LECTURE NOTES

UNIT-I

Introduction & Elementary Data Structures: Introduction, Fundamentals of algorithm(Line Count, Operation Count), Analysis of algorithms(Best, Average, Worst case), Asymptotic Notations(O,Ω,Ө) Recursive Algorithms, Analysis using Recurrence Relations, Master's Theorem.

Review of elementary data structures–Graphs: BFS, DFS, Bi-Connected Components. Sets: representation, UNION, FIND operations.

Course Outcome:

- Analyze time and space complexities of algorithms using asymptotic notations

Introduction

Algorithm

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

We represent algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the *time complexity* of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

Space needed by constants and simple variables in program.

- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

- Best Case: The minimum possible value of f(n) is called the best case
- Average Case : The expected value of f(n).
- Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

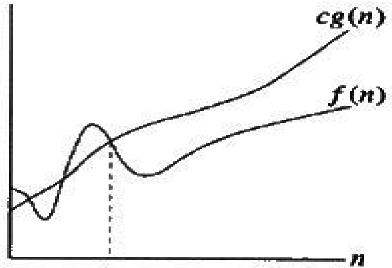
Rate of Growth:

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

- Big–OH (O)
- Big–OMEGA (Ω)
- Big–THETA (Θ)

Big–OH O (Upper Bound)

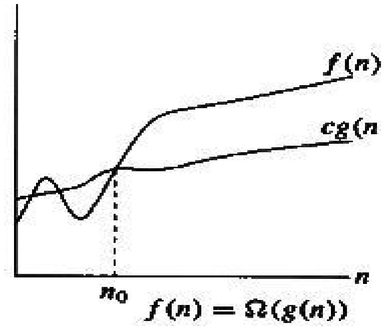
f(n) = O(g(n)), (pronounced order of or big oh), says that the growth rate of f(n) is less than or equal (≤) that of g(n).



$f(n) = O(g(n))$

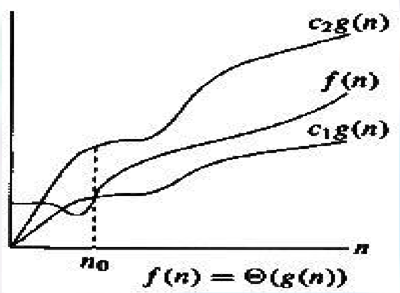
Big–OMEGA Ω (Lower Bound)

f(n)* = Ω *(g(n)) (pronounced omega), says that the growth rate of f(n) is greater than or equal to (≥) that of g(n).



Big–THETA Θ (Same order)

f(n)* = Θ *(g(n)) (pronounced theta), says that the growth rate of f(n) equals (=) the growth rate of g(n) [if f(n) = O(g(n)) and T(n) = Ω (g(n)].



Analyzing Algorithms

Suppose 'M' is an algorithm, and suppose 'n' is the size of the input data. Clearly the complexity f(n) of M increases as n increases. It is usually the rate of increase of f(n) we want to examine. This is usually done by comparing f(n) with some standard functions. The most common computing times are:

O(1), O($\log_2$ n), O(n), O(n. $\log_2$ n), O(n^2), O(n^3), O(2^n), and n^n

The Running time of a program

When solving a problem we are faced with a choice among algorithms. The basis for this can be any one of the following:

- We would like an algorithm that is easy to understand, code and debug.
- We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

Measuring the running time of a program

The running time of a program depends on factors such as:

- The input to the program.
- The quality of code generated by the compiler used to create the object program.
- The nature and speed of the instructions on the machine used to execute the program, and
- The time complexity of the algorithm underlying the program.

Asymptotic Analysis of Algorithms:

Our approach is based on the *asymptotic complexity* measure. This means that we don't try to count the exact number of steps of a program, but how that number grows with the size of the input to the program. That gives us a measure that will work for different operating systems, compilers and CPUs. The asymptotic complexity is written using big-O notation.

Rules for using big-O:

The most important property is that big-O gives an upper bound only. If an algorithm is O(n^2), it doesn't have to take n^2 steps (or a constant multiple of n^2).

Calculating the running time of a program:

Let us now look into how big-O bounds can be computed for some common algorithms.

Example 1:

Let's consider a short piece of source code:

```
x = 3*y + 2;
z = z + 1;
```

If y, z are scalars, this piece of code takes a *constant* amount of time, which we write as O(1). In terms of actual computer instructions or clock ticks, it's difficult to say exactly how long it takes. But whatever it is, it should be the same whenever this piece of code is executed. O(1) means *some* constant, it might be 5, or 1 or 1000.

Example 2:

Analysis of simple for loop

Now let's consider a simple for loop:

```
for (i = 1; i<=n; i++)
        v[i] = v[i] + 1;
```

This loop will run exactly n times, and because the inside of the loop takes constant time, the total running time is proportional to n. We write it as O(n). The actual number of instructions might be 50n, while the running time might be 17n microseconds. It might even be 17n+3 microseconds because the loop needs some time to start up. The big-O notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As long as it's a linear function which is proportional to n, the correct notation is O(n) and the code is said to have *linear* running time.

Example 3:

Analysis for nested for loop

Now let's look at a more complicated example, a nested for loop:

```
for (i = 1; i<=n; i++)
for (j = 1; j<=n; j++)
a[i,j] = b[i,j] * x;
```

The outer for loop executes N times, while the inner loop executes n times for every execution of the outer loop. That is, the inner loop executes n   n = n^2 times. The assignment statement in the inner loop takes constant time, so the running time of the code is O(n^2) steps. This piece of code is said to have *quadratic* running time.

Example 4:

