i \le k \le j$

Equally often then the expected or mean retrieved time (MRT) is $1/n \sum_{i \le j \le n} t_j$

In the optimal storage on tape problem we are required to find a permutation for the n programs so that when they are stored on the tape in this order the MRT is minimized. This problem fits the ordering paradigm.

Minimizing the MRT is equivalent to minimizing d (I) $\sum_{i \le j \le n} \sum_{i \le k \le j} l_{ik}$

Using a Greedy approach we choose the next program on the basis of some optimization measure. The next program to be stored on the tape would be the one that minimizes the increases in 'd' if we have already constructed the permutation $i_1, i_2 \ldots\ldots i_r$ then appending program j gives the permutaion $i_1\ i_2\ i_3 \ldots.. i_{r+1} = j$.

This increases the d value by $\sum l_{ik} + l_i$. Since $\sum_{i \le i \le n} l_{ik}$ is fixed an independent

of j, we trivially observe that the increase in d is minimized if the next program chosen is the one with the least length from among the remaining programs.

The tape storage problem can be extended to several tapes T_0, T_1…..T_{m-1}

Then the programs are to be distributed over these tapes

The total retrieval time (TD) is

$$TD = \sum_{0 \le j \le m-1} d(l_j)$$

The objective is to store the programs in such away so as to minimize (TD). If the j initially ordered so that $l_i \le l_2 \le \ldots \le l_n$. Then the first n programs are assigned to tapes T_0, T_1….T_{m-1} respectively. The next m programs will be assigned to tapes T_0, T_1…T_{m-1} respectively. The general rule is that program is that program i is stored on tape Ti mod m on any given tape the programs are stored in increasing ordered of their length.

2.3 KNAPSACK PROBLEM

The Knapsack problem calls for selecting a subset of the objects and hence fit the subset paradigm. We are given n objects and a knapsack or a bag. Object 'i' has a weight w, and a knapsack has a capacity 'm'. If the fraction x_i, $0 \le x_i \le 1$ of object i is placed into the knapsack then a profit of $p_i x_i$ is earned.

The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Since the knapsack capacity is m, we require the total weight of all chosen objects to be at most m formally the problem can be stated as:

$$\text{Maximize} \quad \sum_{1 \le i \le n} p_i x_i$$

$$\text{Subject to} \quad \sum_{1 \le i \le n} w_i x_i \le m$$

And $0 \le x_i \le 1$, $1 \le i \le n$

In addition to selecting a subset the problem also involves the selection of an x_i for each object. Some of the Greedy strategies to obtain feasible solutions are :

1- Try to fill the knapsack by including the next object with the largest profit. If the object doesn's fit, then a fraction of its included to fill the

knapsack. Thus each time an object is included into a knapsack we obtain the largest possible increase in profit.

2- Alternatively considering the objects in order of decreasing profit values does not yield an optimal solution because even though the objective function value takes on large increases at each step, the no. of steps is few as the knapsack capacity is used up at a rapid stage.

3- The next attempt strikes to achieve a balance between the rate at which profit increases and the rate at which capacity is used. At each step we include that object which has the maximum profit per unit of capacity are considered in order of the ratio p_i/w_i

```
Algorithm Greedy Knapsack (m,n)
{
        For  i : = 1 to n do
        n [i] : = 0.0;
        u : = m;
for i : = 1 to n do
{
if (w[i] > u) then break;
n [i] : =1.0;
u : = u-w [i];
}
if (i ≤ n) then x [i] : = u / w [i];
```

2.4 MAKING CHANGE (MONEY)

First of all, we state a special case of the problem of making change, and then discuss the problem in its general form.

We, in India, have currency notes or coins of denominations of Rupees 1, 2, 5, 10, 20, 50, 100, 500 and 1000. Suppose a person has to pay an amount of Rs. 5896 after having decided to purchase an article. Then, *the **problem** is about how to pay the amount using **minimum** number of coins/notes.*

A simple, frequently and unconsciously used **algorithm** based on Greedy technique is that after having collected an amount A < 5896, choose a note of denomination D, which is s.t.

(i) A + D ≤ 5896 and

(ii) D is of maximum denomination for which (i) is satisfied, i.e., if E > D then A + E > 5896.

In general, the Change Problem may be stated as follow :

Let d_1, d_2,……d_k with $d_i > 0$ for i = 1, 2,…...,k, be the only coins that are available such that each coin with denomination di is available in sufficient quantity for the purpose of making payments. Further, let A, a positive integer, be the amount to be paid using the above-mentioned coins. The problem is to use the minimum number of coins for the purpose.

The problem with above mentioned algorithms based on greedy technique is that in some cases, it may either fail or may yield suboptimal solutions. In order to establish inadequacy of Greedy technique based algorithms, we consider the following two examples.

Example 1.2.1 : Let us assume a hypothetical situation in which we have supply of rupee-notes of denominations 5, 3 and 2 and we are to collect an amount of Rs. 9. Then using Greedy technique, first we choose a note of Rupees 5. Next we choose a 3-Rupee note to make a total amount of Rupees 8. But then according to Greedy technique, we can not go ahead in the direction of collecting Rupees 9. The failure of Greedy technique is because of the fact that there is a solution otherwise, as it is possible to make payment of Rupees 9 using notes of denominations of Rupees 5, Rupees 3 and Rupees 2, viz., 9 = 5+2+2.

2.5 MINIMUM SPANNING TREES

We apply Greedy technique to develop algorithms to solve some well known problems. First of all, we discuss the applications for finding minimum spanning tree for a given (undirected) graph.

Let G = (V, E). be undirected connected graph. A sub-graph $G' = (V', E')$ of G is a spanning tree of G if it is a tree.

The cost of a spanning tree of a weighted undirected graph is the sum of the costs (Weights) of the edges in the spanning tree. A minimum cost spanning tree is a spanning tree of least cost.

In the Greedy method, we construct an optimal solution in stages. At each stage, we make a decision that is the best decision (using some criterion) at this decision later, we make sure that the decision will result in a feasible solution.

For spanning trees, we use a least cost criterion. Our solution must satisfy the following constraints :

(i) We must use only edges within the graph.

(ii) We must use exactly n -1 edges.

(iii) We may not use edges that would produce a cycle.

Let us consider the connected weighted graph G given in Figure 2.5.1

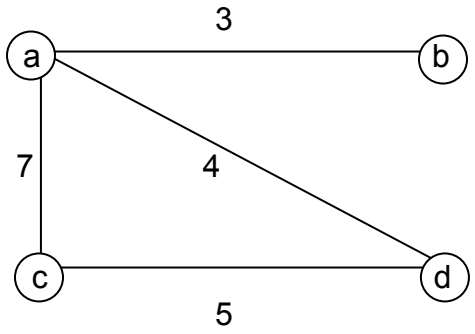
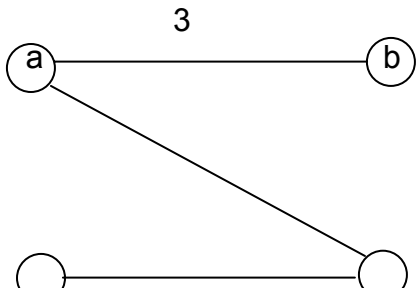


Figure 2.5.1

For the graph of Figure 2.5.1 given above, each of figure 2.5.2, figure 2.5.3 and figure 2.5.4 shows a spanning tree of G, of Weight 3+4+5 =12



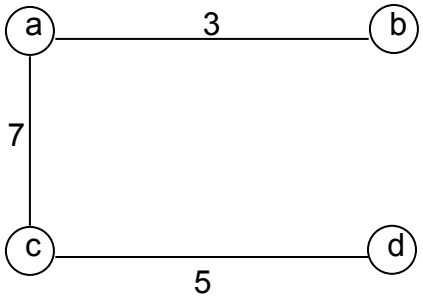
4

c d

5

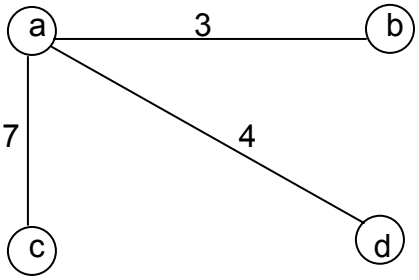
T_1

Figure 2.5.2



T_2

Figure 2.5.3



T_3

Figure 2.5.4

2.5.1. PRIM'S ALGORITHM

Prim's algorithm constructs the minimum cost spanning tree, T, That contains a single vertex. This vertex may be any of the vertices in the original graph. Next, we add a least cost edge (u, v) to T such that TU {(u, v)} is also a tree. We repeat this edge addition step until T Contains n -1 edges.

To make sure that the added edge does not form a cycle, at each step we choose the edge (u, v) such that exactly one of u or v is in T.

A formal description of Prim's algorithms T is the set of tree edges, and Tv is the set of tree vertices, that are currently in the tree.

Algorithms

T = { } ;

TV = { 0 } ; */ start with vertex 0 and no edge */ while (T contains fewer than n -1 edges)

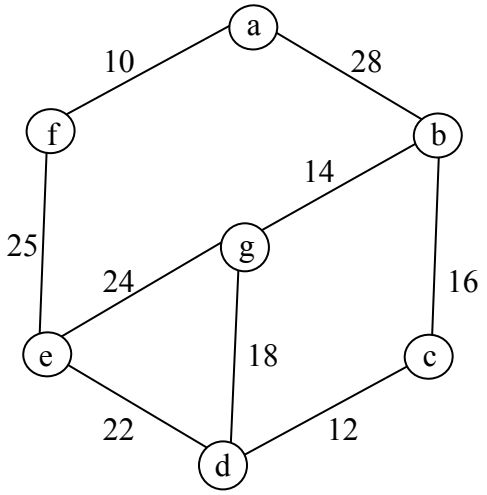
{

Let (u, v) be a least cost edge such that u € TV and v⁄€ TV ;

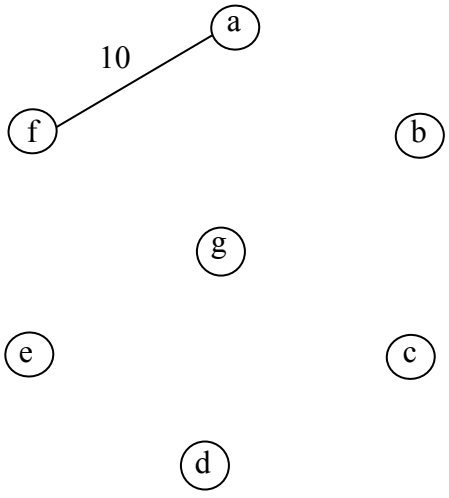
if (there is no such edge)

```
        break ;
add v to TV ;
add (u, v) to T ;
}
if (T contains fewer than n – 1 edges)
Display “no spanning tree”) ;
```

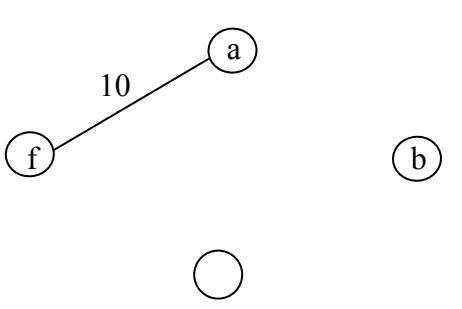
Example : we will construct a minimum cost spanning tree of the following graph.

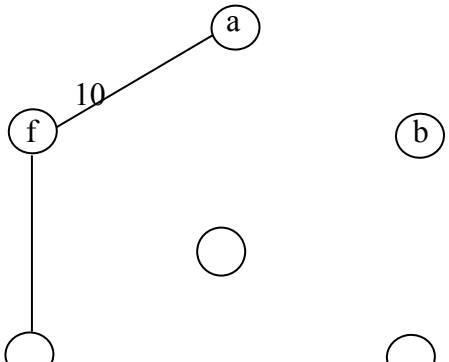


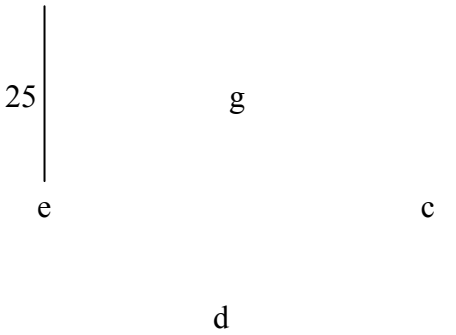
Solution:-



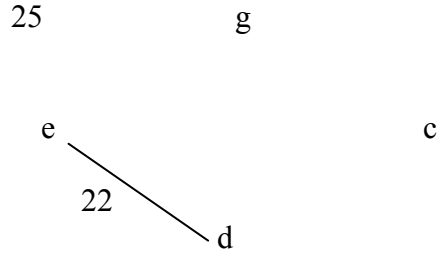
(i)



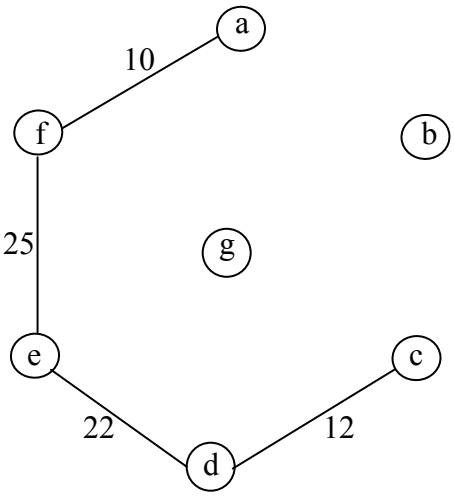




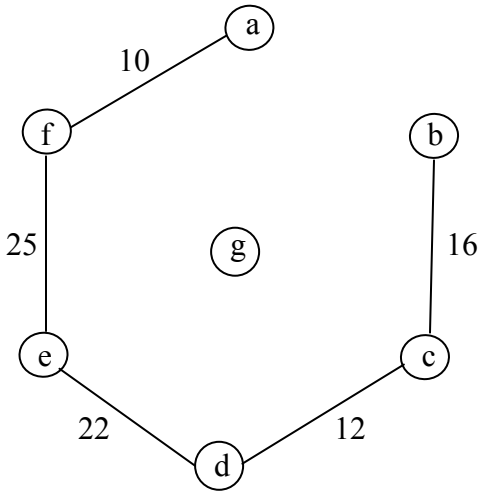
(ii)



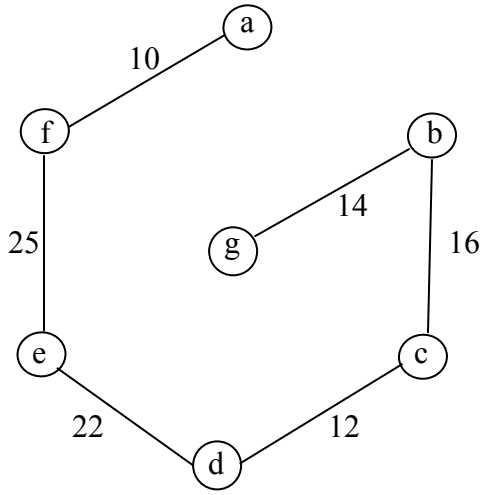
(iii)



(iv)



(v)



(vi)

Figure : Stages in Prim's Algorithms

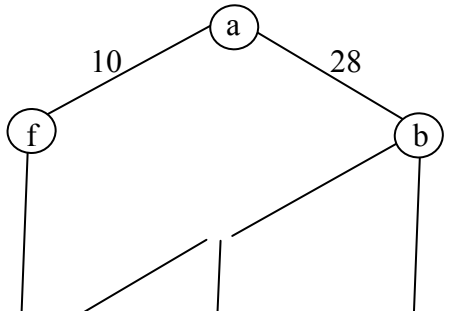
2.5.2 Kruskal's Algorithm

Kruskal's algorithm builds a minimum cost spanning tree T by adding edges to T one at a time. The algorithm selects the edges for inclusion in T in nondecreasing order of their cost. An edge is added to T if it does not form a cycle with the edges that are already in T. Since G is connected and has n > 0 vertices, exactly n – 1 edge will be selected for inclusion in T.