Analysis of matrix multiply

Lets start with an easy case. Multiplying two n n matrices. The code to compute the matrix product C = A * B is given below.

```
for (i = 1; i<=n; i++)
for (j = 1; j<=n; j++)
C[i, j] = 0;
for (k = 1; k<=n; k++)
C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes n*n*n = n^3 times. The innermost statement, which contains a scalar sum and product takes constant O(1) time. So the algorithm overall takes
O(n^3) time.

Recursive Algorithms:

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function. Using a recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Let us consider a problem that a programmer has to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply to add the numbers starting from 1 to n. So the function simply looks like this,

approach(1) – Simply adding one by one
f(n) = 1 + 2 + 3 +……..+ n

but there is another mathematical approach of representing this,

approach(2) – Recursive adding
f(n) = 1 n=1
f(n) = n + f(n-1) n>1

Recursion uses more memory, because the recursive function adds to the stack with each recursive call, and keeps the values there until the call is finished. The recursive function uses LIFO Structure just like the stack data structure.

In the recursive program, the solution to the base case is provided and the solution to the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

In the above example, the base case for n < = 1 is defined and the larger value of a number can be solved by converting to a smaller one till the base case is reached.

Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

T (n) = θ (1) if n=1

$2T\left(\frac{n}{2}\right)$ + θ (n) if n>1

There are four methods for solving Recurrence:

- Substitution Method
- Iteration Method
- Recursion Tree Method
- Master Theorem

Master Theorem

The Master Theorem is a tool used to solve recurrence relations that arise in the analysis of divide-and-conquer algorithms. The Master Theorem provides a systematic way of solving recurrence relations of the form:

T(n) = aT(n/b) + f(n)

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
f(n) = cost of work done outside the recursive calls like dividing into subproblems and cost of combining them to get the solution.

Not all recurrence relations can be solved with the use of the master theorem i.e. if

- T(n) is not monotone, ex: T(n) = sin n
- f(n) is not a polynomial, ex: T(n) = 2T(n/2) + 2n

This theorem is an advance version of master theorem that can be used to determine running time of divide and conquer algorithms if the recurrence is of the following form :-

$$\mathbf{T(n) = aT(n/b) + O(n^k \log^p n)}$$

where n = size of the problem
a = number of subproblems in the recursion and a >= 1
n/b = size of each subproblem
b > 1, k >= 0 and p is a real number.

Then,

1. if a > bk, then T(n) = θ(nlogba)

2. if a = bk, then
   (a) if p > -1, then T(n) = θ(nlogba logp+1n)
   (b) if p = -1, then T(n) = θ(nlogba loglogn)
   (c) if p < -1, then T(n) = θ(nlogba)
1. if a < bk, then
   (a) if p >= 0, then T(n) = θ(nk logpn)
   (b) if p < 0, then T(n) = θ(nk)

Time Complexity Analysis –

- Example-1: Binary Search – T(n) = T(n/2) + O(1)
  a = 1, b = 2, k = 0 and p = 0
  bk = 1. So, a = bk and p > -1 [Case 2.(a)]
  T(n) = θ(nlogba logp+1n)
  T(n) = θ(logn)

- Example-2: Merge Sort – T(n) = 2T(n/2) + O(n)
  a = 2, b = 2, k = 1, p = 0
  bk = 2. So, a = bk and p > -1 [Case 2.(a)]
  T(n) = θ(nlogba logp+1n)
  T(n) = θ(nlogn)

- Example-3: T(n) = 3T(n/2) + n2
  a = 3, b = 2, k = 2, p = 0
  bk = 4. So, a < bk and p = 0 [Case 3.(a)]
  T(n) = θ(nk logpn)
  T(n) = θ(n2)

- Example-4: T(n) = 3T(n/2) + log2n
  a = 3, b = 2, k = 0, p = 2
  bk = 1. So, a > bk [Case 1]
  T(n) = θ(nlogba)
  T(n) = θ(nlog23)

- Example-5: T(n) = 2T(n/2) + nlog2n
  a = 2, b = 2, k = 1, p = 2
  bk = 2. So, a = bk [Case 2.(a)]
  T(n) = θ(nlogbalogp+1n)
  T(n) = θ(nlog22log3n)
  T(n) = θ(nlog3n)

- Example-6: T(n) = 2nT(n/2) + nn
  This recurrence can't be solved using above method since function is not of form T(n) = aT(n/b) + θ(nk logpn)

Graphs

Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).

Components of a Graph

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook

BFS ,DFS:

Both BFS and DFS are types of graph traversal algorithms, but they are different from each other. BFS or Breadth First Search starts from the top node in the graph and travels down until it reaches the root node. On the other hand, DFS or Depth First Search starts from the top node and follows a path to reaches the end node of the path.

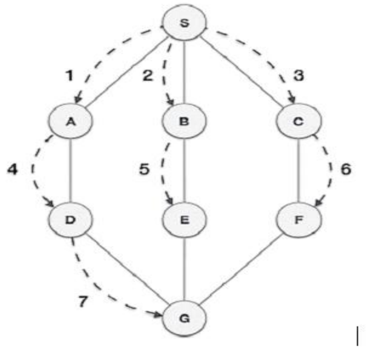
Read this article to learn more about these two graph traversal algorithms and how they are different from each other.

BFS

Breadth First Search (BFS) algorithm traverses a graph in a breadth-ward motion and uses a queue to remember to get the next vertex to start a search when a dead end occurs in any iteration.BFS is basically a nodebased algorithm which is used to find the shortest path in the graph between two nodes. BFS moves through all of its nodes which are connected to the individual nodes.

BFS uses the FIFO (First In First Out) principle while using the Queue to find the shortest path. However, BFS is slower and requires a large memory space.

Example of BFS

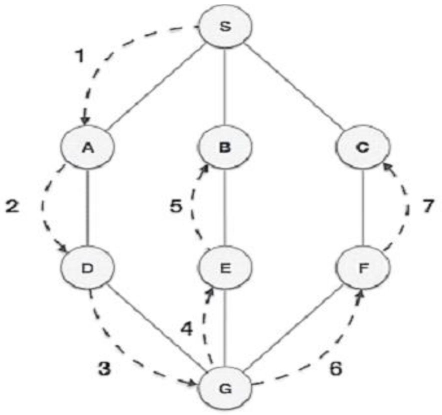


DFS

Depth First Search (DFS) algorithm traverses a graph in a depth-ward motion and uses a stack to remember to get the next vertex to start a search when a deadend occurs in any iteration.

DFS uses LIFO (Last In First Out) principle while using Stack to find the shortest path. DFS is also called Edge Based Traversal because it explores the nodes along the edge or path. DFS is faster and requires less memory. DFS is best suited for decision trees.

Example of DFS



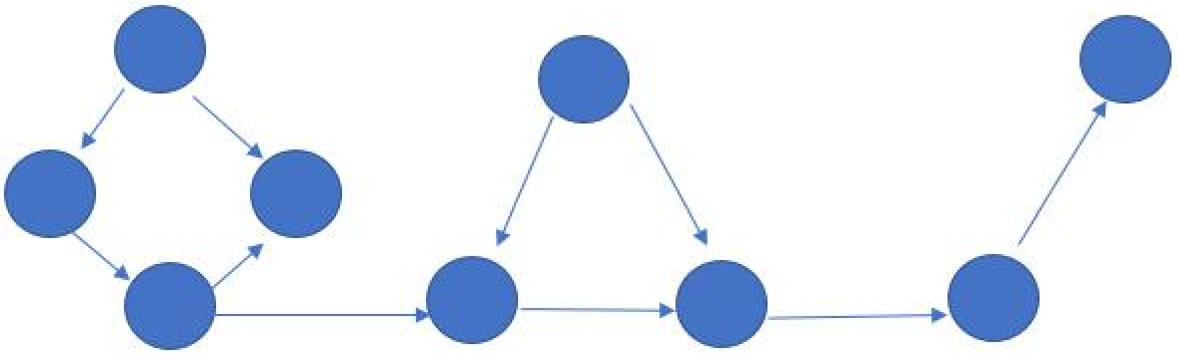
Biconnected Components

The concept of biconnected components is based on the concept of the disc and low values algorithms.

The maximal sub-graph is known to be the concept behind Biconnected Components.

Biconnected components are a part of the graph data structure created.

We will look for the articulation points using the depth-first search and store the visited edges in a stack. When the articulation points are found, the edges stored in the stack data structure will form an entity of biconnected components.



Representation of Biconnected Components

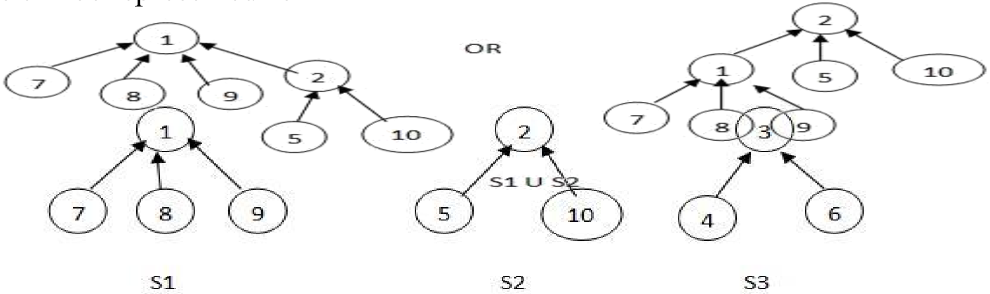
Set Representation:

The set will be represented as the tree structure where all children will store the address of parent / root node. The root node will store null at the place of parent address. In the given set of elements any element can be selected as the root node, generally we select the first node as the root node.

Example:

S1={1,7,8,9} S2={2,5,10} s3={3,4,6}

Then these sets can be represented as

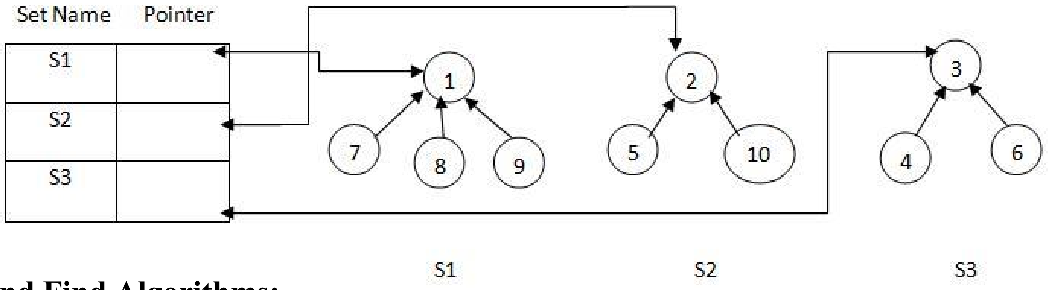


Disjoint Union:

To perform disjoint set union between two sets Si and Sj can take any one root and make it sub-tree of the other. Consider the above example sets S1 and S2 then the union of S1 and S2 can be represented as any one of the following.

Find:

To perform find operation, along with the tree structure we need to maintain the name of each set. So, we require one more data structure to store the set names. The data structure contains two fields. One is the set name and the other one is the pointer to root.

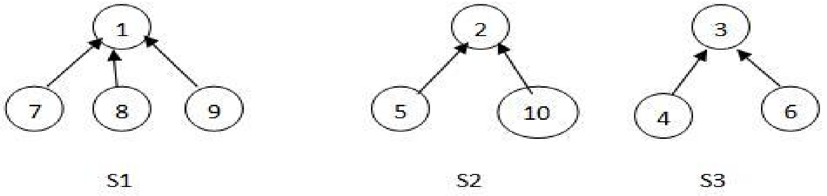


Union and Find Algorithms:

In presenting Union and Find algorithms, we ignore the set names and identify sets just by the roots of trees representing them. To represent the sets, we use an array of 1 to n elements where n is the maximum value among the elements of all sets. The index values represent the nodes (elements of set) and the entries represent the parent node. For the root value the entry will be '-1'.

Example:

For the following sets the array representation is as shown below.



i	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]
p	-1	-1	-1	3	2	3	1	1	1	2

Algorithm for Union operation:

To perform union the **SimpleUnion(i,j)** function takes the inputs as the set roots i and j . And make the parent of i as j i.e, make the second root as the parent of first root.

```
Algorithm SimpleUnion(i,j)
{
P[i]:=j;
}
```

Algorithm for find operation:

The SimpleFind(i) algorithm takes the element i and finds the root node of i. It starts at I until it reaches a node with parent value -1.

```
Algorithms SimpleFind(i)
{
while( P[i]≥0) do i:=P[i];
return i;
}
```

Analysis of SimpleUnion(i,j) and SimpleFind(i):

Although the SimpleUnion(i,j) and SimpleFind(i) algorithms are easy to state, their performance characteristics are not very good. For example, consider the sets

Union(2,3)……. Union(n-1,n) and sequence of Find(1), Find(2)……… Find(n).

Since, the time taken for a Union is constant, the n-1 sequence of union can be processed in time O(n). And for the sequence of Find operations it will take time $O(n^2)$

We can improve the performance of union and find by avoiding the creation of degenerate tree by applying weighting rule for Union.

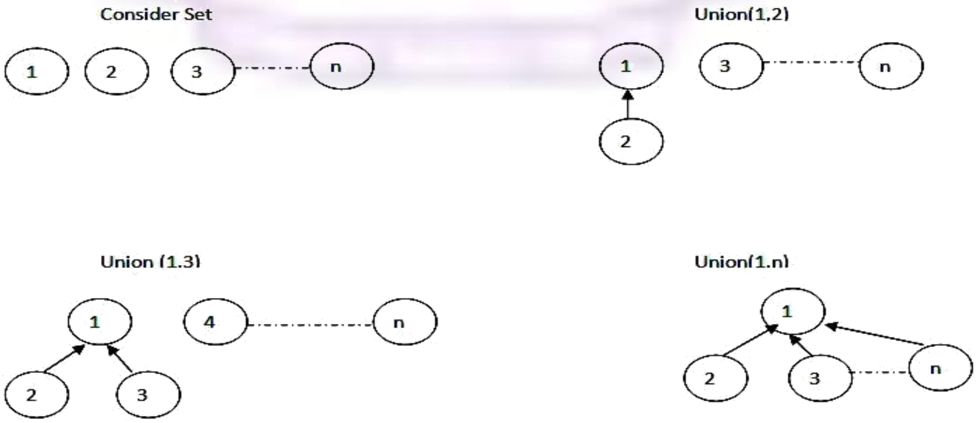
Weighting rule for Union:

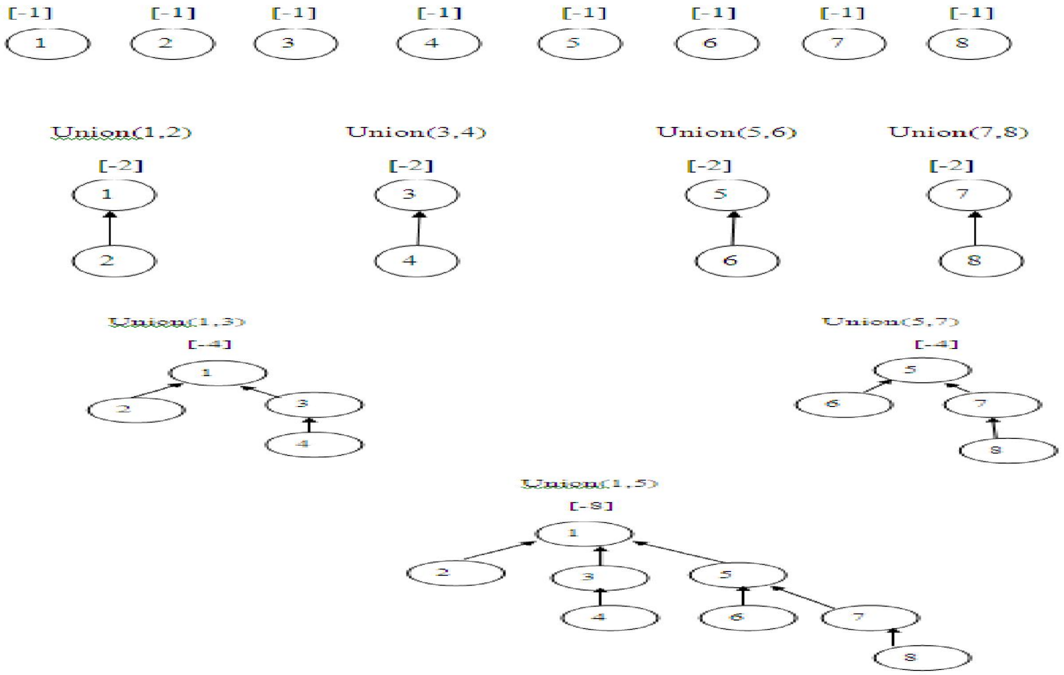
If the number of nodes in the tree with root i is less than the number in the tree with the root j, then make ‘j’ the parent of i; otherwise make ‘i' the parent of j.

To implement weighting rule we need to know how many nodes are there in every tree. To do this we maintain “count” field in the root of every tree. If ‘i' is the root then count[i] equals to number of nodes in tree with root i.

Since all nodes other than roots have positive numbers in parent (P) field, we can maintain count in P field of the root as negative number.

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j , i≠j using the weighted rule P[i]=-count[i] and p[j]= -count[j]
{
temp:= P[i]+P[j]; if if (P[i]>P[j])
then
{
// i has fewer nodes
P[i]:=j;
P[j]:=temp;
}
else
{
// j has fewer nodes P[j]:=i;
P[i]:=temp;
}
}
```





Collapsing rule for find:

If j is a node on the path from i to its root and p[i]≠root[i], then set P[j] to root[i]. Consider the tree created by Weighted Union() on the sequence of 1≤i≤8. Union(1,2), Union(3,4), Union(5,6) and Union(7,8)

Now process the following eight find operations

Find(8), Find(8)………………………Find(8)

If SimpleFind() is used each Find(8) requires going up three parent link fields for a total of 24 moves .

When Collapsing find is used the first Find(8) requires going up three links and resetting three links. Each of remaining seven finds require going up only one link field. Then the total cost is now only 13 moves.(3 going up + 3 resets + 7 remaining finds).

```
Algorithm CollapsingFind(i)
// Find the root of the tree containing element i
// use the collapsing rule to collapse all nodes from i to root.
{
r:=i;
while(P[r]>0) do r:=P[r]; //Find root while(i≠r)
{
//reset the parent node from element i to the root s:=P[i];
P[i]:=r;
i:=s;
}
}
```

UNIT-II

Divide-and-Conquer Method: The general method, Binary search, Finding maximum and minimum, Merge sort, Quick sort.
Brute Force: Knapsack, Traveling salesman problem, Convex-Hull
Course Outcome:

- Describe the Divide-and-Conquer paradigm and explain when an algorithmic design situation calls for it and to derive and solve recurrences describing the performance of divide-and-conquer algorithms.

Introduction

Divide-and-conquer general method

Both merge sort and quick sort employ a common algorithmic paradigm based on recursion. This paradigm, divide-and-conquer, breaks a problem into sub problems that are similar to the original problem, recursively solves the sub problems, and finally combines the solutions to the sub problems to solve the original problem. Because divide-and-conquer solves sub problems recursively, each sub problem must be smaller than the original problem, and there must be a base case for sub problems.

Divide the problem into a number of subproblems that are smaller instances of the same problem.

Conquer the sub problems by solving them recursively. If they are small enough, solve the sub problems as base cases.

Combine the solutions to the sub problems into the solution for the original problem. You can easily remember the steps of a divide-and-conquer algorithm as divide, conquer, combine. Here's how to view one step, assuming that each divide step creates two sub problems (though some divide-and-conquer algorithms create more than two):

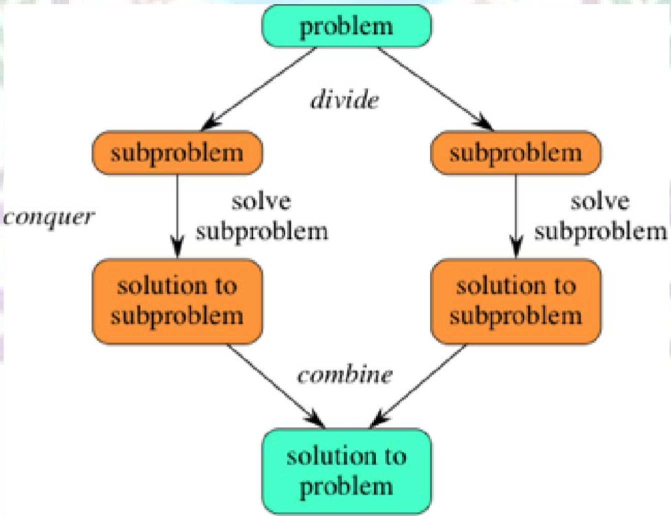


Fig. Divide and Conquer

Control Abstraction of Divide and Conquer

A control abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined. The control abstraction for divide and conquer technique is DANDC(P), where P is the problem to be solved.

```
DANDC (P)
{

        if SMALL (P) then return S (p); else
        {

                divide p into smaller instances p1, p2, …. Pk, k  1; apply DANDC to
                each of these sub problems;
                return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
        }

}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function 'S' is invoked otherwise, the problem 'p' into smaller sub problems. These sub problems p1, p2, . . . , pk are solved by recursive application of DANDC.

]Binary Search

If we have 'n' records which have been ordered by keys so that x1 < x2 < … < xn . When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).
In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every unsuccessful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration. Since the array size is roughly halved often each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about log 2n times before reaching a trivial length, the worst case complexity of Binary search is about log2n

Algorithm

```
// array a(1 : n) of elements in increasing order, n 0,
// determine whether 'x' is present, and if so, set j such that x = a(j)
// else return j
    {
    low :=1 ; high :=n ;
    while (low < high) do
    {
    mid :=|(low + high)/2|
    if (x < a [mid]) then high:=mid – 1;
    else if (x > a [mid]) then low:= mid + 1
    else return mid;
    }
    return 0;
    }
```

low and high are integer variables such that each time through the loop either ‘x’ is found or low is increased by at least one or high is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually low will become greater than high causing termination in a finite number of steps if ‘x’ is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101

The number of comparisons required for searching different elements is as follows:

1. Searching for x = 101

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

Number of comparisons = 4

2. Searching for x = 82

low	high	mid
1	9	5
6	9	7
8	9	8
		found

Number of comparisons = 3

3. Searching for x = 42

low	high	mid
1	9	5
6	9	7
6	6	6
7	6 not found	

Number of comparisons = 4

4. Searching for x = -14

low	high	mid
1	9	5
1	4	2
1	1	1
2	1 not found	

Number of comparisons = 3

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If $x < a[1]$, $a[1] < x < a[2]$, $a[2] < x < a[3]$, $a[5] < x < a[6]$, $a[6] < x < a[7]$ or $a[7] < x < a[8]$ the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons. Thus the average number of element comparisons for an unsuccessful search is:

$(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4$

The time complexity for a successful search is O(log n) and for an unsuccessful search is Θ(log n).

Successful searches Θ(1), Θ(log n), Θ(log n)

Best average worst un-successful searches Θ(log n) best, average and worst case

Finding Maximum and Minimum

- Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.
- Here 'n' is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.
- If the list has more than 2 elements, P has to be divided into smaller instances.

For example, we might divide 'P' into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2], a[[n/2]+1],….., a[n]) After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

```
MaxMin(i, j, max, min)