Algorithm:-

```
T = { };
While (T contains less than n-1 edges x & E is not empty)
{
choose a least cost edge (v, w) from E;
delete (v, w) from E;
if ((v, w) does not create a cycle in T)
                add (v, w) to T;
        else
                discard (v, w);
}
if (T contains fewer than n – 1 edges)
display ("no spanning tree");
```

Example:- Construct a minimum cost spanning tree of the following graph.



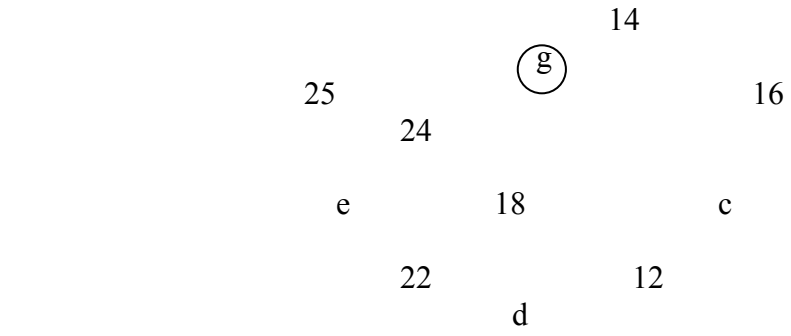
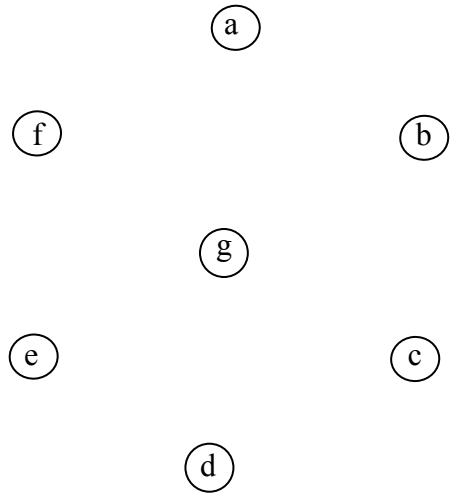
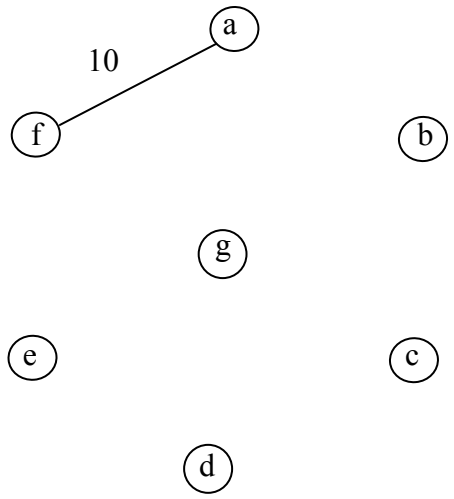


Figure : K

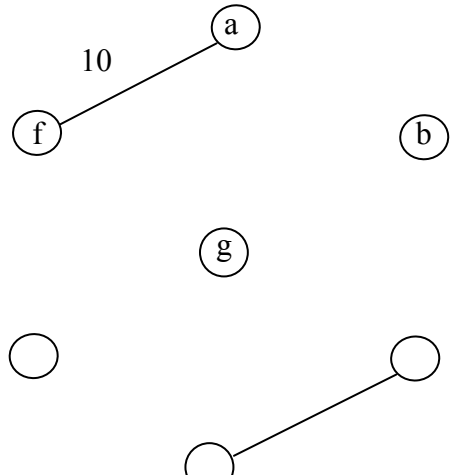
Solution:-

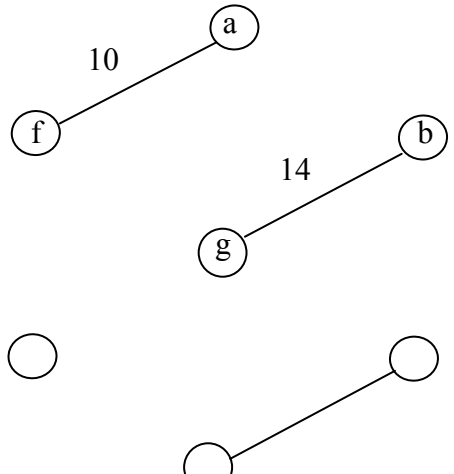


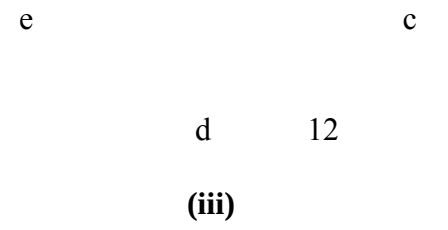
(i)



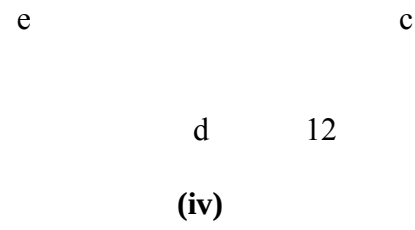
(ii)



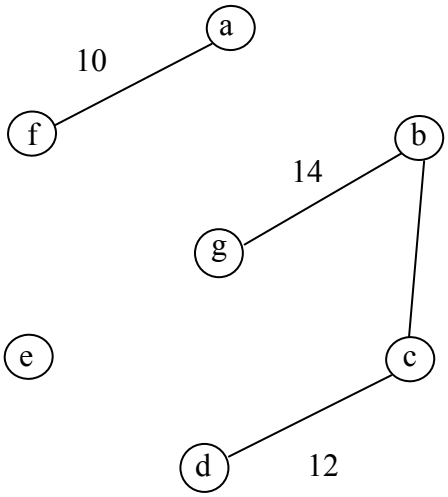




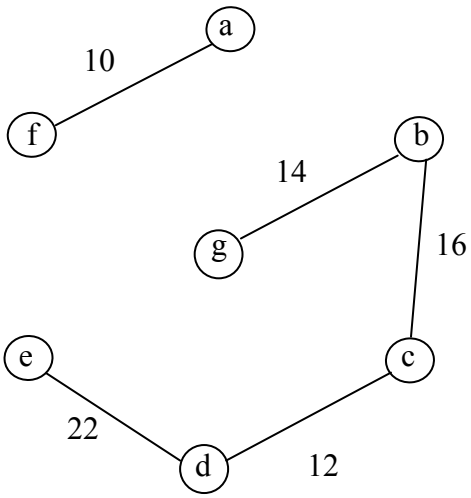
(iii)



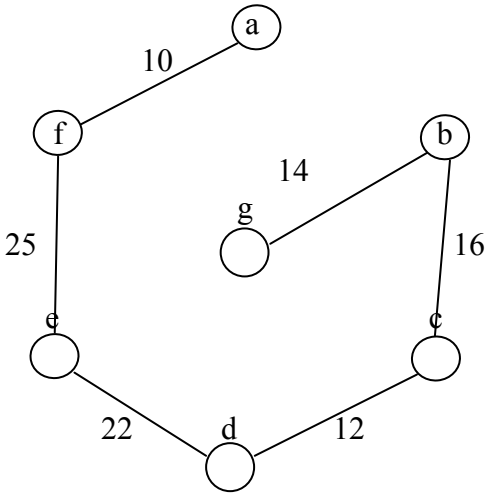
(iv)



(v)



(vi)



(vii)

Working produre for Kruskal's algorithm:-

Adge (a, f) is the first considered for inclusion. Since it cannot create a cycle, it is added to the tree. The result is the tree of figure K (ii). Similarly, edge (c, d) is considered next to add in the tree, and the result is shown in figure K (iii).

This process continues until the spanning tree has n – 1 edges (Figure K (vii)).

The cost of spanning tree is 99.

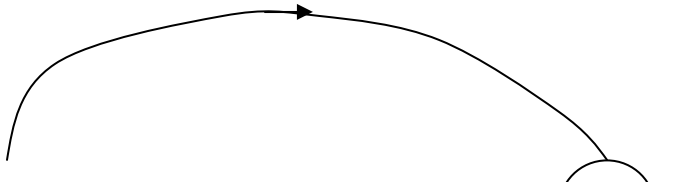
edge	Weight	Result	Figure
-	-	initial	Figure K (i)
(a, f)	10	added to tree	Figure K (ii)
(c, d)	12	added	Figure K (iii)
(b, g)	14	added	Figure K (iv)
(b, c)	16	added	Figure K (v)
(d, g)	18	discarded	----------
(d, e)	22	added	Figure K (vi)
(e, g)	24	discarded	----------
(e, f)	25	added	Figure K (vii)
(a, b)	28	not condered	----------

Summary of Kruskal's Algorithm

2.6 SINGLE SOURCE SHORTEST PATHS

Graphs may be used to represent the highway structure of country with vertices representing cities and edges representing sections of a highway. The edges usually assigned weights, which are the distance between the cities, connected by the edge. Such a graph can be used to identify the shortest path between two places A and B represented by two vertices in the graph. The length of a path is the sum of the weights of the edges of that path. The starting vertex of the path is called the source and the last vertex the destination.

In this problem we have to consider a directed graph G = (v, t) a weighing function c (e) for the edges of G and a source vertex v_0 to all remaining vertices of G.



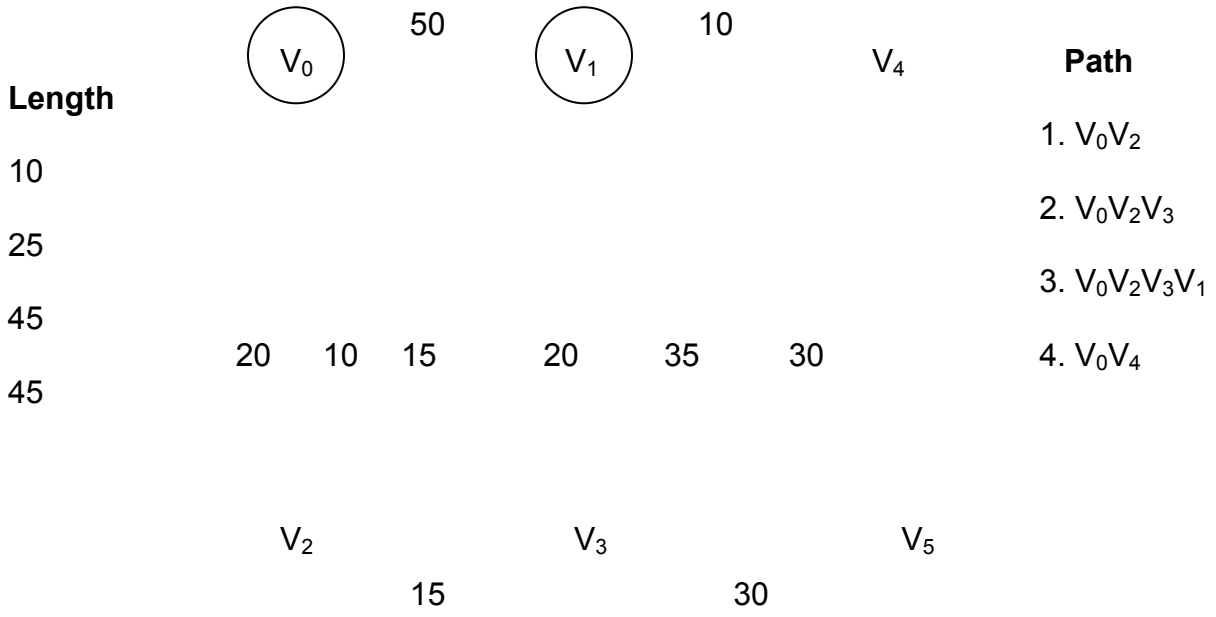


Figure 2.6 Graph and Shortest paths from V_0

The Greedy method builds up the shortest paths one by one. As an optimization measure we can use sum of lengths of all paths so far generated. For this each individual path must be of minimum length. That is, if we have already constructed i shortest paths then the next shortest minimum length path. The algorithm generates the shortest paths from v_0 to the remaining vertices to non-decreasing order of path length. First a shortest path to the nearest vertex is generated.

Then a shortest path to the second nearest vertex is generated and so on. Let S denote set of vertices (including v_0) to which the shortest path starting from v_0 going through only those vertices which are in S and ending at W, we can see that :- if the next shortest path is to vertex u, v_0 then the path begins at v_0 , end at u and goes through only those vertices which are in S this is because the paths are generated in non-decreasing order pf path lengths.

The destination of the next path generally must be that vertex u, which has the minimum distance DIST (u), among all vertices not in S.

Having selected a vertex u and generated the shortest v_0 to u path, vertex u becomes a number of S.

Algorithms shortest paths (v, cost, dist, n)

{

```
    for i=1 to n do
     {
        S [i] : = false ; dist [i] : =cost [v, i];
     }
S[v] : = true ; dist [v] : =0.0 ;
For  num : = 2 to n -1 do
      {
Choose u from among those vertices not in S such that dist [u] is minimum ;
S [u] : = true ;
For (each w adjacent to u with S [w] = false) do
If (dist) [w] > dist [u] + cost [u, w])
Then dist [w] := dist [n] + cost [u,w]
        }
}
```

2.6.1 DIJKSTRA'S ALGORITHM

Directed Graph :

So far we have discussed applications of Greedy technique to solve problems involving undirected graphs in which each edge (a, b) from a to b is also equally an edge from b to a. In other words, the two representations (a, b) and (b, a) are for the same edge. Undirected graphs represent symmetrical relations. For example, the relation of 'brother' between male members of, say a city is symmetric. However, in the same set, the relation of 'father' is not symmetric. Thus a general relation may be symmetric or asymmetric. A general relation is represented by a **directed** graph, in which the (directed) edge, also called an arc, (a, b) denotes an edge from a to b. However, the directed edge (a, b) is not the same as the directed edge (b, a). In the context of directed graphs, (b, a) denotes the edge from b to a. Next, we formally define a directed graph and then solve some problems, using Greedy technique, involving directed graphs.

Actually, the notation (a, b) is mathematics is used for ordered pair of the two elements viz., a and b in which a comes first and then b follows. And the ordered pair (b, a) denotes a different ordered set in which b comes first and then a follows.

// and Distance D (v) of any other vertex v is taken as ∞.