// a[1:n] is a global array. Parameters i and j are integers, // 1≤i≤j≤n. The effect is to set max and
min to the largest and // smallest values in a[i:j],
{
   if (i=j) then max := min := a[i]; //Small(P)
   else if (i=j-1) then // Another case of Small(P)
{
if (a[i] < a[j]) then max := a[j]; min := a[i];
else max := a[i]; min := a[j];
}
else
{
// if P is not small, divide P into sub-problems.
// Find where to split the set.
mid := ( i + j )/2;
// Solve the sub-problems.
MaxMin( i, mid, max, min );
MaxMin( mid+1, j, max1, min1 );
// Combine the solutions.
if (max < max1) then max := max1;
if (min > min1) then min := min1;
}
}
```

Complexity:

Now what is the number of element comparisons needed for MaxMin? If T(n) represents this number, then the resulting recurrence relation is

$$T(n) = \begin{cases} 0 & n=1 \\ 1 & n=2 \\ T(n/2) + T(n/2) + 2 & n>2 \end{cases}$$

When n is a power of two, n = 2k for some

positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/4) + 2) + 2 \\ &= 4T(n/4) + 4 + 2 \\ &. \\ &. \\ &. \\ &= 2k-1\ T(2) + \textstyle\sum(1 \le i \le k-1)\ 2k \\ &= 2k-1 + 2k - 2 \\ &= 3n/2 - 2 = O(n) \end{aligned}$$

Note that 3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two.

Merge Sort

Merge sort algorithm is a classic example of divide and conquer. To sort an array, recursively, sort its left and right halves separately and then merge them. The time complexity of merge mort in the *best case, worst case* and *average case* is O(n log n) and the number of comparisons used is nearly optimal.

This strategy is so simple, and so efficient but the problem here is that there seems to be no easy way to merge two adjacent sorted arrays together in place (The result must be build up in a separate array).

The fundamental operation in this algorithm is merging two sorted lists. Because the lists are sorted, this can be done in one pass through the input, if the output is put in a third list.

The basic merging algorithm takes two input arrays ‘a’ and ’b’, an output array ‘c’, and three counters, *a ptr, b ptr* and *c ptr,* which are initially set to the beginning of their respective arrays. The smaller of *a[a ptr]* and *b[b ptr]* is copied to the next entry in ‘c’, and the appropriate counters are advanced. When either input list is exhausted, the remainder of the other list is copied to ‘c’.

To illustrate how merge process works. For example, let us consider the array ‘a’ containing 1, 13, 24, 26 and ‘b’ containing 2, 15, 27, 38. First a comparison is done between 1 and 2. 1 is copied to ‘c’. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
h ptr			

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1							
i ptr							

and then 2 and 13 are compared. 2 is added to ‘c’. Increment *b ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
j ptr			

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2						
	i ptr						

then 13 and 15 are compared. 13 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
	h ptr		

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13					
		i ptr					

24 and 15 are compared. 15 is added to ‘c’. Increment *b ptr* and *c ptr*.

1	*2*	*3*	*4*
1	13	24	26
		h ptr	

5	*6*	*7*	*8*
2	15	27	28
	j ptr		

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	2	13	15				
			i ptr				

24 and 27 are compared. 24 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
		h ptr	

5	6	7	8
2	15	27	28
		j ptr	

1	2	3	4	5	6	7	8
1	2	13	15	24			
				i ptr			

26 and 27 are compared. 26 is added to ‘c’. Increment *a ptr* and *c ptr*.

1	2	3	4
1	13	24	26
			h ptr

5	6	7	8
2	15	27	28
		j ptr	

1	2	3	4	5	6	7	8
1	2	13	15	24	26		
					i ptr		

As one of the lists is exhausted. The remainder of the b array is then copied to ‘c’.

1	2	3	4
1	13	24	26

h
ptr

5	6	7	8
2	15	27	28
		j ptr	

1	2	3	4	5	6	7	8
1	2	13	15	24	26	27	28

i
ptr

Algorithm MERGESORT (low, high)

```
// a (low : high) is a global array to be sorted.

{ if (low < high)
        { mid :=    (low + high)/2  //finds where to split the set
                MERGESORT(low, mid) //sort one subset MERGESORT(mid+1, high) //sort the
                other subset MERGE(low, mid, high) // combine the results
        }

}
```

Algorithm MERGE (low, mid, high)

```
// a (low : high) is a global array containing two sorted subsets
// in a (low : mid) and in a (mid + 1 : high).
// The objective is to merge these sorted sets into single sorted // set residing
in a (low : high). An auxiliary array B is used.
{ h :=low; i := low; j:= mid + 1; while ((h ≤ mid) and (J
        ≤ high)) do
        { if (a[h] ≤ a[j]) then

                { b[i] := a[h]; h := h + 1;
                } else
                { b[i] :=a[j]; j := j + 1;
                } i := i + 1;
```

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

Quick Sort

The main reason for the slowness of Algorithms like SIS is that all comparisons and exchanges between keys in a sequence w1, w2, ... , wn take place between adjacent pairs. In this way it takes a relatively long time for a key that is badly out of place to work its way into its proper position in the sorted sequence.

Hoare his devised a very efficient way of implementing this idea in the early 1960's that improves the O(n^2) behavior of SIS algorithm with an expected performance that is O(n log n).

In essence, the quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value.

The chosen value is known as the *pivot element*. Once the array has been rearranged in this way with respect to the pivot, the very same partitioning is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers 'i' and 'j' which are moved toward each other in the following fashion:

- Repeatedly increase the pointer 'i' until a[i] >= pivot.
- Repeatedly decrease the pointer 'j' until a[j] <= pivot.

If j > i, interchange a[j] with a[i]

- Repeat the steps 1, 2 and 3 till the 'i' pointer crosses the 'j' pointer. If 'i' pointer crosses 'j' pointer, the position for pivot is found and place pivot element in 'j' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

- It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

- Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ...

. . . x[j-1] and x[j+1], x[j+2], . . .x[high].

- It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).
- It calls itself recursively to sort the right sub-array x[j+1], x[j+2], ... x[high] between positions j+1 and high.

Algorithm

```
   QUICKSORT(low, high)
    /* sorts the elements a(low), . . . . . , a(high) which reside in the global array A(1 : n)into ascending order a
    (n + 1) is considered to be defined and must be greater than all elements in a(1 : n);  */