// iteratively distances of other vertices are modified taking into consideration the

// minimum distances of the various nodes from the node with most recently modified

// distance

D (s) ← 0

For each vertex v ≠ s do

D (v) ← ∞

// *Let Set-Remaining-Nodes be the set of all those nodes for which the final*

minimum

// distance is yet to be determined. Initially

Set-Remaining –Nodes ← V

while (Set-Remaining –Nodes ≠ Φ) **do**

begin

choose v £ Set-Remaining-Nodes such that D (v) is minimum

Set-Remaining-Nodes ← Set-Remaining-Nodes ~ {v}

For each node x £ Set-Remaining-Nodes such that w (v, x) ≠ ∞ do

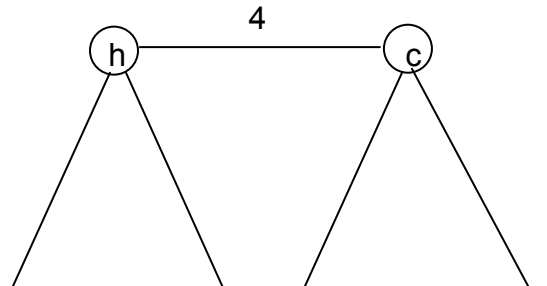
D(x) ← min {D (x), d (v) + w (v, x)}

end

Next, we consider an example to illustrate the **Dijkstra's Algorithm**

Example:

For the purpose, let us take the following graph in which, we take a as the source



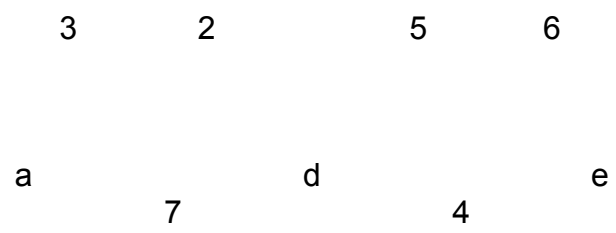


Figure : 2.7.2

Step	Additional node	S = Set-of-Remaining nodes	Distance from source of b, c, d, e
Initialization	**a**	**(b, c, d, e)**	**[3, ∞, 7, ∞]**
1	b	(c, d, e)	[3, 3 + 4, 3 + 2, ∞]
2	d	(c, e)	[3, 3+4, 3+2, 3+2+4]
3	c	(e)	[3, 7, 5, 9]

For minimum distance from a, the node b is directly accessed: the node c is accessedthrough b; the node d is accessed through b; and the node is accessed through b and d.

2.7 SUMMARY

In this unit, we have discussed the Greedy technique **the essence of which is : In the process of solving an optimization problem, and at subsequent stages, evaluate the costs/benefits of the various available alternatives for the next step.**

In this context, it may be noted that *the overall solution, yielded by choosing locally optimal steps, may not be optimal.* Next, well-known algorithms viz., Prim's and Kruskal's that use Greedy technique, to solving Single-Source-Shortest path problem, again using Greedy algorithms, is discussed.

2.8 KEYWORD

Greedy method : Apply for solving optimization problems x well known problems including shortest path problem.

Spanning Tree : suppose G = (V, E). be undirected connected graph. A sub-graph

$G' = (V', E')$ of G is a spanning tree of G if it is a tree.

SSSP : The problem of finding the shortest distances of each of the vertices of a

given weighted connected graph from some fixed vertex of the given graph.

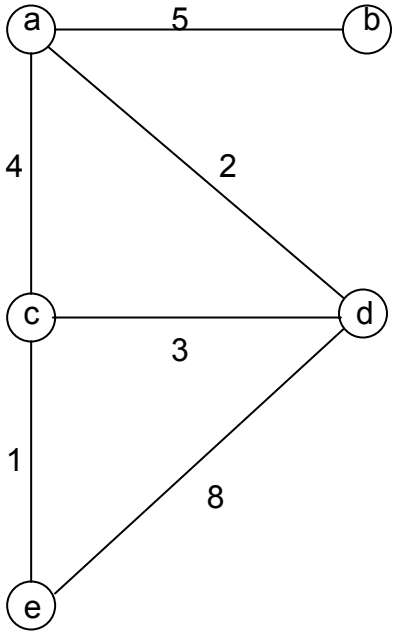
2.9 REVIEW QUESTIONS

1- Explain the strategies that can be applied for the Knapsack problem.

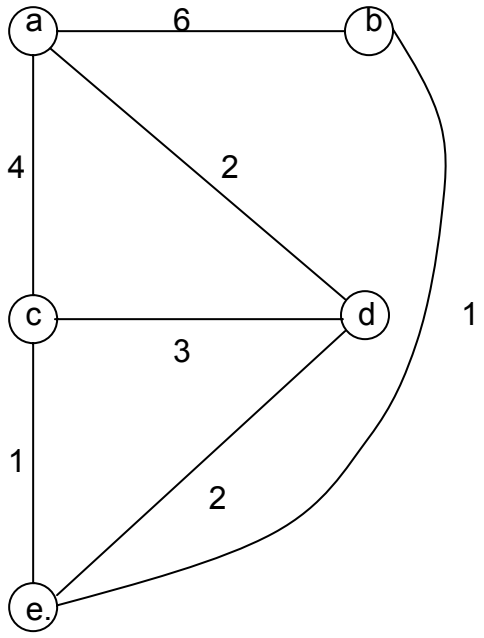
2- Find an optimal solution to the Knapsack instance n = 7, m = 15, (P_1, P_2, P_3,…P_7) = (10, 5, 15, 7, 6, 18, 3) and (w_1, w_2, …w_7) = (2, 3, 5, 7, 1, 4, 1).

3- Explain how the prime algorithms can be used to obtain a minimum-cost spanning tree.

4- Using Kruskal's algorithms, find a minimal spanning tree for the following g graph



5- Using Dijkstra's algorithm, find the minimum distances of all the nodes from node b which is taken as the source node, for the following graph.



2.10 FURTHER READINGS

4. Horowitz E and Sahni S., "Fundamental of Computer Algorithm" Galgotia Publications.
5. Aho A. V. Hoperoft, J.E. and Ullman, J.D., "Design and Analysis of Algorithm" Addison Wesley.
6. D. Harel., " Algorithmics : The Spirit of Computing" Addison Wesley.
7. Ravi Sethi, "Programming Languges- Concept and Constructs" Pearson Education, Asia, 1996.

Subject: Analysis and Design of Computer Algoriths
Paper Code: MCA 403 **Author: Sh. Ganesh Kumar**
Lesson: Dynamic Programming **Vetter: Jyoti**
Lesson No. : 03

STRUCTURE

3.1 INTRODUCTION

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions.

For the Knapsack problem the sequence of decisions decide the values of xi, $1 \leq i \leq n$.

an optimal sequence of decision maximizes the objective function.

One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequence. We could enumerate all decision sequences and then pick out the best but the time and space requirement may be prohibited.

Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optional sequences of decision is obtained by making explicit appeal to the principle of optimality.

Principle of optimality.

The principle of optimality states that an optimal sequences of decisions has the property that what ever the initial state and decision are the remaining decision must constitute optimal decision sequences with regards to the stage resulting from the first decision.

In the Greedy method only one decision sequence is generated. In dynamic programming several decision sequences are generated.

3.2 ALL PAIR SHORTEST PATH

In the all pairs shortest path problem we must find the shortest paths between all pairs of vertices v_i, v_j, $i \neq j$. We could solve this problem using shortest path with each of the vertices in V (G) as the source. Since G has n vertices and shortest path has a time complexity of O (n^2), the total time required would be O (n^3). However, we can obtain a conceptually simpler algorithm that works correctly even if some edges in *G* have negative weights. Although this algorithm still has a computing time of O (n^3), it has a smaller constant factor. The algorithm uses the dynamic programming method.

We represent the graph *G* by its cost adjacency matrix with cost [*i*] [*j*] = *0, i = j.* If the edge < *i, j* >, $i \neq j$ is not in *G,* we set cost [*i*] [*j*] to some sufficiently large number using the same restrictions discussed in the single source problem. Let A^k [*i*] [*j*] be the cost of the shortest path from i to j, using only those intermediate vertices with an index ≤ *k*. The cost of the shortest path from i to j is A^{n-1}[*i*] [*j*] as no vertex in *G* has an index greater than n – 1. Further, A^{-1}[*i*] [*j*] = *cost* [*i*] [*j*] since the only i to j paths allowed have no intermediate vertices on them.

The basic idea in the all pairs algorithm is to begin with the matrix A^{-1} and successively generate the matrices $A^0, A^1, A^2, \ldots, A^{n-1}$. If we have already generated A^{k-1}, then we may generate A^k by realizing that for any pair of vertices *i, j* one of the two rules below applies.

(1) The shortest path form *i* to *j* going through no vertex with index greater than *k* does not go through the vertex with index *k* and so its cost is A^{k-1} [*i*] [*j*].

(2) The shortest such path does go through vertex *k*. Such a path consists of a path from *i* to *k* followed by one from *k* to *j*. Neither of these goes through a vertex with index greater than *k -1*. Hence, their costs are A^{k-1}[*k*] [*j*]

The roles yield the following formulas for $A^k [i] [j]$:

$$A^k [i] [j] = min \{A^{k-1} [i] [j], A^{k-1} [i] [k] + \{A^{k-1} [k] [j], k \geq 0$$

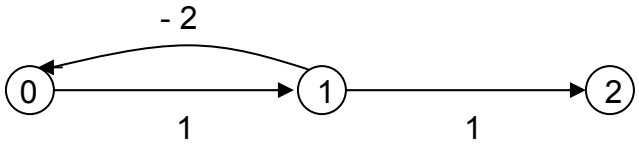
and

$$A^{-1} [i] [j] = \text{cost} [i] [j]$$

Example:- Figure below shows a digraph together with it's A^{-1} matrix. For this graph $A^1 [0][2] \neq \min \{A^1[0][2], A^0 [0][1] + A^0 [1][2]\} = 2$. Instead, $A^1 [0][2] = - \infty$ because the length of the path :

0 1 0 1 0 1 ……0 1 2

can be made arbitrarily small. This situation occurs because we have a cycle, 0, 1, 0, that has a negative length (- 1).



(a) Directed Graph

$$\begin{pmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{pmatrix}$$

(b) A^{-1}

Floyd–Warshall algorithm

the **Floyd–Warshall algorithm** is a graph analysis algorithm for finding shortest paths in a weighted, directed graph. A single execution of the algorithm will find the shortest paths between all pairs of vertices.

Algorithm

The Floyd-Warshall algorithm compares all possible paths through the graph between each pair of vertices. It is able to do this with only V^3 comparisons. This is remarkable considering that there may be up to V^2 edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is known to be optimal.

Consider a graph G with vertices V, each numbered 1 through N. Further consider a function shortestPath(i,j,k) that returns the shortest possible path from i to j using only vertices 1 through k as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only nodes 1 through $k + 1$.

There are two candidates for this path: either the true shortest path only uses nodes in the set $(1...k)$; or there exists some path that goes from i to $k + 1$, then from $k + 1$ to j that is better. We know that the best path from i to j that only uses nodes 1 through k is defined by shortestPath(i,j,k), and it is clear that if there were a better path from i to $k + 1$ to j, then the length of this path would be the concatenation of the shortest path from i to $k + 1$ (using vertices in $(1...k)$) and the shortest path from $k + 1$ to j (also using vertices in $(1...k)$).

Therefore, we can define $shortestPath(i,j,k)$ in terms of the following recursive formula:

This formula is the heart of Floyd Warshall. The algorithm works by first computing $shortestPath(i,j,1)$ for all (i,j) pairs, then using that to find $shortestPath(i,j,2)$ for all (i,j) pairs, etc. This process continues until k=n, and we have found the shortest path for all (i,j) pairs using any intermediate vertices.

Pseudocode

Conveniently, when calculating the kth case, one can overwrite the information saved from the computation of $k - 1$. This means the algorithm uses quadratic memory. Be careful to note the initialization conditions:

```
 1 /* Assume a function edgeCost(i,j) which returns the cost of the edge from
i to j
 2    (infinity if there is none).
```

```
 3    Also assume that n is the number of vertices and edgeCost(i,i)=0
 4 */
 5
 6 int path[][];
 7 /* A 2-dimensional matrix. At each step in the algorithm, path[i][j] is
the shortest path
 8    from i to j using intermediate vertices (1..k-1).  Each path[i][j] is
initialized to
 9    edgeCost(i,j) or infinity if there is no edge between i and j.
10 */
11
12 procedure FloydWarshall ()
13    for k: = 1 to n
14       for each (i,j) in {1,..,n}^2
15          path[i][j] = min ( path[i][j], path[i][k]+path[k][j] );
```

3.3 OPTIMAL BINARY SEARCH TREES.

A binary search tree is a tree, which has a finite set of nodes that is either empty of a root, and two disjoint binary trees called the left and right sub trees.

For a binary: -

1- All identifiers in the left sub-tree are less than the identifiers in the root node
2- All identifiers in the right sub-tree are greater than the identifiers in the root node
3- The left and right sub-trees are also binary search trees.

In a general situation we expect different identifiers to be searched for with different frequencies (probabilities). In addition we can expect unsuccessful searches also to made.

Let us assume that the given set of identifiers in $\{a_1, a_2, ...a_n\}$ with $a_1 < a_2 <....<a_n$ Let p (i) be the probability with which we search for a_i. Let q (i) be the probability that the identifier x being searched for is such that $a_i < x < a_i + 1$ where I is between o and n.

Then $\sum$ q (i) is the probability of an 0 i ≤ n

Unsuccessful search. Clearly

$\sum p(i) + \sum q(i) = 1$

$1 \le i \le n \qquad 0 \le i \le b$

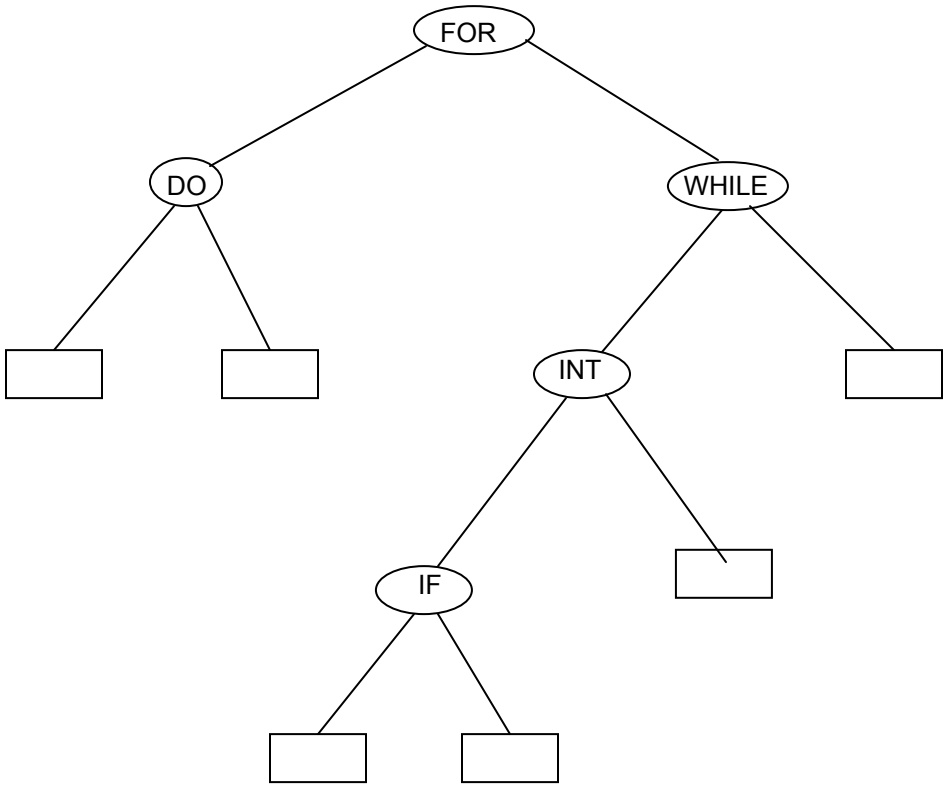


Figure 3.1 Binary Search Trees with external nodes

In obtaining a cost function for binary search trees its useful to add a fictitious node in place for every empty sub-tree in the search tree. Such nodes are called external node identifiers then the x will be exactly n internal nodes and n +1 fictitious or external nodes. Every internal node represents a point where an unsuccessful search may terminate.