    { if low < high then
            { j := PARTITION(a, low, high+1);
                                                    // J is the position of the partitioning element
                    QUICKSORT(low, j – 1);
                    QUICKSORT(j + 1 , high); }
    }
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)        pivot =
arr[high];
    i = (low - 1)  // Index of smaller element     for (j = low; j
<= high- 1; j++)
   {
      // If current element is smaller than the pivot         if (arr[j]
< pivot)
       {
          i++;    // increment index of smaller element
          swap arr[i] and arr[j]
       }
   }
   swap arr[i + 1] and arr[high])
   return (i + 1)
}
```

Brute Force

A brute force approach is an approach that finds all the possible solutions to find a satisfactory solution to a given problem. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found.Such an algorithm can be of two types:

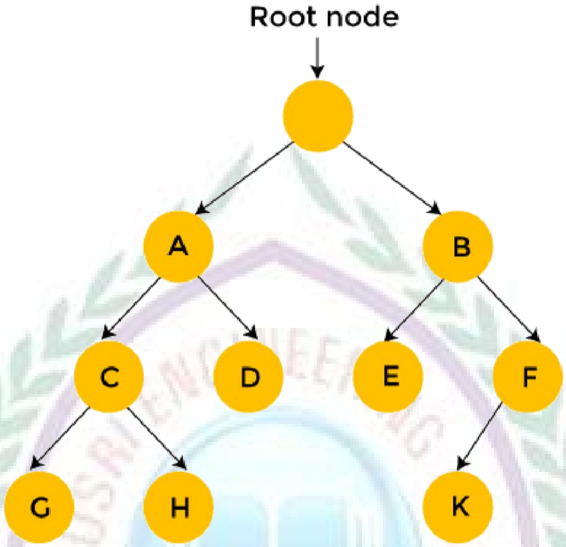
Optimizing: In this case, the best solution is found. To find the best solution, it may either find all the possible solutions to find the best solution or if the value of the best solution is known, it stops finding when the best solution is found. For example: Finding the best path for the travelling salesman problem. Here best path means that travelling all the cities and the cost of travelling should be minimum.

Satisficing: It stops finding the solution as soon as the satisfactory solution is found. Or example, finding the travelling salesman path which is within 10% of optimal.Often Brute force algorithms require exponential time. Various heuristics and optimization can be used:

Heuristic: A rule of thumb that helps you to decide which possibilities we should look at first.

Optimization: A certain possibilities are eliminated without exploring all of them.

Suppose we have converted the problem in the form of the tree shown as below:



Brute force search considers each and every state of a tree, and the state is represented in the form of a node. As far as the starting position is concerned, we have two choices, i.e., A state and B state. We can either generate state A or state B. In the case of B state, we have two states, i.e., state E and F.

In the case of brute force search, each state is considered one by one. As we can observe in the above tree that the brute force search takes 12 steps to find the solution.

On the other hand, backtracking, which uses Depth-First search, considers the below states only when the state provides a feasible solution. Consider the above tree, start from the root node, then move to node A and then node C. If node C does not provide the feasible solution, then there is no point in considering the states G and H. We backtrack from node C to node A. Then, we move from node A to node D. Since node D does not provide the feasible solution, we discard this state and backtrack from node D to node A.

We move to node B, then we move from node B to node E. We move from node E to node K; Since k is a solution, so it takes 10 steps to find the solution. In this way, we eliminate a greater number of states in a single iteration. Therefore, we can say that backtracking is faster and more efficient than the brute force approach.

Advantages of a brute-force algorithm

The following are the advantages of the brute-force algorithm:

This algorithm finds all the possible solutions, and it also guarantees that it finds the correct solution to a problem.

This type of algorithm is applicable to a wide range of domains.

It is mainly used for solving simpler and small problems.

It can be considered a comparison benchmark to solve a simple problem and does not require any particular domain knowledge.

Disadvantages of a brute-force algorithm
The following are the disadvantages of the brute-force algorithm:

It is an inefficient algorithm as it requires solving each and every state.

It is a very slow algorithm to find the correct solution as it solves each state without considering whether the solution is feasible or not.

The brute force algorithm is neither constructive nor creative as compared to other algorithms.

Knapsack Problem

You are given the following-

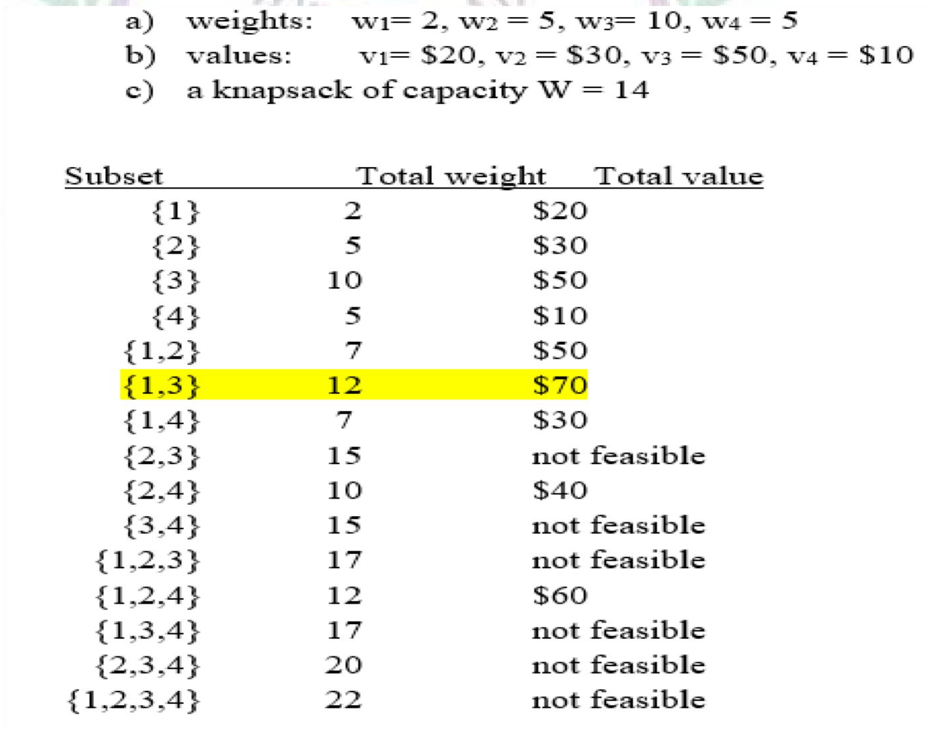
- A knapsack (kind of shoulder bag) with limited weight capacity.
- Few items each having some weight and profit. **The problem states-**

Which items should be placed into the knapsack such that-

- The profit obtained by putting the items into the knapsack is maximum.
- And the weight limit of the knapsack does not exceed. **Knapsack Problem Variants-**

Knapsack problem has the following two variants-

1. Fractional Knapsack Problem
2. 0/1 Knapsack Problem



a) weights: $w_1 = 2$, $w_2 = 5$, $w_3 = 10$, $w_4 = 5$