If a successful search terminates at an internal node at level 'l' then 'l' iterations of the while loop are needed. Hence the expected cost contribution from the internal node for ai is

P (i) * level (ai).

The identifiers not in the binary tree can be partitioned into n +1 equivalence classes; Ei, $0 \le i \le n$. the class Eo contains all identifiers x, such that x, a_1, Ei contains all x, such that ai < x < ai < 1, $1 \le l \le n$. Ex contains x such that x >an. For identifiers in the same class Ei, the search terminates at the same external node.

```
Procedure Search (T, x, I)
{
        i ← T
        While i ≠ 0 do
        Case
        : x < IDENT (i);
                i ← LCHILD (i); // Search left sub-tree
        : x = IDENT (i);
                  return;
        : x > IDENT (i);
                i ← RCHILD (i);
end case
```

3.4 O/I KNAPSACK

A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables X_1, X_2, …X_n. A decision on variable X_i involves determining which of the values 0 or 1 is to be assigned to it.

Let us assume the decisions on the X_i, are made in the order X_n, X_{n-1}….x_1. Following a decision on X_n, we may be in one of the two possible states.

1. The capacity remaining in the Knapsack is m and no profit has occurred.
2. The capacity remaining is m-w_n and the profit P, has to be accured.

   It is clear that the remaining decisions x_{n-1}…x_1 must be optimum with respect to the problem state resulting from the decision on x_n.

Let Fj(y) be the value of an optimal solution to KNAP (1, j, y). Since the principle of optimality holds we obtain :

$F_n(n) = \max \{F_{n-1}(m), F_{n-1}(m-w_0) + P_n\}$

That can be solved with knowledge

That FO(y) = O for all y

$Fi(y) = -\infty, y < 0.$

When the Wi are integers we need to compute Fi(y) for integers y, 0, y, m. Since each Fi can be computed from F_{i-1} in 0(n) times it takes 0(m n) time to compute Fn.

The explicit 0(m n) computation of Fn may not be the most efficient computation so we consider an alternative.

$F_1(y)$ is an ascending step function that is there are a finite numbers of Y's, $0 = y_1, < y_2 <\ldots<y_k$. Such that $Fi(y_1), <Fi(y_2) <\ldots<Fi(Y_k)$.

We used an ordered set

$S^i = (F(y_i), y_i/1 \le j \le k\}$ to

Represent F_i (y)

S^i is pair (P, W) where P = Fi(y j). Notice that $S^0 = \{(0, 0)\}$ we cab compute S^{i+1} from S by the first computing

$S^i_1 = \{(P, W)/ (P-P_i, W-W_i) \in S^i\}$

S^{i+1} can be computed by merging the pairs S^i and S^i_1 together. If S^{i+1} contains two pairs (P_j , W_j) and (P_k, W_k) with the property that $P_j \le P_k$ and $W_j \le W_k$, then the pair P_j W_j can be discarded.

Discarding or perging rules are known as dominance rules. To obtain S^{i+1} the possibilities for $X_{i+1} = 0$ or $X_{i+1} = 1$, the resulting stage when $X_{i+1} = 0$ is same as for S^i. When $X_{i+1} = 1$ the resulting stage are obtained by adding $(P_{i+1,} W_{i+1})$ to each stage is S^i. We call these set of additional stage are S^i_1.

Algorithm DKP (p, w, n, m)

{

S^0: = {(0, 0)};

For i: = 1 to n-1 do

{$S1^{i-1}$: = {(p, w) / (p-q), w-w_1) $\in S^{i-1}$ and w ≤ m};

S^i: = merge Purge (S^{i-1}, $S1^{i-1}$);

}

(P x, Wx^1) : = last pair in S^{n-1};

(P y, W y) : = (P^1+P_n , W^1+W_n) where W^1 is the largest W in any pair in S^{n-1} such that $W + W_n \le m$;

If (P x > P y) then xn: = 0

Else x_n : = 1;

Trace back for (x_{n-1}, …x_1);

3.5 THE TRAVELING SALESPERSON PROBLEM

Travelling saleperson problem

The **Travelling Salesman Problem** (**TSP**) is a problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

In the theory of computational complexity, the decision version of TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSP problems. In other words, it is likely that the worst case running time for any algorithm for TSP increases exponentially with the number of cities, so even some instances with only hundreds of cities will take many CPU years to solve exactly.

As a graph problem

Symmetric TSP with four cities

TSP can be modelled as a graph: the graph's vertices correspond to cities and the graph's edges correspond to connections between cities, the length of an edge is the corresponding connection's distance. A TSP tour is now a Hamiltonian cycle in the graph, and an optimal TSP tour is a shortest Hamiltonian cycle.

Often, the underlying graph is a complete graph, so that every pair of vertices is connected by an edge. This is a useful simplifying step, because it makes it easy to find a solution, however bad, because the Hamiltonian cycle problem in complete graphs is

easy. Instances where not all cities are connected can be transformed into complete graphs by adding very long edges between these cities, edges that will not appear in the optimal tour.

Exact algorithms

The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute force search). The running time for this approach lies within a polynomial factor of $O(n!)$, the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. One of the earliest applications of dynamic programming is an algorithm that solves the problem in time $O(n^2 2^n)$

The dynamic programming solution requires exponential space. Using inclusion–exclusion, the problem can be solved in time within a polynomial factor of 2^n and polynomial space.[15]

Improving these time bounds seems to be difficult. For example, it is an open problem if there exists an exact algorithm for TSP that runs in time $O(1.9999^n)$

Other approaches include:

- Various branch-and-bound algorithms, which can be used to process TSPs containing 40-60 cities.
- Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities.
- Implementations of branch-and-bound and problem-specific cut generation; this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85,900 cities.

Special cases

Metric TSP

A very natural restriction of the TSP is to require that the distances between cities form a metric, i.e., they satisfy the triangle inequality. That is, for any 3 cities A, B and C, the distance between A and C must be at most the distance from A to B plus the distance from B to C. Most natural instances of TSP satisfy this constraint.

In this case, there is a constant-factor approximation algorithm due to Christofides that always finds a tour of length at most 1.5 times the shortest tour. In the next paragraphs, we explain a weaker (but simpler) algorithm which finds a tour of length at most twice the shortest tour.

The length of the minimum spanning tree of the network is a natural lower bound for the length of the optimal route. In the TSP with triangle inequality case it is possible to prove upper bounds in terms of the minimum spanning tree and design an algorithm that has a provable upper bound on the length of the route. The first published (and the simplest) example follows.

- Construct the minimum spanning tree.
- Duplicate all its edges. That is, wherever there is an edge from *u* to *v*, add a second edge from *u* to *v*. This gives us an Eulerian graph.
- Find a Eulerian cycle in it. Clearly, its length is twice the length of the tree.
- Convert the Eulerian cycle into the Hamiltonian one in the following way: walk along the Eulerian cycle, and each time you are about to come into an already visited vertex, skip it and try to go to the next one (along the Eulerian cycle).

It is easy to prove that the last step works. Moreover, thanks to the triangle inequality, each skipping at Step 4 is in fact a shortcut, i.e., the length of the cycle does not increase. Hence it gives us a TSP tour no more than twice as long as the optimal one.

Human performance on TSP

The TSP, in particular the Euclidean variant of the problem, has attracted the attention of researchers in cognitive psychology. It is observed that humans are able to produce

good quality solutions quickly. The first issue of the Journal of Problem Solving is devoted to the topic of human performance on TSP.

TSP path length for random pointset in a square

It is known that, for *N* points in a unit square, the TSP always has length at most proportional to the square root of *N*, and that randomly distributed point sets will have length that is (in expectation) at least proportional to the square root. However, the constants of proportionality, both for worst-case point sets and for random point sets, are not known.

Consider N stations randomly distributed in a 1 x 1 square with N>>1.

Lower bound

A lower bound of the shortest tour length is , obtained by assuming the mover stands on station j and always visits j's nearest as next.

A better lower bound is , obtained by assuming j's next is j's nearest, and j's previous is j's second nearest. This can be written as

A similar result obtained by dividing the points into equal disjoint sets and considering steps forwards and backwards from point j on the shortest path gives a lower bound

Upper bound

By applying simulated annealing method on samples of N=40000, computer shows an upper bound

, where 0.72 comes from boundary effect.

Because the actual solution is only the shortest path, for the purposes of programmatic search another upper bound is the length of any previously discovered approximation.

3.6 FLOW SHOP SCHEDULING

In a general flow – shop we may have n jobs each requiring n tasks. T_1, T_2, …T_{ni} is to be performed on processor Pj, $1 \le j \le n$ task T_{ji} is to be performed on processor P_i . The time required to complete the task T_{ji} is t_{ji}. A schedule for the n jobs is an assignment of tasks, to time the intervals on the processors. Task T_{ji} must be assigned to processor P_j .

No processor may have more than one task assigned to it in any time interval. Additionally for any job i the processing of task T_{ji} with j >I cannot be stared until task T_{ji} has been completed.

A non pre-emptive schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete. A schedule for which theis condition need not be true is called pre-emptive. The finish time Fi(s) of job I is the time at which all task of job I have been competed in schedule s is given by :

$F(s) = \max \{F_i(s)\}$

$1 \le i \le n$

the mean flow time MFT (s) is defined to be :

$MFT(s) = 1/n \sum /_{1si} \le _n Fi(s)$

An optimal finish time (OFT) schedule for a given set of jobs is a non pre-emptive schedule S for which F(s) is a minimum over all non-preemptive schedules S.

Two jobs have to be scheduled on three processors. The task time are given by matrix J

$$J = \begin{pmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{pmatrix}$$

The two possible schedules are

Time 0 2 5 6 10 12

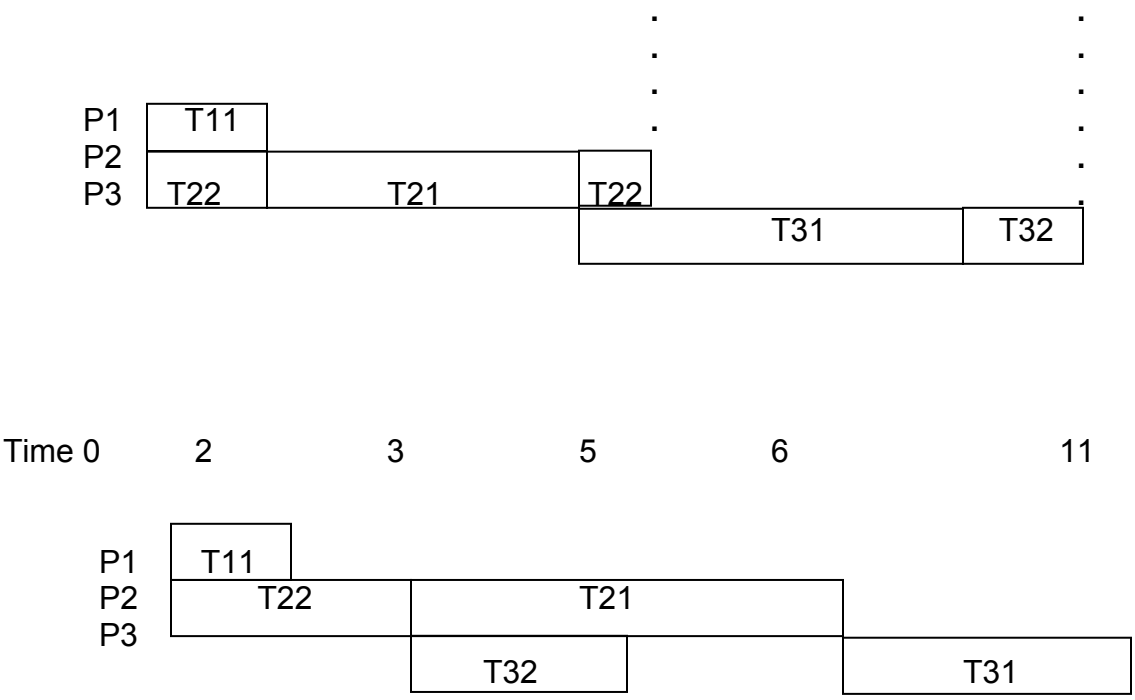


Figure : Two possible schedules.

3.7 SUMMARY

1- In order that Dynamic Programming technique is applicable in solving an optimisation problem, it is necessary that the principle of optimality is applicable to the problem domain.

2- **The Knapsack Problem:** We are given *n objects* and a Knopsack. For I = 1, 2,…., n, object i has a positive *weight* w_i and a positive *value* v_i. The Knapsack can carry a weight *not exceeding W.* The problem requires that the Knapsack is filled in a way that maximizes the value of the objects included in the knapsack.

Further, a special case of Knapsack problem may be obtained in which the objects may not be broken into pieces. In other words, either a whole object is to be included or it has to be excluded.

3.8 KEYWORDS

- **Dynamic Programming: -** It is that we should avoid calculating the same quantity more than once by keeping a table of known results for simple instances.
- **Principle of Optimality:** - States that components of a globally optimum solution must themselves be optimal.
- **Knapsack Problem:** - Can be obtained by making a sequence of decisions on n variables.
- **Flow Shop Scheduling:**- flow shop we may have n jobs each requiring n tasks, is to be performed on n processor.

3.9 REVIEW QUESTIONS.

1. Explain the terms:
    a. Principle of optimality.
    b. Pre-emptive scheduling.
    c. Mean flow time.
2. Explain how dynamic programming is used to obtain an optimal binary search tree.
3. Use an example and show a pre-emptive and a non-pre-emptive schedule.

3.10 FURTHER READING

8. Horowitz E and Sahni S., “Fundamental of Computer Algorithm” Galgotia Publications.
9. Aho A. V. Hoperoft, J.E. and Ullman, J.D., “Design and Analysis of Algorithm” Addison Wesley.
10. D. Harel., “ Algorithmics : The Spirit of Computing” Addison Wesley.
11. Ravi Sethi, “Programming Languges- Concept and Constructs” Pearson Education, Asia, 1996.

Subject: Analysis and Design of Computer Algoriths
Paper Code: MCA 403 **Author: Sh. Ganesh Kumar**
Lesson: Backtracking Method **Vetter: Ms Jyoti**
Lesson No. : 04

STRUCTURE

4.1 INTRODUCTION

In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. In many applications of the backtrack method, the desired solution is expressible as an n-triple (x,……, xn), where the xi are chosen from some finite set Si. After the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a criterion function P (x,…, xn). Some times it seeks all vectors that satisfy P.

Suppose mi is the size of the set Si. Then there are M = M1, M2…Mn, M-triples that are possible candidates for satisfying the function P. The brute force approach would be to form track algorithm has as its virtue the ability to yield the same answer with far fewer than 'm' trials. Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions P_i (x_1,……x_i) [called bounding functions] to test whether the vector being formed has any chance of success.

The major advantage of the method is this: if is realized that the partial vector (x_1, x_2, ….x_i) can in no way lead to an optimal solution, then Mi +1 ……m_n possible test vectors can be ignored entirely.

Many of the problems solved using backtracking requires that all the solutions satisfy a complex set of constraints can be divided into two categories: **explicit and implicit.**

Explicit constraints are rules that restrict each x_i to take on values only from a given set. They depend on the particular instance I of the problem being solved. All types that satisfy the explicit constraints define a possible solution space for I.

The implicit constraints are rules that determine which of the triple in criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other.

Backtracking algorithm determine problem solutions by systematically searching the solution space for the given for the given problem instance. This search is facilitated by using a **tree organization** for the solution space.

```
Algorithm Backtrack (k)
{
  for leach x [k] E t (x[1], …….., x [k-1])  do
        {
              if (Bk (x [1], x [2], …….., x [k]) ≠ 0) then
                    {
if (x[1], x [2], ……x [k] is a path to an answer node) then write (x [1:k];
if (k < n) then Backtrack (k = 1);
              }
        }
}
```

Recursive backtracking algorithm. An iterative version of the backtrack algorithm is a follow:

```
Algorithm I Back track (n)
{
   K : = 1;
    while (K ≠ 0)  do
     {
        if (three remains an untried
```

```
                x [K] ET (x [1], x [2]…….x [K-1])
                and Bk (x [1], …….x [K] is true) then
{
        if (x [1], ….x [k] is a path to an answer node) then write (x [1 :kj);
        K : = K + 1;
        }
        else K : = K-1;
    }
}
```

Each node in this tree defines a **problem state.** All paths from the root to other nodes define the state space of the problem. **Solution states** are those problem states 's' for which the path from the root to s defines a triple in the solution space. **Answer states** are those solution states 's' for which the path from the root to s defines a triple that is a member of the set of solutions of the problem. The tree organization of the solution space is referred to as the **state space tree.**

State space tree organizations that are independent of the problem instance being solved are called as static frees. Tree organizations that are problem instance dependant are called dynamic trees.

Beginning with the root all other nodes are generated. Anode which has been generated & all of whose children have not yet been generated is called a live node. Alive node whose children are currently being generated is called E-nod (node being expanded).

As a new child c or the current E-node R is generated, this child will become the new E-node. Then R will become the E-node again when the subtree c has been fully explored. This is the depth first generation of the problem states. Depth first node generation with bounding functions is called backtracking.

4.2 THE 8-QUEENS PROBLEM

The 8-Queens problem is a classic combinatorial problem to place eight queens or an 8 x 8 chessboard so that no two "attack", that is, so that no two of them are on the

same row, column or diagonal. All solutions to the same 8-queenss problem can be represented as 8-types (x_1, x_2……x_8) where x_i is the column on which queen i is placed.

We generalize the problem for an n x n chessboard and try to find all ways to place n non-attacking queens. We can let (x_1, x_2, …x_n) represent a solution in which x_i is the column of the i th row where the i th queen is placed. The x_i's will all be distinct since no two queens can be placed in the same column.

Considering the chessboard squares being numbered as the indices of the two-dimensional array a[1:n, 1:n] then every element on the same diagonal that runs from upper left to lower right has the same row-column value. Suppose two queens are placed at positions (i, j) and (k, l). Then they are on the same diagonal only if

$$i - j = k - l \quad \text{or} \quad i + j = k + l$$

The first implies

$$j - l = i - k$$

The second implies

$$j - l = k - l$$

Therefore two queens lie on the same diagonal if and only if / j – l / = / i – k /.

Place (k, i) returns a Boolean value that is true if the k th queen can be placed on column i. It tests both whether i is distinct from all previous values x [1],…x [k-1] and whether there is no other queen on the same diagonal. Its computing time is 0(k-1).

```
Algorithm place (k, i)
        {
        for j : = 1 to  k-1 do
        if ((x[j] – i) = Abs (j – k)))
                then return false
        return true;
        }
        Algorithm N Queens (k, n)
        {
        for i : = 1 to n do
        {
        if place (k, i) then
        {
        x [k] : = i;
```

```
if (k = n) then write (x[1 :n]) ;
else N Queens (k + 1);
        }
    }
}
```

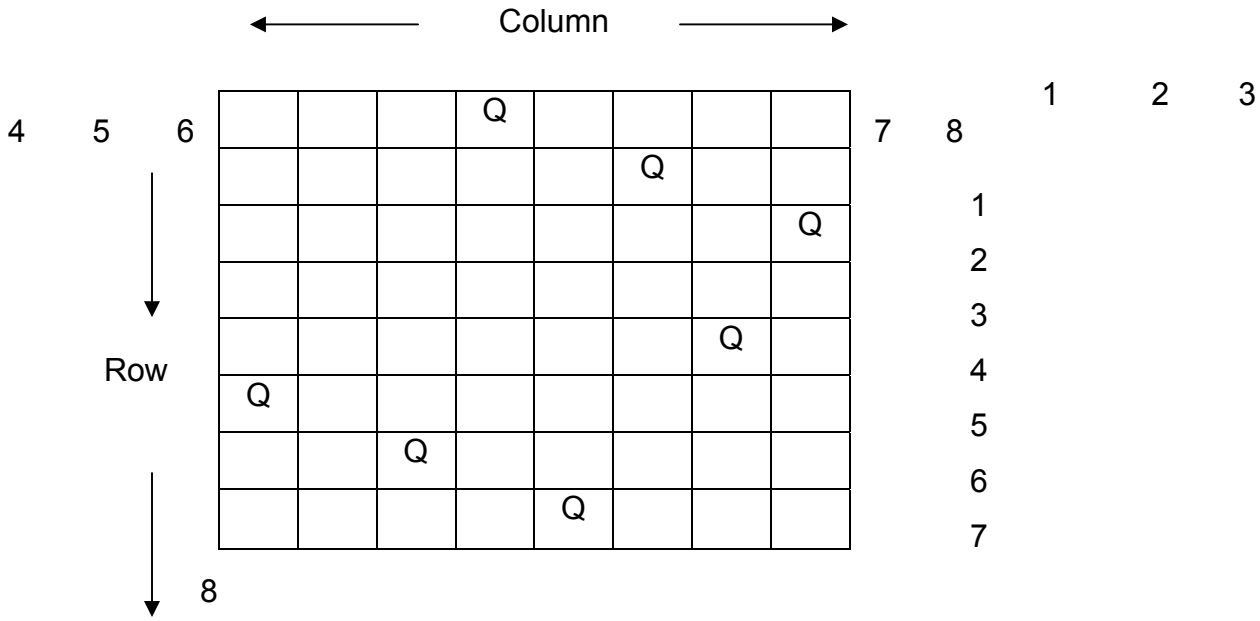


Figure: 4.2 One solution to the 8-queens problem.

The 8 Queens Problem

We will solve the 8 Queens problem using BACKTRACKING. Actually we will solve the n Queens problem.

- What is BACKTRACKING?
- What is the 8 Queens problem?
- What is the n Queens problem?

Backtracking

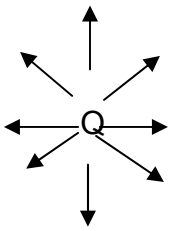
Backtracking is kind of solving a problem by trial and error. However, it is a well organized trial and error. We make sure that we never try the same thing twice. We

also make sure that if the problem is finite we will eventually try all possibilities (assuming there is enough computing power to try all possibilities).

The 8 Queens Problem:

Given is a chess board. A chess board has 8x8 fields. Is it possible to place 8 queens on this board, so that no two queens can attack each other?

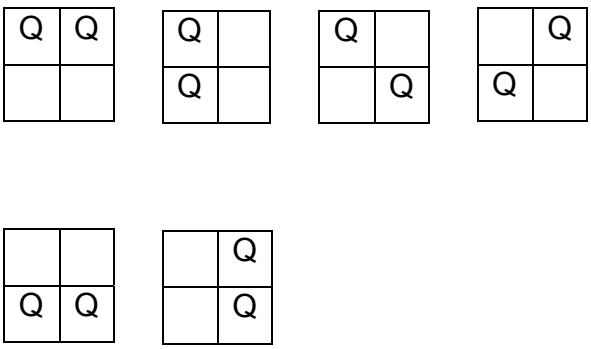
NOTES: A queen can attack horizontally, vertically, and on both diagonals, so it is pretty hard to place several queens on one board so that they don't attack each other.



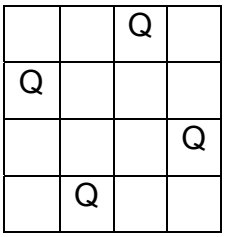
The **n** Queens problem:

Given is a board of **n** by **n** squares. Is it possible to place **n** queens (that behave exactly like chess queens) on this board,
without having any one of them attack any other queen?

Example: 2 Queens problem is not solvable.



Example 2: The 4-queens problem is solvable:



Basic idea of solution:

· Start with one queen in the first column, first row.
· Start with another queen in the second column, first row.
· Go down with the second queen until you reach a permissible situation.

· Advance to the next column, first row, and do the same thing.
· If you cannot find a permissible situation in one column and reach the bottom of it, then you have to go back to the
previous column and move one position down there. (This is the backtracking step.)

· If you reach a permissible situation in the last column of the board, then the problem is solved.
· If you have to backtrack BEFORE the first column, then the problem is not solvable.

4.3 SUM OF SUBSET

Suppose we are given n distance positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sum are m. This problem can be formulated using either the fixed or variable sized tuples.

In the variable tuple size formulation, the edges are labeled such that an edge from a level i nodes to level i + 1 nodes are labeled with value of x_i, which is either zero or one.

We consider a back tracking solution using the fixed tuple size strategy in this case the element x_i of the solution vector is either or zero depending on whether the weight w_i is included or not : The children of any node at level i are the left child corresponding to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding function is Bk (x_1…..x_k) = true iff

$$\sum_{i=1}^{K} W_i X_i + \sum_{i=1+k+1}^{n} W_i \geq m$$

The bounding function can be strengthened if we assume the w_i's are initially in non-decreasing order. In this case $x_1 \ldots x_k$ can not lead to an answer node if

$$\sum_{i=1}^{k} W_i X_i + W_{k+1} > m$$

The bounding functions we use are therefore B_k ($x_1 \ldots x_k$) = true iff $\sum_{i=1}^{K} W_i X_i + \sum_{i=1+k+1}^{n} W_i \geq m$

$$\sum_{i=1}^{k} W_i X_i + W_{k+1} \leq m$$

Algorithms sum of sub (s, k, r)

```
{
    x [k] : = 1;
    if (s + w [k] = m) then write (x[1 : k]);
    else if (s + w[l] + w[k+ 1] ≤ m);
    then sum of sub (s + w[k], k + 1, r-w[k]);
    if ((s + r-w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
    {
```

```
x [k] : = 0;
sum of sub (s, k+1, r-w [k]);
        }
}
```

4.4 KNAPSACK PROBLEM

Given 'n' positive weights w_i n positive profits p_i, and a positive number m that is the Knapsack capacity, this problem calls for crossing a subset of the weights such that.

$\sum W_i X_i \leq$ and $\sum p_i x_i$ maximized

$1 \leq i \leq n$ $1 \leq i \leq n$

The 2^ distinct ways to assign zero or one values to the x's

Backtracking algorithms for the problem use bounding functions that are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so fast, then that live node can be killed. Using the fixed tuple size formulation. If at node Z the values of xi, 1 ≤I ≤ k, have already been determined, then an upper bound for Z can be obtained by relaxing the requirement xi = 0 or 1 to 0 ≤ xi ≤ 1 for k + 1 < I < n and using the Greedy algorithm.

Function bound (ip, iw, k) determines an upper bound on the best solution obtainable by expanding any node Z at level k+1 of the state space tree. The object weights and profits are w[I].

```
Algorithm Bound (cp, cw, k)
{
  b : cp; c: = cw;
    for i : = k+1 to n do
      {   c : = c+w [i]
            if (c<m) then b : = b+p [ i ];
            else return b + (1-(c-m) / w [i]) * p [ i ];
      }
      Return b:
```

}

From Bound it follow that the bound for a feasible left child of a node Z is the same as that for Z. Hence the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of a node the resulting algorithm is Bknap. Initially set Fp : = -1;

This algorithm is invoked as

Bknap (1, 0, 0);

When fp ≠ -1 , x [i], 1 ≤ l ≤ n, is such that $\sum_{i=1}^{n}$ p[l] x [l]

```
Algorithm Bknap (k, co, cw)
{
if (cw + w [k] ≤ m ) then
{
y [k] : = 1;
if (k<x) then Bknap (k+1, cp+p[k], cw +w [k]);
if ((cp + p[k]> fp) and (k=n)) then
{
fp := cp +p[k]; fw : = cw +w [k];
for j : = 1 to k do x [ j ] : =y [ j ];
        }
}
if (Bound (cp, cw, k) ≥ fp) then
{
Y[k] : = 0; if (j < n) then
Bknap (k+1, cp, cw);
if ((cp> fp) and (k = n)) then
{
fp : = cp; fw : = cw;
for j : = 1 to k do
x [ j ] : = y [ j ];
        }
    }
}
```

The path y [i], 1 ≤ i ≤ k is the path to the current node. The current weight

$$cw = \sum_{i=1}^{k-i} w[i] * y[i]$$

and $$cp = \sum_{i=1}^{k-i} p[i] * y[i]$$

4.5 GRAPH COLORING

Let G be a graph and m be a given positive integer. The m- colorability decision is to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. If d is the degree of the given graph, then it can be colored with d+ 1 color. The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is called as the chromatic number of the graph.

A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.

Suppose we represent a graph by its adjacency matrix G [1:n, 1:n] where G [I, j] = 1 if (I, j) is an edge of G and G [I, j] = 0 otherwise. The colors are represented by integers 1, 2, …..m and the solutions are given by the n tuple (x_1,…x_n) where xi is the color of node i. The state space tree used is a tree of degree m and height n+1. Each node at level i have m children corresponding to the m possible arrangements n + 1 is leaf node.

Function next value produces the possible colors for x_k after x_1 through x_{k-1} have been defined. An upper bound on the computing time for n coloring can be arrived at by noticing that

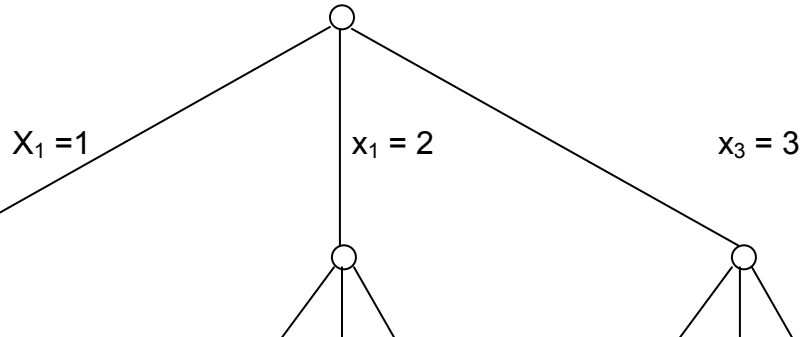
the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node 0(m,n)

time is spent by next value to determine the children corresponding to legal colorings. Hence the total time is bounded by

$$\sum_{i=0}^{n-1} m^{i+1} n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2) / (m-1)$$

= 0 ($n\ m^n$)

```
Algorithm m coloring (K)
{
Repeat
{
        Next value (K);
        if (x [K] = 0 then return;
        if (K = n) then
        write (x [1 : n])
        else nColoring (K+1);
} until (false);
}
Algorithm next value (k)
{
Repeat
        {x [k] : = (x [k] + 1) nod (m+1);
        if (x [k] = 0) then return
        for j : = 1 to n do
{
        if (( G [k, j] ≠ 0) and (x [k] = x [ j ]  ))
                then break;
        }
        if  (j = n+1) then return;
        } until (false);
        }
```



$X_2 = 1$

3

2

$X_3 = 1$

Figure : State Space free for nColoring when n = 3 and m = 3

4.6 SUMMARY

This unit discusses some backtracking techniques for finding one vector that maximizes or minimizes or satisfies a criterion function.