b) values: $v_1 = \$20$, $v_2 = \$30$, $v_3 = \$50$, $v_4 = \$10$
c) a knapsack of capacity W = 14

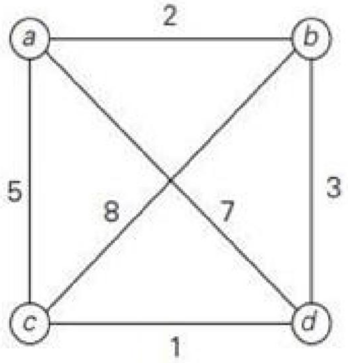
Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	not feasible
{2,4}	10	$40
{3,4}	15	not feasible
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Traveling salesman problem

The travelling salesman problem is a graph computational problem where the salesman needs to visit all cities (represented using nodes in a graph) in a list just once and the distances (represented using edges in the graph) between all these cities are known. The solution that is needed to be found for this problem is the shortest possible route in which the salesman visits all the cities and returns to the origin city.

Consider the following graph.

Sales person has to start from a vertex and reach vertex a with minimum cost using Brute force approach.



Tour	Length	
a ---> b ---> c ---> d ---> a	$l = 2 + 8 + 1 + 7 = 18$	
a ---> b ---> d ---> c ---> a	$l = 2 + 3 + 1 + 5 = 11$	optimal
a ---> c ---> b ---> d ---> a	$l = 5 + 8 + 3 + 7 = 23$	
a ---> c ---> d ---> b ---> a	$l = 5 + 1 + 3 + 2 = 11$	optimal
a ---> d ---> b ---> c ---> a	$l = 7 + 3 + 8 + 5 = 23$	
a ---> d ---> c ---> b ---> a	$l = 7 + 1 + 8 + 2 = 18$	

Convex-Hull

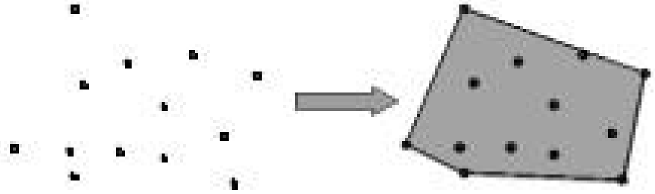
Suppose we have a set of points. We have to make a polygon by taking less amount of points, that will cover all given points

The convex hull of a set of points in S is the boundary of the smallest convex region that contain all the points of S inside it or on its boundary. Definition:

Given a finite set of points *P*={*p*1,…,*pn*}, the convex hull of *P* is the smallest convex set *C* such that *P*Ì*C*.

Definition:

Given a finite set of points *P*={*p*1,…,*pn*}, the convex hull of *P* is the smallest convex set *C* such that *P*Ì*C*.



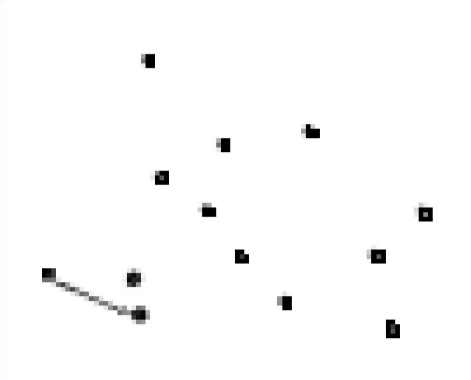
There are many algorithms for computing the convex hull:

- Brute Force O(n^3)
- Gift Wrapping (Jarvis March) O(nlogn),O(n^2)
- Quick hull O(nlogn)
- Divide and Conquer O(nlogn)

Gift Wrapping (Jarvis March) Key Idea:

Iteratively growing edges of the convex hull, we want to turn as little as possible.

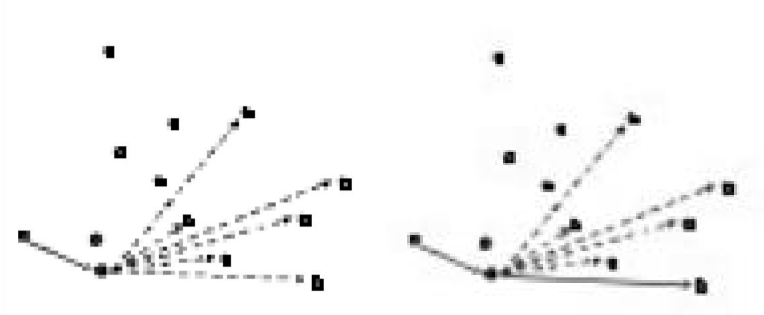
• Given a directed edge on the hull



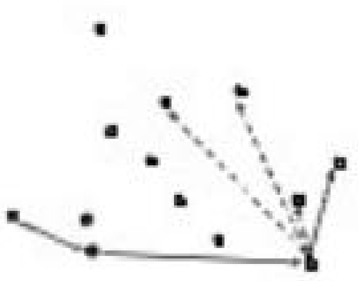
Key Idea:

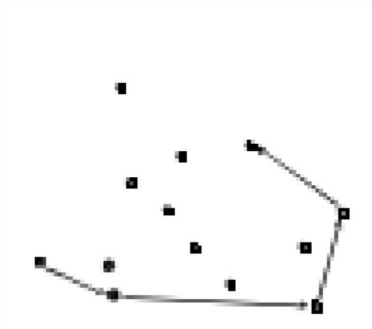
Iteratively growing edges of the convex hull, we want to turn as little as possible.

Given a directed edge on the hull… • Of all the vertices the next edge can connect to Choose the one which turns least.

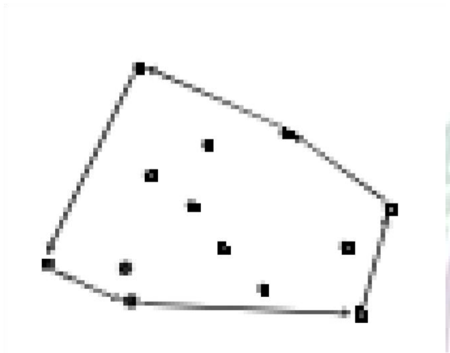


Repeat the process





The final Convex hull is



UNIT–III

Greedy Method: Knapsack problem, Minimum spanning trees, Single source shortest path, Job sequencing with deadlines, Optimal storage on tapes, Optimal merge pattern
Dynamic programming method: All pairs shortest paths, Optimal binary search tress, 0/1 Knapsack problem, Reliability design, Traveling salesman problem**.**
Course Outcome:

- Describe the Greedy paradigm & dynamic Programming and explain when an algorithmic design situation calls for it.

Greedy Method:

The simplest and straight forward approach is the Greedy method. In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases.

Components of Greedy Algorithm:

Greedy algorithms have the following five components −
A candidate set − A solution is created from this set.
A selection function − Used to choose the best candidate to be added to the solution.
A feasibility function − Used to determine whether a candidate can be used to contribute to the solution.
An objective function − Used to assign a value to a solution or a partial solution.
A solution function − Used to indicate whether a complete solution has been reached.

Knapsack:

The knapsack problem states that − given a set of items, holding weights and profit values, one must determine the subset of the items to be added in a knapsack such that, the total weight of the items must not exceed the limit of the knapsack and its total profit value is maximum.

It is one of the most popular problems that take greedy approach to be solved. It is called as the Fractional Knapsack Problem.

Algorithm

1) Consider all the items with their weights and profits mentioned respectively.
2) Calculate Pi/Wi of all the items and sort the items in descending order based on their Pi/Wi values.
3) Without exceeding the limit, add the items into the knapsack.
4) If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.
5) Hence, giving it the name fractional knapsack problem.

Examples

For the given set of items and the knapsack capacity of 10 kg, find the subset of the items to be added in the knapsack such that the profit is maximum.

Items	1	2	3	4	5
Weights (in kg)	3	3	2	5	1
Profits	10	15	10	12	8

Solution

Step 1

Given, n = 5

Wi = {3, 3, 2, 5, 1}

Pi = {10, 15, 10, 12, 8}

Calculate Pi/Wi for all the items:

Items	1	2	3	4	5
Weights (in kg)	3	3	2	5	1
Profits	10	15	10	20	8
P_i/W_i	3.3	5	5	4	8

Step 2

Arrange all the items in descending order based on Pi/Wi.

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
P_i/W_i	8	5	5	4	3.3

Step 3

Without exceeding the knapsack capacity, insert the items in the knapsack with maximum profit.

Knapsack = {5, 2, 3}

However, the knapsack can still hold 4 kg weight, but the next item having 5 kg weight will exceed the capacity. Therefore, only 4 kg weight of the 5 kg will be added in the knapsack

Items	5	2	3	4	1
Weights (in kg)	1	3	2	5	3
Profits	8	15	10	20	10
Knapsack	1	1	1	4/5	0

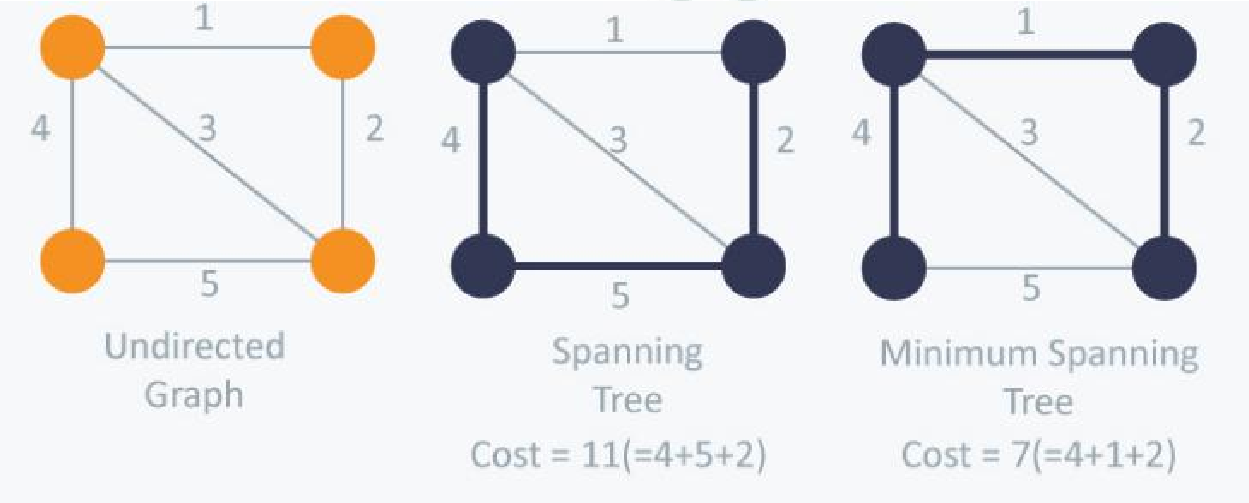
Hence, the knapsack holds the weights = [(1 * 1) + (1 * 3) + (1 * 2) + (4/5 * 5)] = 10, with maximum profit of [(1 * 8) + (1 * 15) + (1 * 10) + (4/5 * 20)] = 37.

Spanning Tree:

Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G)

Minimum Spanning Tree:

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.



There are two famous algorithms for finding the Minimum Spanning Tree:.

>Prim's Minimal Spanning Tree.

>Kruskal's Minimal Spanning Tree.

Prim's Minimal Spanning Tree:

This algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a subgraph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm, similar to any shortest path algorithm, begins from a vertex that is set as a root and walks through all the vertices in the graph by determining the least cost adjacent edges.

Prim's Algorithm

To execute the prim's algorithm, the inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S. A minimum spanning tree of graph G is obtained as an output.

Algorithm

Declare an array visited[] to store the visited vertices and firstly, add the arbitrary root, say S, to the visited array.

Check whether the adjacent vertices of the last visited vertex are present in the visited[] array or not.

If the vertices are not in the visited[] array, compare the cost of edges and add the least cost edge to the output spanning tree.

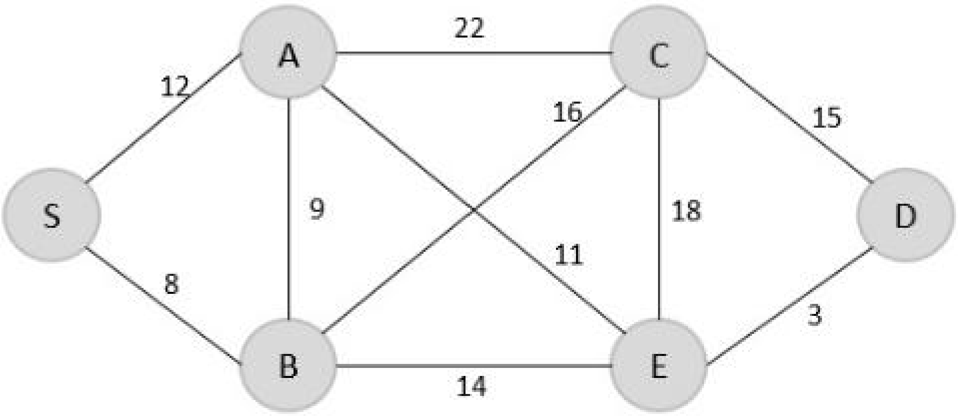
The adjacent unvisited vertex with the least cost edge is added into the visited[] array and the least cost edge is added to the minimum spanning tree output.

Steps 2 and 4 are repeated for all the unvisited vertices in the graph to obtain the full minimum spanning tree output for the given graph.

Calculate the cost of the minimum spanning tree obtained.

Examples

Find the minimum spanning tree using prim's method (greedy approach) for the graph given below with S as the arbitrary root.



Solution

Step 1

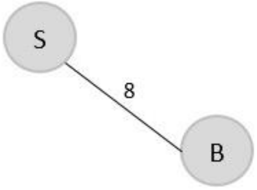
Create a visited array to store all the visited vertices into it.

V = { }

The arbitrary root is mentioned to be S, so among all the edges that are connected to S we need to find the least cost edge.

S → B = 8

V = {S, B}



Step 2

Since B is the last visited, check for the least cost edge that is connected to the vertex B.

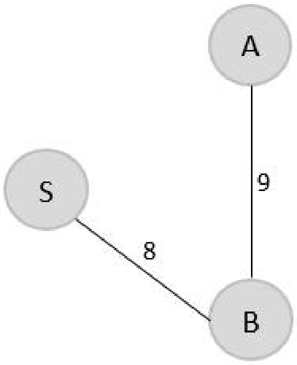
B → A = 9

B → C = 16

B → E = 14

Hence, B → A is the edge added to the spanning tree.

V = {S, B, A}



Step 3

Since A is the last visited, check for the least cost edge that is connected to the vertex A.

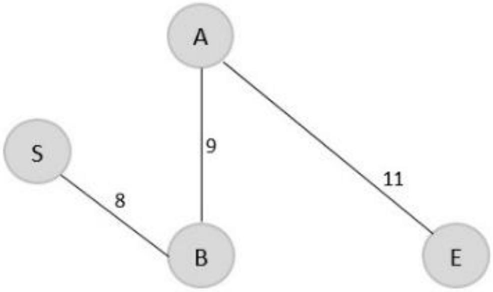
A → C = 22

A → B = 9

A → E = 11

But A → B is already in the spanning tree, check for the next least cost edge. Hence, A → E is added to the spanning tree.

V = {S, B, A, E}



Step 4

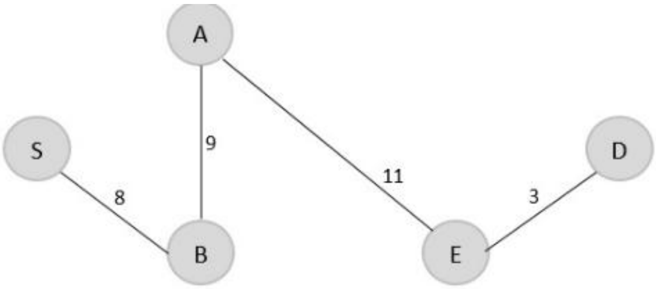
Since E is the last visited, check for the least cost edge that is connected to the vertex E.

E → C = 18

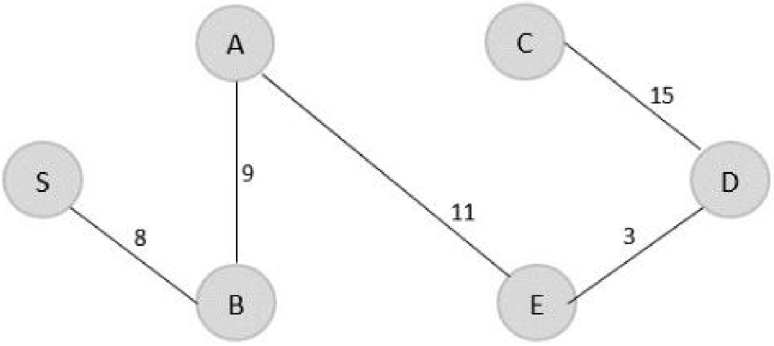
E → D = 3

Therefore, E → D is added to the spanning tree.

V = {S, B, A, E, D}



Step 5
Since D is the last visited, check for the least cost edge that is connected to the vertex D.
D → C = 15
E → D = 3
Therefore, D → C is added to the spanning tree.
V = {S, B, A, E, D, C}



The minimum spanning tree is obtained with the minimum cost = 46

Kruskal's Minimal Spanning Tree:
Kruskal's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a subgraph that connects all the vertices present in the main graph with the least possible edges and minimum cost (sum of the weights assigned to each edge).

The algorithm first starts from the forest – which is defined as a subgraph containing only vertices of the main graph – of the graph, adding the least cost edges later until the minimum spanning tree is created without forming cycles in the graph.

Kruskal's Algorithm
The inputs taken by the kruskal's algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S and the minimum spanning tree of graph G is obtained as an output.

Algorithm

- Sort all the edges in the graph in an ascending order and store it in an array edge[].