The technique is illustrated with examples of its applications to solving problems of the 8 queens problem. Sum of subsets, Knapsack problem and graph coloring of a given information.

4.7 KEYWORDS

8-Queens problem: the problem for an n x n chessboard and try to find all ways to place n non-attacking queens.

Sum of Subset: n distance positive numbers to find all combinations of these numbers whose sum are m.

Graph Coloring: If d is the degree of the given graph, then it can be colored with d+ 1 color.

4.8 REVIEW QUESTIONS

1. Explain the terms:
   (a) Implicit constraints & explicit constraints.

(b) Solution space

(c) Static and Dynamic trees

2. Explain the backtracking technique as applied to the N Queens problem where N = 4. Draw the tree organization.
3. What is the chromatic number? How is the nColoring algorithm used to assign colors to the various nodes?
4. What is the bounding function that is used for sum of subsets problem?

4.9 FURTHER READINGS

12. Horowitz E and Sahni S., “Fundamental of Computer Algorithm” Galgotia Publications.
13. Aho A. V. Hoperoft, J.E. and Ullman, J.D., “Design and Analysis of Algorithm” Addison Wesley.
14. D. Harel., “ Algorithmics : The Spirit of Computing” Addison Wesley.
15. Ravi Sethi, “Programming Languges- Concept and Constructs” Pearson Education, Asia, 1996.

Subject: Analysis and Design of Computer Algoriths
Paper Code: MCA 403 **Author: Sh. Ganesh Kumar**
Lesson: Branch And Bound **Vetter: Ms Jyoti**
Lesson No. : 05

STRUCTURE

5.1 INTRODUCTION

In branch and bound method we have state space search methods in which all children of the E- node are generated before any other live node can become the E-node. In branch – and – bound terminology, a BFS like state space search will be called FIFO (First in First Out) search as the list of live nodes is a queue. A D – search – like state space search will be called LIFO (Last in First out) Search as the list of live nodes is a last-in-first-out list (or stack). Boundary functions are used to help avoid the generation of sub trees that do not contain an answer node.

LEAST COST (LC) SEARCH:-

The search for an answer node can often be speeded by using an “intelligent” ranking function i < > for live nodes. The next E- node is selected on the basis of this ranking function. The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node. For any node x, this cost could be (i) the number of nodes in the sub tree x that need to be generated before an answer node is generated or, more simply (2) the number of levels the nearest answer node is from x. if cost measure 1 is used, then the search would always generate the minimum no. of nodes every branch and bound type algorithm must generate. If cost measure 2 is used, then the only nodes to become E – nodes are nodes on the path from the root to the nearest answer node.

Search algorithms usually rank nodes only on the basis of an estimate g (.) of their cost. Leg g (x) be an estimate of the additional effort needed to reach an answer node from x. node x is assigned a rank using a function i (.) such that I (x) = f (n (x)) + g (x), where n (x) is the cost of reaching x from the root and f (.) is any non decreasing function.

A search strategy that uses a cost function i (x) = f (n (x)) + g (x) to select the next E-node a live node with least i (.). Hence such a search strategy is called an LC search.

BOUNDING:-

Each answer node x has a cost c (x) associated with it and that a minimum – cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC.

A cost function $\hat{c}$ (.) such that $\hat{c}$ (x) ≤ c (x) is used to provide lower bounds on solutions obtainable from any node x. if upper is an upper bound on the cost of a minimum –cost solution, then all live nodes x with $\hat{c}$ (x) > upper may be killed as all answer nodes reachable from x have cost c (x) ≥ i (x) > upper. The starting value for upper can be set to ∞. Clearly so long as the initial value for upper is no less than the cost of a minimum cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum cost is found, the value of upper can be updated.

```
List ode = record {
    List node * next, * parent; float cost;
```

```
    }
Algorithm < c search (t)
{
   if * t is an answer node then output * t and return;
E : = t;
Initialize the list of live nodes to be empty;
Repeat
{
  for each child x of E do
    {
    if x is an answer node then output the path from x to t and return;
    Add (x);
    (x → parent) : = E;
}
    if there are no more live nodes then
    {
       write ("No answer node");
    return;
    }
    E : = Least ( );
    } until (false);
    }
```

5.2 I/O KNAPSACK PROBLEM

The branch – and – bound technique deals with only minimization problems. The knapsack however is an maximization problem. This difficulty is overcome by replacing the objective function $\sum$ pi xi by the function - $\sum$ pi xi. Clearly, $\sum$ pi xi is maximized if f - $\sum$ pi xi is minimized. The modified knapsack problem is hence

$$\text{minimize} - \sum_{i=1} p_i x_i$$

$$\text{subject to} \sum w_i x_i \leq m$$

i = 1

xi = 0 or 1, 1 ≤ n

Every leaf node in the state space tree representing an assignment for which $\sum_{1 \le i \le n} W_i X_i \le n$ is an answer node. All other leaf nodes are infeasible. For a minimum cost answer node to correspond to any optimal solution, we need to define $C(x) = -\sum_{1 \le i \le n} P_i X_i$ for every answer node x. The cost C(x) = for infeasible leaf nodes. For non leaf nodes, ((x) is recursively defined to be min {C (1 child (x)), c (r child (x))}

We need 2 functions i (x) such that i (x) such that i (x) ≤ c (x) ≤ u (x) for every node x the cost i (.) and (.) satisfying this requirements may be obtained as follows, Let x be a node at level j, 1 ≤ j ≤ n +21. at node x assignments have already been made to x_i 1 ≤ i ≤ j. the cost of these-

assignments is $-\sum p_i x_i$. So c (x) ≤ $-\sum p_i x_i$ and we may use $u(x) = -\sum_{1 \le i \le j} p_i x_i$. if $q = -\sum_{1 \le i \le j} p_i x_i$,

Then an improved upper bound function u (x) is u (x) = U Bound ($q, \sum_{1 \le i \le j} w_i x_i$), j – 1, m).

```
Algorithm U bound (cp, cw, k, m)
{
    b: cp;  c: = cw;
    for i : = K+1 to n do
{ if (c + w [ i ] ≤ m) then
{ e : = c + w [ i ];
b : = b –p [ i ]
}
}
Return b;
}
```

5.3 TRAVELLING SALESPERSON

Let G = (V, E) be a directed graph defining an instance of the traveling sales-person problem. Let c_{ij} equal the cost of edge < i, j >, c_{ij} = α if < i, j > ∉ ……E, and let/V/ =n. We can assume that every tour starts and ends at vertex 1. The solution space S is given by S = {1, л, 1/ л is a permutation of (2, 3,…n)}. Then /S/ = (n – 1)! The size of S

can be reduced by restricting S so that $(1, i_1, i_2, \ldots i_{n-1}, 1) \in S$ if $\langle i_j, i_j+1\rangle$ EE, $0 \; j_{n-1}$, and $i_0 = i_n = 1$.

To use LCBB to search the the traveling sales person state space tree, we need to define a cost function C (.) and two other functions C (.) and u (.) such that $C(r) \leq u(r) \leq u(r)$ for all nodes r. The cost C (.) is such that the solution node with least C (.) corresponds to a shortest tour in G. One choice for C (.) is

$$C(A) = \begin{cases} \text{Length of tour defined by the path form root to A, if A if A is a} \\ \text{leaf cost of a minimum-cost leaf in the sub tree A, if A is not} \\ \text{a leaf.} \end{cases}$$

A simple C (.) such that $C(A) \leq c(A)$ for all A is obtained by defining C (A) to be the length of the path defined at node A. A row is said to be reduced iff it contains at least one zero and all remaining entries are non-negative. A matrix is reduced iff every row and column is reduced.

We can associate a reduced cost matrix with every node in the traveling salesperson state space tree. Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R,S) corresponds to including edge <i, j> in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows:

(1) Change all entries in row i and column j of A to α. This prevents the use of any more edges leaving vertex i or entering vertex j.

(2) Set A (j, i). to α. This prevents the use of any edge < j, i>.

(3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only α. Let the resulting matrix be B.

Let the resulting matrix be B. if r is the total amount subtracted in step (3) then C (s) = C (r) + A (i, j) +r for leaf nodes (.) = c () is easily computed as each leaf defines a unique tour. For the upper bound function u. we can use u (R) = α for all nodes R.

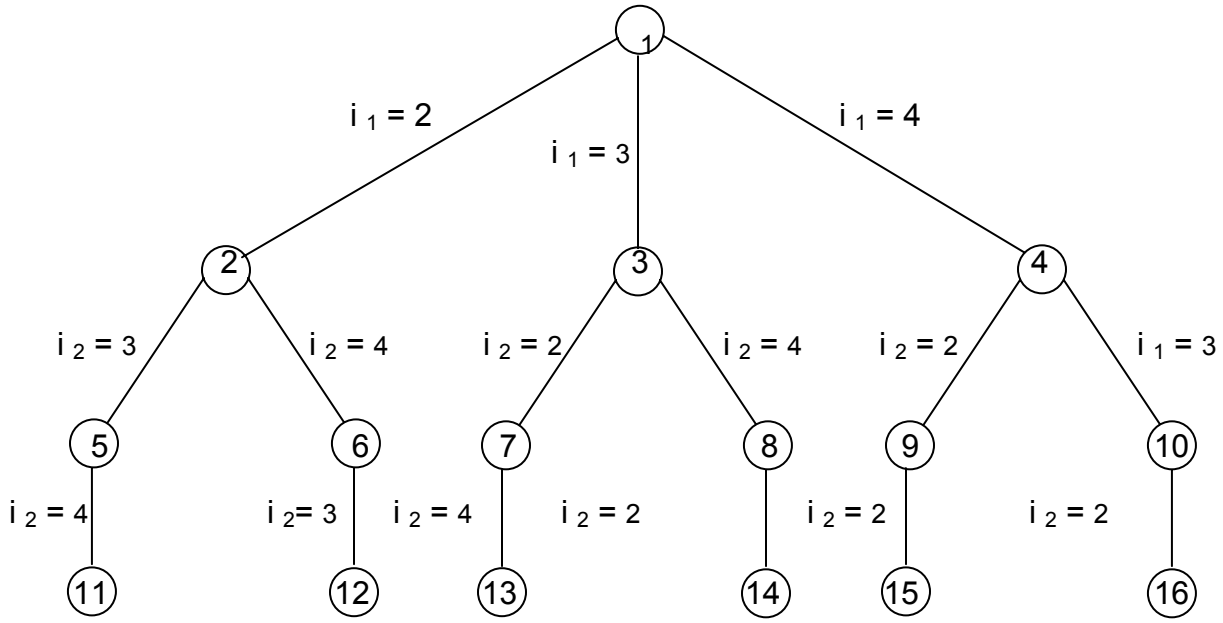


Figure: State space tree for the traveling sales person problem with n=4 i0 = is =1

20	20	30	10	11	∞	10	17	0	1
15	∞	16	4	2	12	∞	11	2	0
3	5	∞	2	4	0	3	∞	0	2
19	6	18	∞	3	15	3	12	∞	0
16	4	7	16	∞	11	0	0	12	∞

Costmatrix

Reduced cost matrix

L = 25

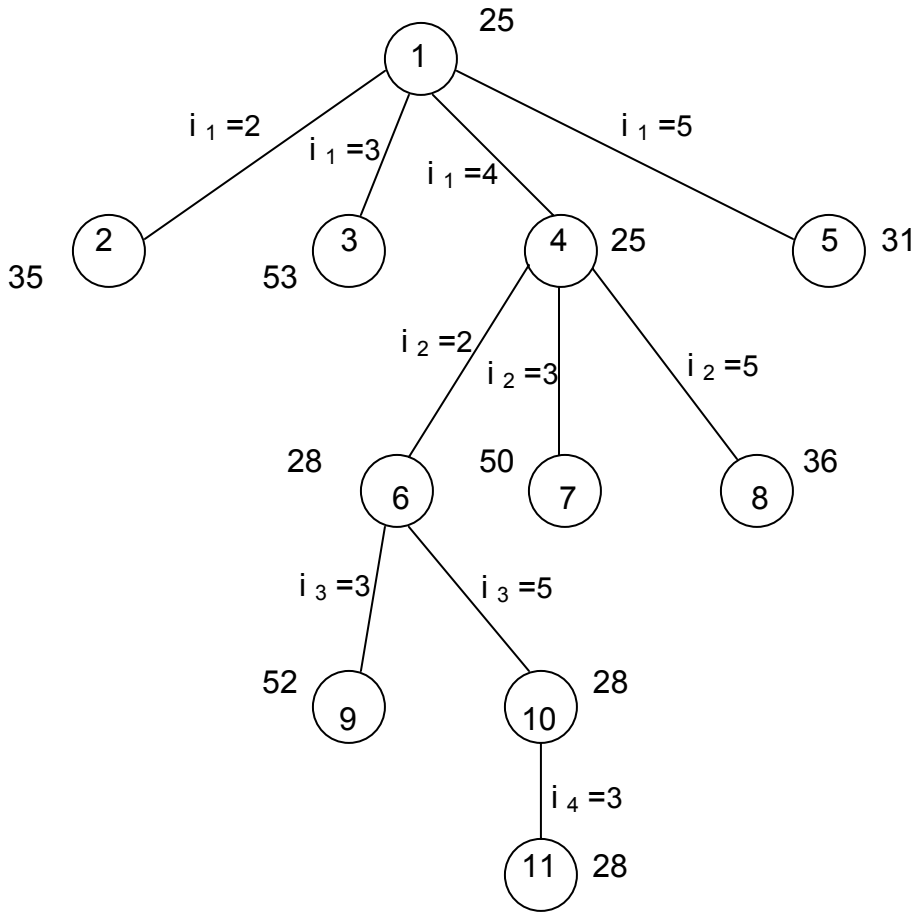


Figure: State Space Tree Generated by latest-lost-branch-bound number outside the node are C value.

5.4 LOWER BOUND THEORY

Oour main task is to obtain correct and efficient solutions. If two algorithms for solving the same problem were discovered and their times differed by an order of magnitude, then the one with the smaller order was generally regarded as superior. But still, we are left with the question: is there a faster method? The purpose of this chapter is to expose you to some techniques that have been used to establish that a given algorithm is the most efficient possible. The way that this is done is by discovering a function g(n) that is a lower bound on the time algorithm *any* algorithm must take to solve the given problem. If we take an

algorithm whose computing time is the same order as g(n) , then we know that asymptotically we can do no better.

f (n) - running time of some algorithm

g (n) - lower bound -- (Omega)

Def. $f(n) = \Omega(g(n))$

iff there exists positive constants c and n_o such that $f(n) \geq cg(n)$ for all n , $n \geq n_o$ (g(n) is a lower bound for f(n))

if $f(n) \geq O(g(n))$ and $f(n) = \Omega(g(n))$
then asymptotically, we cannot have any improvement.
Improvement in constant possible.

Deriving good lower bounds is often more difficult than devising efficient algorithms. Perhaps this is because a lower bound states a fact for *all* possible algorithms for solving a problem. Usually we cannot enumerate and analysize all these algorithms, so lower bound proofs are often hard to obtain

Trivial lower bounds:

Max of set of n numbers

have to look at all inputs, n-1 comparisons : f(n) is O(n) ; $\Omega(n)$

n x n matrix multiplicaton: $2n^2$ inputs , n^2 outputs
$\Omega(n^2)$

but this is lower than the best known algorithm

Best known algorithm requires f (n) = 0 $(n^{2+\epsilon})$

Improvement possible or tighter lower bound exists. (See Chapter 3 for more details.)

Comparison Trees: Searching/Sorting
Useful for modeling the way in which a large number of sorting and searching algorithms work,

Comparison-based algorithms - algorithms which work solely by making comparisons between elements; no arithmetic involving elements is permitted

A [1 : n]

p: permutation of {1, 2, ..., n}

The *sorting problem* calls for determining p, a permuation of the integers 1 to n, such that the n distinct values from S (the Set) stored in A[1:n] satisfy

A (p (1)) < A (p (2)) < . . .< A (p (n)) (ordering in terms of indices, not values) p(1) is the first in the permutation.

The *ordered searching problem* : A (1) < A (2) . . .< A (n)

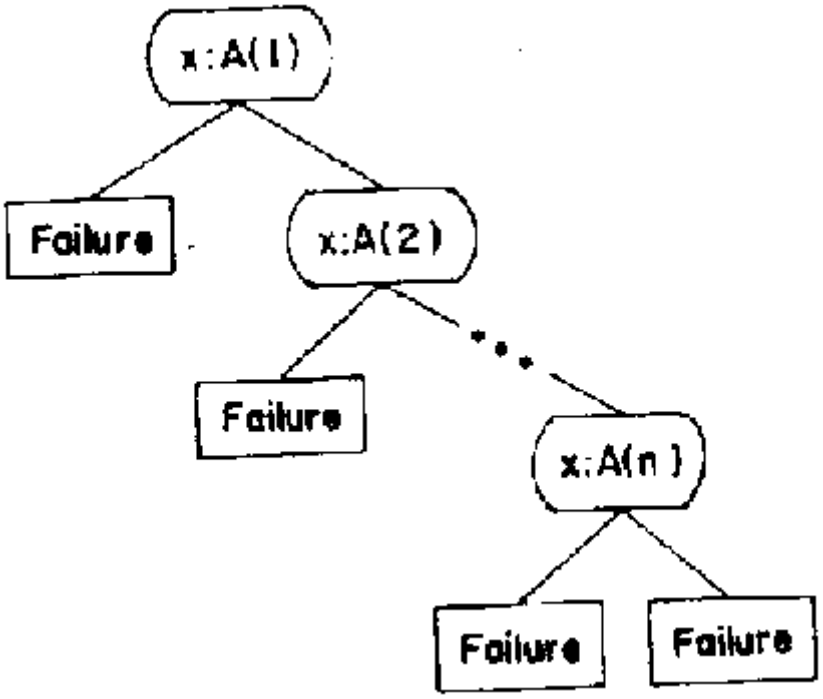
i	if x = A(i) for some i
0	otherwise

again: restriction on class of algorithms for which lower bound applies -
only comparison between elements - no arithmetic on keys.

Comparison Trees

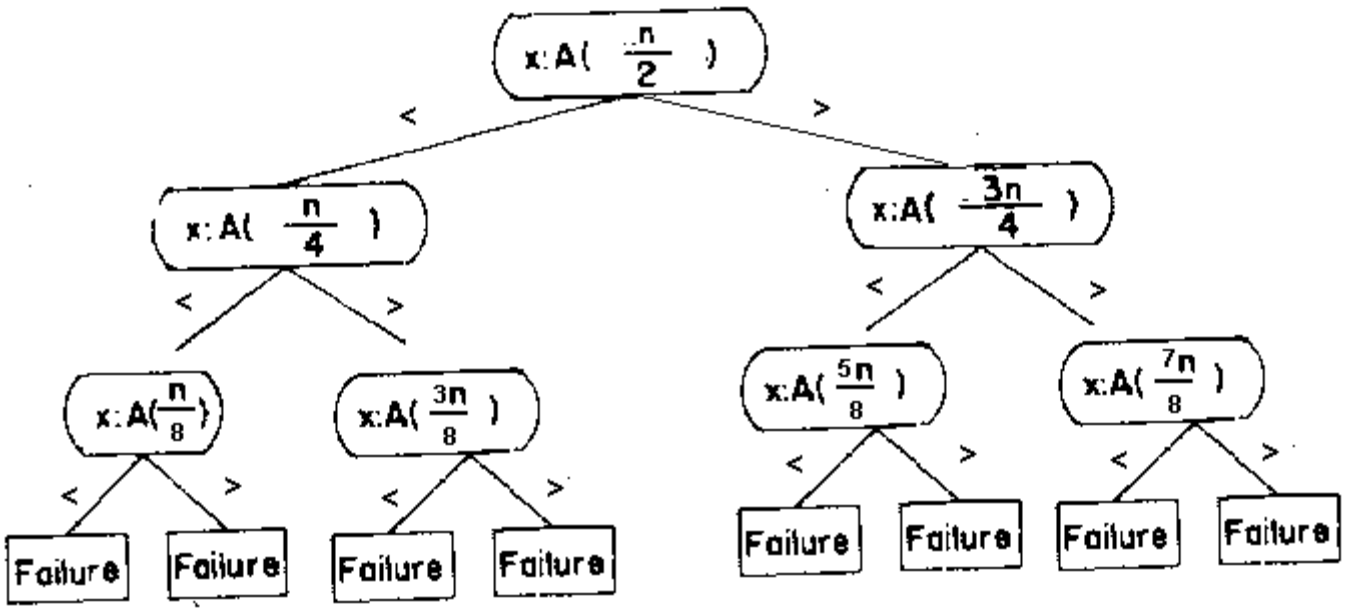
Search

Linear Search (ordered array)



running time - longest path in tree
- O(n)

Binary Search



In general

f (n) = O(log n)

If search terminates at interior nodes - success . . . . . external (leaf) nodes - failure

How do we reason to determine lower bound?

consider any comparison tree to search A [1 : n]

there must be n internal nodes corresponding to n possible successful outcomes.

k = maximum level of any internal node

then, number of internal nodes $\leq 2^k - 1$ (proof by induction)

$n \leq 2^k - 1$ implies $k \geq \log_2 (n+1)$

number of comparisons (worst case)

$= k = \log_2 (n+1) = 0 (\log n)$

$f (n) = \Omega (\log n)$ $f (n) = O(\log n)$

So best - optimal algorithm is binary search since it achieves this

Note that we did not look at any specific algorithm to detemine the lower bound, instead we considered information about the problem and considered what is the best we could do.

And, actually, we would want to consider the worst case of our best algorithm. Why? To see if it is the best possible.

Sort

comparison tree for sorting

number of external (leaf) nodes = number of distinct permutations of {1, 2, . . ., n}

= n!

let k be max. level of any internal node in tree

T(n) = k

If max level of any internal node is k, then at most 2^k external nodes
$n! \leq 2^k$; $k = T(n) \geq \log_2(n!)$

By Stirling's approximation (see page 462) = O(n log n)

Hence any comparison-based sorting algorithm needs Ω *(nlog n) time*

> One of the proof techniques that is useful for obtaining lower bounds consists of making use of an oracle. The most famous oracle in history was called the Delphic oracle, located in Delphi, Greece. This oracle can still be found situated in the side of the hill embedded in some rocks. In olden times people would approach the oracle and ask it a question. After some period of time elapsed, the oracle would reply and a caretaker would interpret the oracle's answer
>
> A similar phenomenon takes place when we use an oracle to establish a lower bound. Given some model of computation such as comparison trees, the oracle tells us the outcome of each comparison. To derive a good lower bound, the oracle tries its best to cause the algorithm to work as hard as it can. It does this by choosing as the outcome of the next test, the result that causes the most work to be required to determine the final answer. And by keeping track of the work done, a worse-case lower bound for the problem can be derived.

Polynomial Reductions

Here we discuss a very important technique that can be used to derive lower bounds. This technique calls for *reducing* the given problem to another problem for which the lower bound is already known.

Two problems L_1 and L_2

$T_1(n)$ = time to solve L_1

$T_2(n)$ = time to solve L_2

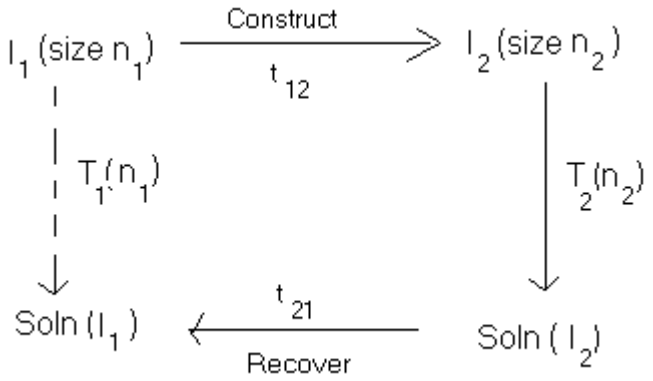
$L_1 \propto L_2$ L_1 reduces to L_2 implies $T_1(n) \leq p(n) * T_2(n)$ Where p is some polynomial

Definition: Let P_1 and P_2 be any two problems. We say *P_1 reduces to P_2* in time p(n) if an instance of P_1 can be converted into an instance of P_2 and a solution of P_1 can be obtained from a solution of P_2 in time $\leq$ p(n)

proc L_1 ()
  call L_2 ( )
end

proc L_2 ()
end

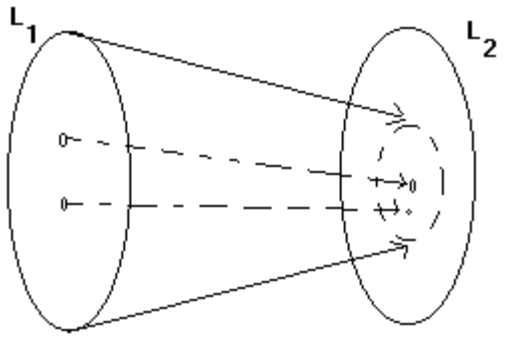
For each instance I_1 of L_1 construct an instance I_2 of L_2 such that a solution to I_1 can be recovered from a solution to I_2.



(size n_2 since we may need to add some more structure to our problem in order to cast is as a problem in L_2)

$T_1(n_1) = t_{12} + T_2(n_2) + t_{21} \leq p(n_1) * T_2(n_1)$
n_2 = poly (n_1) (i.e., must be bounded by a polynomial in n_1 - otherwise, t_{12} non polynomial) what?

$T_1(n_1) \leq p(n_1) * T_2(n_1)$



L_1: single source shortest path in undirected graphs

L_2: single source shortest path in directed graphs

proc L_1 (G = (V,E), c(E), p)
// G - undirected graph
// c - edge costs
// p - shortest path

// construction: instance of l_1 making it an instance of l_2
// make all directions go both ways

$E^{->}$= { $(u,v)^{->}$, $(v,u)^{->}$ | (u,v) ∈ E}

$G^{->}$ = $(V,E^{->})$

$c(u,v)^{->}$ = $c(v,u)^{->}$ = c(u,v) for all (u,v) ∈ E

call L_2 ($G^{->}$=$(V,E^{->})$, $c(E^{->})$, p)

// recover //

p = **undirected version of $p^{->}$**

end

(Notice that part that says ***undirected version of $p^{->}$***)
This is important...**any** directed to undirected does not work - it needs to maintain the information (we will see this in next lesson)

Claim: p shortest path in G <==> $p^{->}$ shortest path in $G^{->}$

not for **any** directed graph ... for ones considering in such a problem reduction

Proof:

==>
suppose p is a shortest path in G. Then there exists a path $p^{->}$ in $G^{->}$ of same cost.

l_1 and l_2 are optimization problems. Suppose S_1 is an optimal solution for l_1. $S^{->}{}_1$ is a solution to l_2 such that the value $OPT(S^{->}{}_1) = OPT(S_1)$

But, did conversion of l_1 to l_2 change anything? (is it possible that there exists some u -> q -> v in $G^{->}$ such that the cost($q^{->}$) < cost (q)?)

No. Since we never introduced a path in l_2 that does not exist in l_1

<==
suppose $p^{->}$ is shortest path in $G^{->}$. Then the undirected version of $p^{->}$ has same cost.

In human words...why does this work?

We have an undirected graph and want the shortest path. If we have something undirected, it can be interpreted as directed both ways. Since we have a way to find a solution for a shortest path when we have directions, what is the harm (or difference) in pretending that we **do** have directions? Pretend, Solve, stop Pretending. We did not change anything in the definition of the original graph.

Cost of construction + recover = O(|V| + |E|)

$T_1(n_1) \leq$ poly $(n_1) * T_2(n_2)$ number of edges in directed is at most twice what it was before

If $T_2(n)$ is also polynomial in n then we go one step further
$T_1(n_1) \leq t_{12} + T_2(n_2) + t_{21} \leq t_{12} + t_{21} + poly(n) * T_2(n_1) \leq poly_2(n_1) * T_2(n_1)$

and $T_1(n_1)$ is hence some poly. in n.

5.5 NP HARD & NP COMPLETE PROBLEM

Earlier we (informally) explained that a problem is called NP-Complete if P has at least one Non-Deterministic polynomial-time solution and further, so far, no polynomial-time Deterministic TM is known that solves the problem.

In this section, we formally define the concept and then describe a general technique of establishing the NP-Completeness of problems and finally apply the technique to show some of the problems as NP-Complete. We have already explained how a problem can be thought of as a language L over some alphabet $\sum$. Thus the tems *problem and language* may be interchangeably used.

For the formal definition of NP-Completeness, *polynomial-time reduction,* as defined below, plays a very important role.

In the previous unit, we discussed *reduction technique* to establish some of the problems as undecidable. The method that was used for *establishing undecidability* of a language using the technique of reduction may be briefly described as follows:

Let P_1 be a problem which is *already known* to be undecidable. We want to check whether a problem P_2 is undecidable or not. If we are able to design an algorithm which transforms or constructs an instance of P_2 for each instance of P_1, then P_2 is also undecidable.

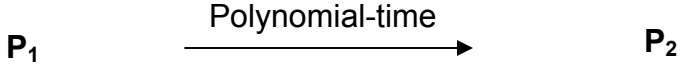
The process of transformation of the instaces of the problem **already known to the undecidable to** instances of the problem, the undecidability is to checked, is called reduction.

Some-what similar, but, slightly different, rather special, reduction called **polynomial-time reduction** is used to establish NP-Completeness of problems P_1.

A Polynomial-time reduction is a polynomial-time algorithm which constructs the instances of a problem P_2 from the instances of some other problems P_1.

A method of establishing the NP-Completeness (to be formally defined later) of a problem P_2 constitutes of designing a *polynomial time reduction* that constructs an instance of P_2 for each instance of P_1, where P_1 is already known to be NP-Complete.

The direction of mapping must be clearly understood as shown below:



Reduction

(Problem already known to be undecidable ⟶ (Problem whose NP-Complete is to be established)

Though we have already explained the concept of NP-Completeness, yet for the sake of completeness, we give below the formal definition of NP-Completeness.

Definition: NP-Complete Problem: A Problem P or equivalently its language L_1 is said to be NP-Complete if the following two conditions are satisfied:

(i) The problem L_2 is in the class NP

(ii) For any problem L_2 in NP, there is a polynomial-time reduction of L_1 to L_2.

In this context, we introduce below another closely related and useful concept.

Definition: NP-Hard Problem: A problem L is said to be NP-hard if for any problem L_1 in NP, there is a polynomial-time reduction of L_1 to L.

In other words, a *problem L is hard if* only condition (ii) of NP-Completeness is satisfied. But the problem has may be so hard that establishing L as an NP-class problem is so far not possible.

However, from the above definitions, it is clear that every NP-complete problem L must be NP-Hard and additionally should satisfy the condition that L is an NP-class problem.

In the next section, we discuss NP-completeness of some of problems discussed in the previous section.

ESTABLISHING NP-COMPLETENESS OF PROBLEMS

In general, the process of establishing a problem as NP-Complete is a two-step process. **The first step**, which in most of the cases is quite simple, constitutes of guessing possible solutions of the instance at a time, of the problem and then **verifying** whether the guess actually is a solution or not.

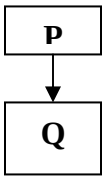
The second step involves designing a polynomial-time algorithm which reduces instances of an already known NP-Complete problem to instances of the problem, which is intended to be shown as NP-Complete.

However, to begin with, there is a major hurdle in execution of the second step. The above technique of reduction can not be applied unless we already have established at least one problem as NP-Complete. Therefore, for the first NP-Complete problem, the NP-Completeness has to be established in a different manner.

As mentioned earlier, Stephen Cook (1971) established Satisfiability as the first NP-Complete problem. The proof was based on explicit reduction of the language of any non-deterministic, polynomial-time TM to the satisfiability problem.

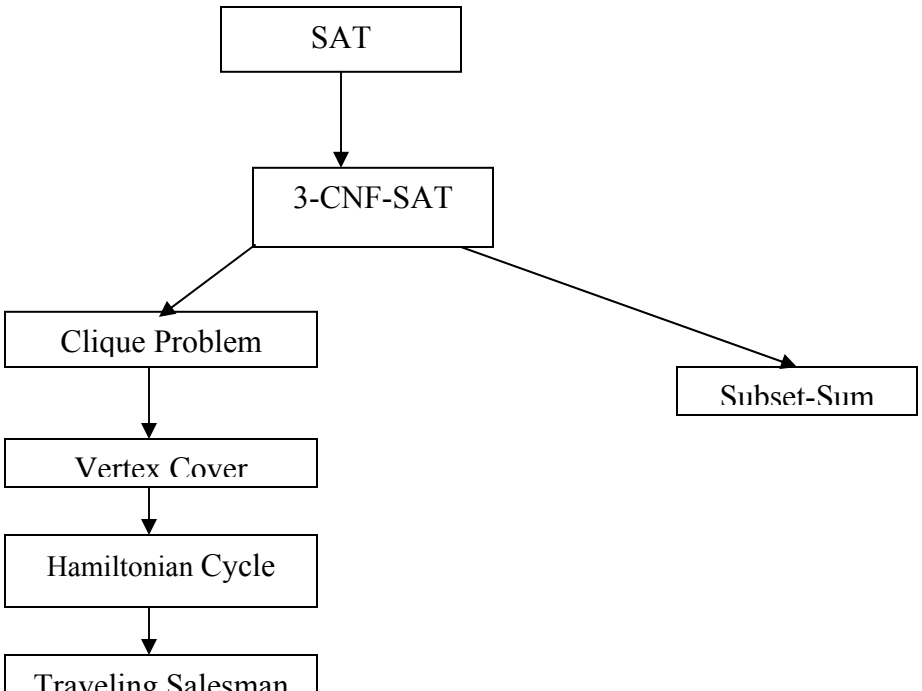
The proof of Satisfiability problem as the first NP-Complete problem is quite lengthy and we skip the proof. Interested readers may consult any of the text given in the reference.

Assuming the satisfiality problem as NP-Complete, the rest of the problems that we establish as NP-complete, are established by reduction method as explained above. A diagrammatic notation of the form.



Indicates: *Assuming P is already established as NP-Complete, the NP-Completeness of Q is established by through a polynomial-time reduction from P to Q.*

A scheme for establishing NP-Completeness of some the problems mentioned in Section 2.2 is suggested by Figure. 3.1 given below:



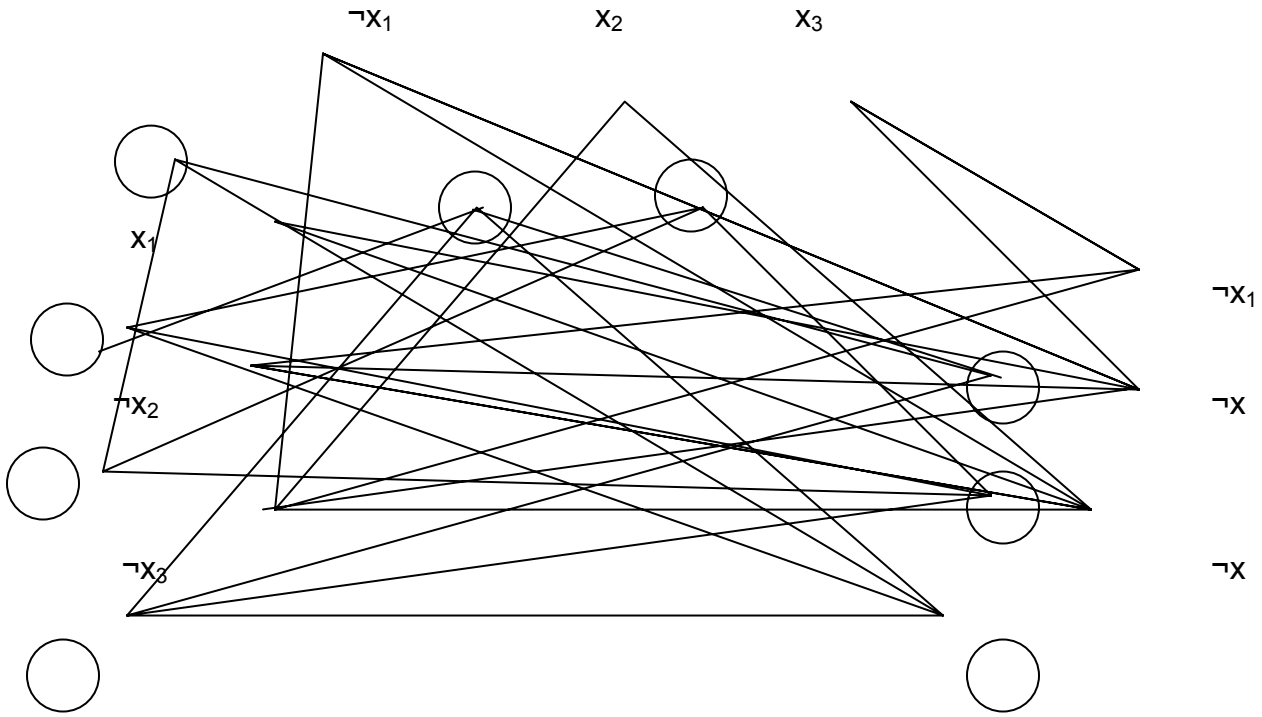
Example: Show that the Clique problem is an NP-Complete Problem.

Proof: The verification of whether every pairs of vertices is connected by an edge in E, is done for different paris of vertices by a Non-deterministic TM, i.e., in parallel. Hence, it takes only polynomial time because for each of n vertices we need to verify at most n (n+1)/2 edge, the maximum number of edges in a graph with n vertices.

We next show that 3-CNF-SAT problem can be transformed to clique problem in polynomial time.

Take an instance of 3CNF-SAT. An instance of 3CNF-SAT consists of a set of n clauses, each consisting of exactly 3 literals, each being either a variable or negated variable. It is satisfiable if we can choose literals in such a way that:

- at least one literal from each clause is chosen
- if literal of form x is chose, no literal of form ¬x is considered.



For each of the literals, create a graph node, and connect each node to every node in other clauses, except those with the same variable but different sign. This graph can be easily computed from a Boolean formula ø in 3-CNF-SAT in polynomial time.

Consider an example, if we have-

$Ø = (¬ x_1 V x_2 V x_3) ^ (x_1 V ¬x_2 V ¬x_3) ^ (¬ x_1 V ¬x_2 V ¬x_3)$

then G is the graph shown in Figure 3.2 above

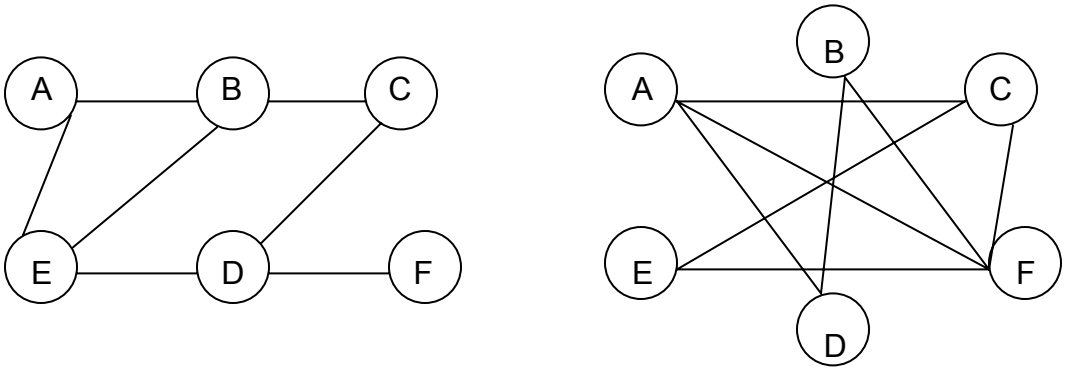
In the given example, a satisfying assignment of Ø is (x_1=0, x_2 = 0, x_3 = 1). A corresponding clique of k = 3 consists of the vertices corresponding to x_2 from the first clause, $¬x_3$ from the second clause, and $¬ x_3$ from the third clause.

The problem of finding n-element clique is equivalent to finding a set of literals satisfying SAT. Because there are no edges between literals of the same clause, such a clique must contain exactly one literal from each clause. And because there are no edges between literals of the same variable but different sign, if node of literal x is in the clique, no node of literal of form ¬x is.

This proves that finding n-element clique in 3-n-element graph in **NP-Complete.**

Example:5- Show that the Vertex cover problem is an **NP-Complete.**

A vertex cover of an undirected graph G = (V, E) is a subset V of the vertices of the graph which contains at least one of the two endpoints of each edge.



The vertex cover problem is the optimization problem of finding a vertex cover of minimum size is a graph. The problem can also be stated as a decision problem:

VERTEX-COVER = {<*G, k*>| graph *G* has a vertex cover of size k}.

A deterministic a algorithm to find a vertex cover in a graph is to list all subsets of vertices of size *k* and check each one to see whether it forms a vertex cover. This algorithm is exponential in *k.*

Proof: To show that Vertex cover problem Є **NP**, for a given graph G = (V, E), we take V' □ V and verifies to see if it forms a vertex cover. Verification can be done by checking for each edge (u, v) Є E whether u Є V' or v Є V'. This verification can be done in polynomial time.

Now, we show that clique problem can be transformed to vertex cover problem in polynomial time. This transformation is based on the notion of the complement of a graph G. Given an undirected graph G = (V, E), we define the complement of G as G' = (V, E') where E' = {(u, v) | (u, v) ∉ E}. i.e. G' is the graph containing exactly those edges that are not in G. The transformation takes a graph G and k of the clique problem. It computes the complement G' which can be done in polynomial time.

To complete the proof, we can show that this transformation is indeed reduction: the graph has a clique of size k if and only if the graph G' has vertex cover of size |v| - k.

Suppose that G has a clique V' □ V with |V'| = k. We claim that V – V' is a vertex cover in G'. Let (u, v) be any edge in E'. Then, (u, v) ∉ E, which implies that atleast one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. Equivalently, atleast one of u or v is n V – V'. Since (u, v) was chosen arbitrarily from E', every edge of E' is covered by a vertex in V – V'. Hence, the set V – V', which has size |V| - k, forms a vertex cover for G'.

Conversely, suppose that G' has a vertex cover V' □ V, where |V'| = |V| - k, then, for all v, v Є V, if (u, v) Є E', then u Є V' or v Є V' or both. The contra positive of this implication is that for all u, v Є V, if u ∉ V' and v Є V', then (u, v) Є E. In other words, V – V' is a clique, and it has size |V| - |V'| = k.

For example, the graph G (V, E) has a clique {A, B, E}. The complement of graph G is given by G' and have independent set given by {C, D, F}.

This proves that finding the vertex cover is **NP-Complete.**

5.6 EFFICIENCY OF BRANCH AND BOUND

The efficiency of the branch & bound algorithms is determined by the following things.

1. The starting value of u (upperbound) can't be decreased by expanding x, hence such an expansion cannot affect the operation of the algorithms of the remainder of the tree.
2. If u_1 and u_2 are two initial upper bounds on the cost of a minimum cost solution node in the states space tree T1 and $u_1 < u_2$, then branch and bound algorithms beginning with u_1 will generate no more nodes than they would if they started with u_2 as the initial upper bound.
3. The use of a better c function in conjunction with branch and bound algorithms will not increase the number of the nodes generated.
4. If a better c function is used in branch & bound algorithms, the number of nodes generated may increase.
5. The number of nodes generated during a branch & bound search for a least cost solution node may increase when a stranger dominance relation is used.

5.7 SUMMARY

This unit discusses some branch and Bound techniques for least cost search, for minimization problems.

These techniques are illustrated with examples of its applications to solving problems of I/O Knapsack problem traveling sales person, lower bound theory and NP-HARD & NP Complete Problems.

5.8 KEYWORDS

- **Branch and Bound:** deals with only minimization problems.
- **NP Hard:** A problem L is said to be NP-Hard if for any problem L_1 in NP, there is a Polynomial time reduction of L_1 to L.

5.9 REVIEW QUESTIONS

1. Explain how function

C (x) = f ((h) (x)) + g (x) the LC search strategy can also be used for BFS and D-search.

2. Explain the O/I Knapsack problem and how the cost functions are defined for the various nodes.
3. Consider the traveling sales person instance defined by the cost matrix.

$$\begin{bmatrix} \alpha & 7 & 3 & 12 & 8 \\ 3 & \alpha & 6 & 14 & 9 \\ 5 & 8 & \alpha & 6 & 18 \\ 18 & 14 & 9 & 8 & \alpha \end{bmatrix}$$

(a) Obtain the reduced cost matrix

(b) Generate the state space tree.

4. What is bounding?

5.10 FURTHER READINGS

16. Horowitz E and Sahni S., “Fundamental of Computer Algorithm” Galgotia Publications.
17. Aho A. V. Hoperoft, J.E. and Ullman, J.D., “Design and Analysis of Algorithm” Addison Wesley.
18. D. Harel., “ Algorithmics : The Spirit of Computing” Addison Wesley.
19. Ravi Sethi, “Programming Languges- Concept and Constructs” Pearson Education, Asia, 1996.