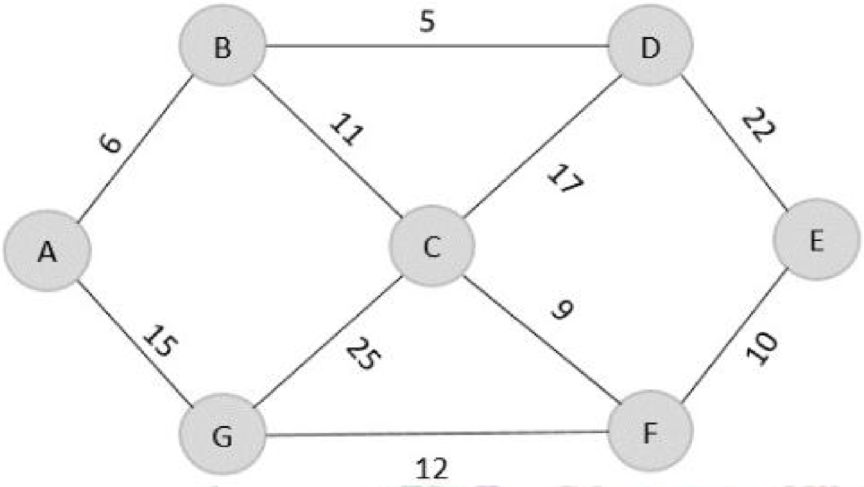
Construct the forest of the graph on a plane with all the vertices in it.

- Select the least cost edge from the edge[] array and add it into the forest of the graph. Mark the vertices visited by adding them into the visited[] array.
- Repeat the steps 2 and 3 until all the vertices are visited without having any cycles forming in the graph

- ➢ When all the vertices are visited, the minimum spanning tree is formed.
- ➢ Calculate the minimum cost of the output spanning tree formed.

Examples

Construct a minimum spanning tree using kruskal’s algorithm for the graph given below –

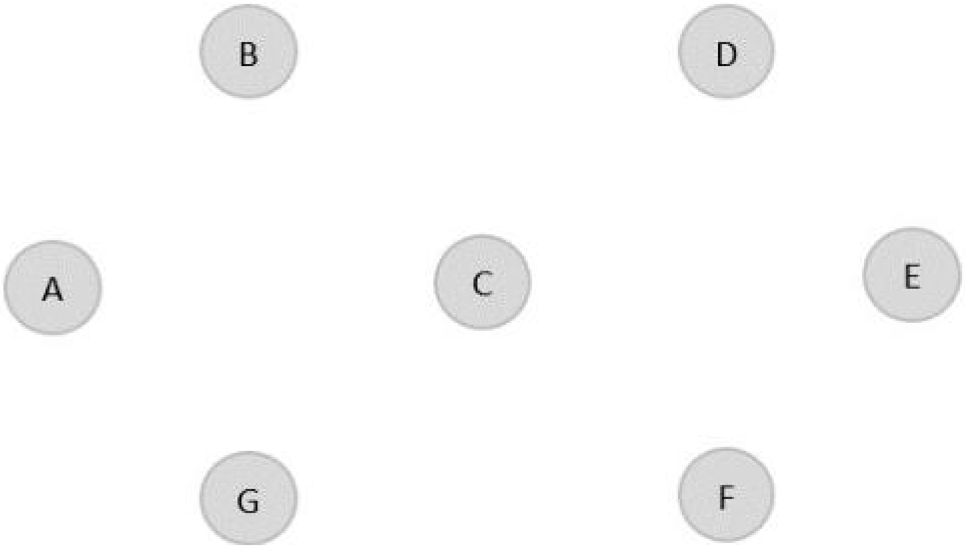


Solution

As the first step, sort all the edges in the given graph in an ascending order and store the values in an array.

Edge	B→D	A→B	C→F	F→E	B→C	G→F	A→G	C→D	D→E	C→G
Cost	5	6	9	10	11	12	15	17	22	25

Then, construct a forest of the given graph on a single plane.

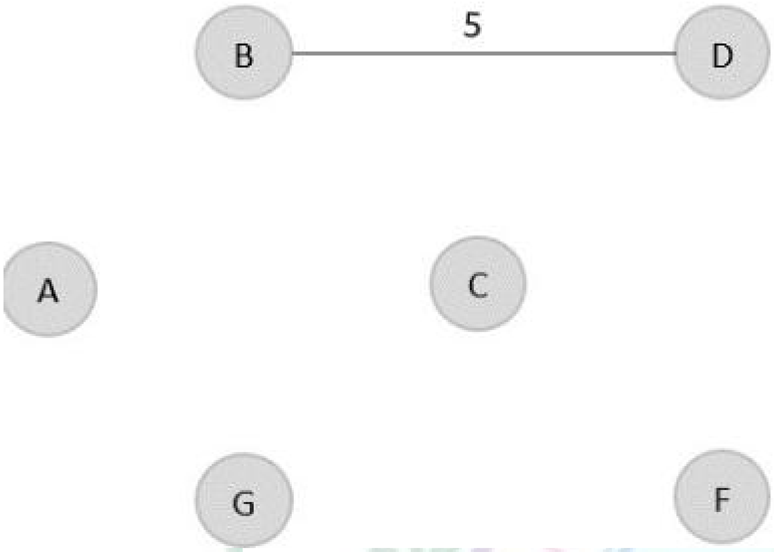


From the list of sorted edge costs, select the least cost edge and add it onto the forest in output graph.

B → D = 5

Minimum cost = 5

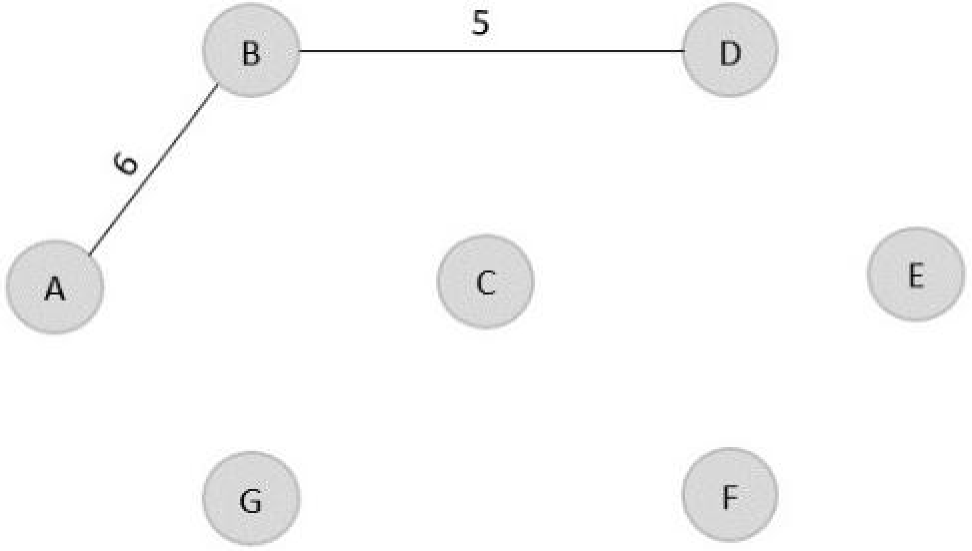
Visited array, v = {B, D}



Similarly, the next least cost edge is B → A = 6; so we add it onto the output graph.

Minimum cost = 5 + 6 = 11

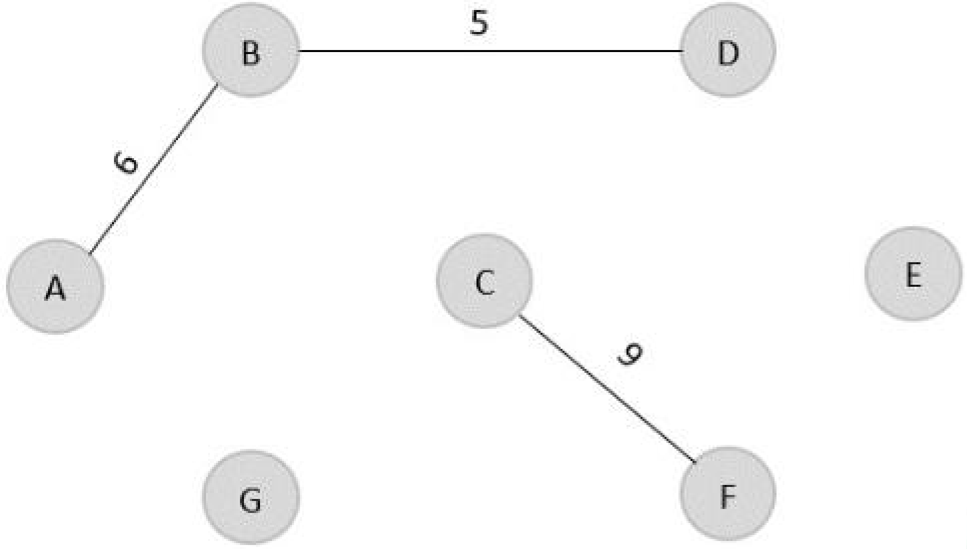
Visited array, v = {B, D, A}



The next least cost edge is C → F = 9; add it onto the output graph.

Minimum Cost = 5 + 6 + 9 = 20

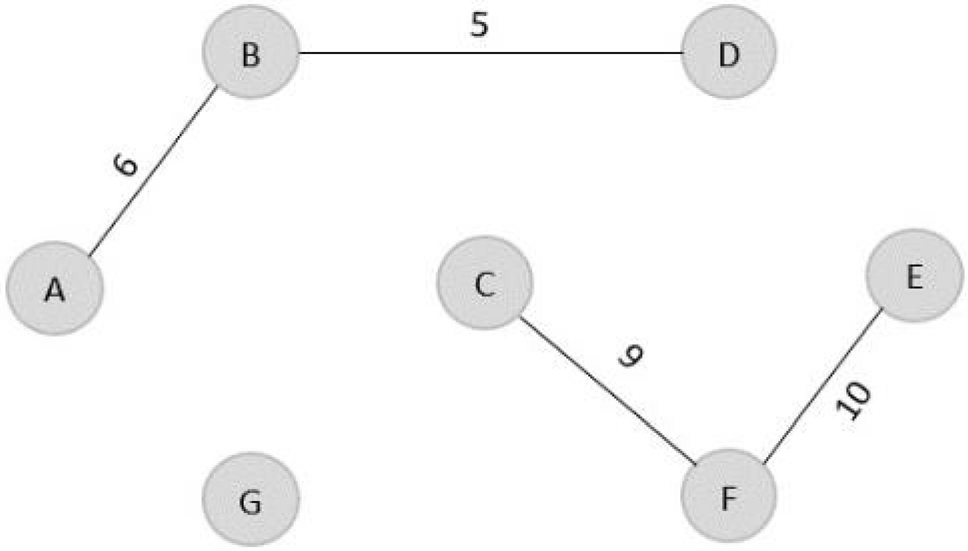
Visited array, v = {B, D, A, C, F}



The next edge to be added onto the output graph is F → E = 10.

Minimum Cost = 5 + 6 + 9 + 10 = 30

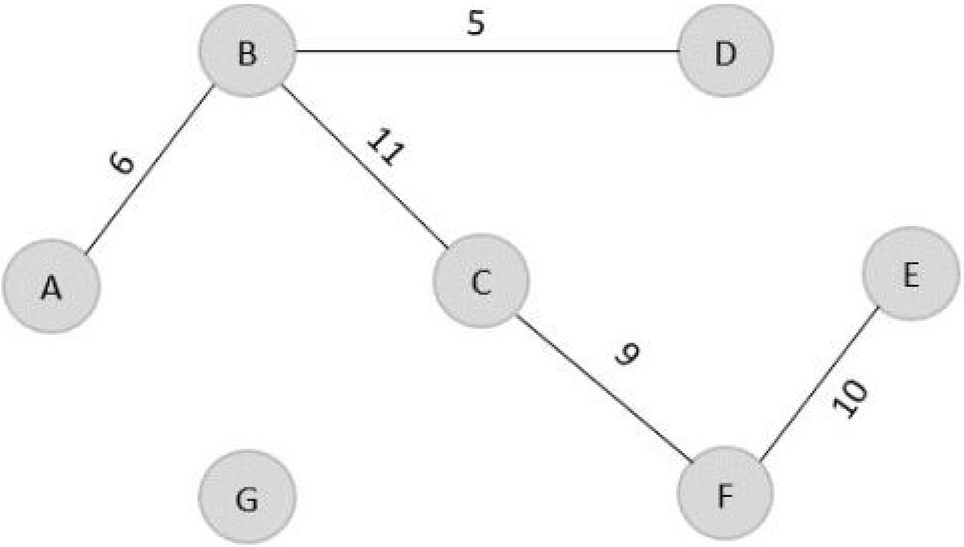
Visited array, v = {B, D, A, C, F, E}



The next edge from the least cost array is B → C = 11, hence we add it in the output graph.

Minimum cost = 5 + 6 + 9 + 10 + 11 = 41

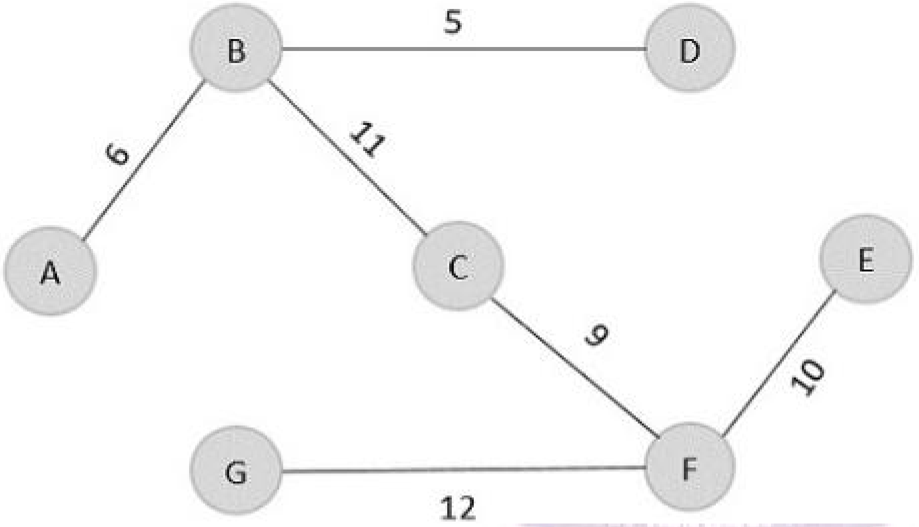
Visited array, v = {B, D, A, C, F, E}



The last edge from the least cost array to be added in the output graph is F → G = 12.

Minimum cost = 5 + 6 + 9 + 10 + 11 + 12 = 53

Visited array, v = {B, D, A, C, F, E, G}



The obtained result is the minimum spanning tree of the given graph with cost = 53.

Single source shortest path:

The Single-Pair Shortest Path (SPSP) problem consists of finding the shortest path between a single pair of vertices. This problem is mostly solved using Dijkstra, though in this case a single result is kept and other shortest paths are discarded.

Dijkstra’s Shortest Path Algorithm:

Dijkstra’s Algorithm

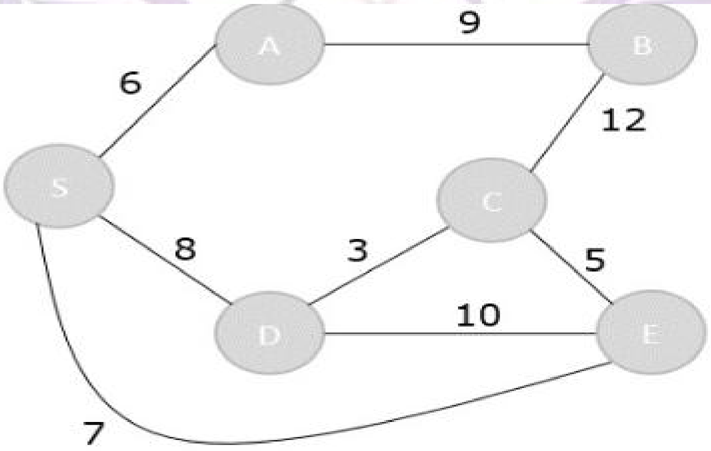
The dijkstra’s algorithm is designed to find the shortest path between two vertices of a graph. These two vertices could either be adjacent or the farthest points in the graph. The algorithm starts from the source. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S. And the output is the shortest path spanning tree**.**

Algorithm

- Declare two arrays − distance[] to store the distances from the source vertex to the other vertices in graph and visited[] to store the visited vertices.
- Set distance[S] to ‘0’ and distance[v] = ∞, where v represents all the other vertices in the graph.
- Add S to the visited[] array and find the adjacent vertices of S with the minimum distance.
- The adjacent vertex to S, say A, has the minimum distance and is not in the visited array yet. A is picked and added to the visited array and the distance of A is changed from ∞ to the assigned distance of A, say d1, where d1 < ∞.
- Repeat the process for the adjacent vertices of the visited vertices until the shortest path spanning tree is formed.

Examples

To understand the dijkstra’s concept better, let us analyze the algorithm with the help of an example graph −



Step 1

Initialize the distances of all the vertices as ∞, except the source node S.

Vertex	S	A	B	C	D
Distance	0	∞	∞	∞	∞

Now that the source vertex S is visited, add it into the visited array.

visited = {S}

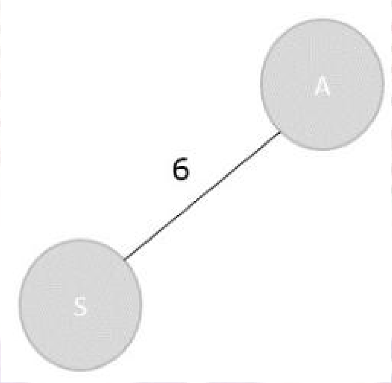
Step 2

The vertex S has three adjacent vertices with various distances and the vertex with minimum distance among them all is A. Hence, A is visited and the dist[A] is changed from ∞ to 6.

S → A = 6
S → D = 8
S → E = 7

Vertex	S	A	B	C	D	E
Distance	0	6	∞	∞	8	7

Visited = {S, A}

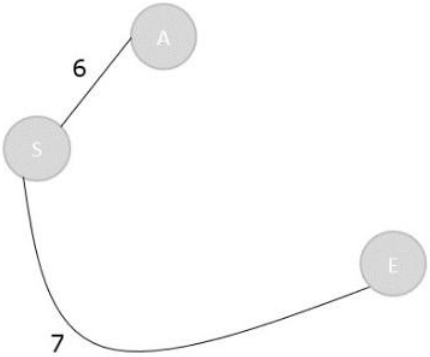


Step 3

There are two vertices visited in the visited array, therefore, the adjacent vertices must be checked for both the visited vertices.
Vertex S has two more adjacent vertices to be visited yet: D and E. Vertex A has one adjacent vertex B.
Calculate the distances from S to D, E, B and select the minimum distance −
S → D = 8 and S → E = 7.
S → B = S → A + A → B = 6 + 9 = 15

Vertex	S	A	B	C	D	E
Distance	0	6	15	∞	8	7

Visited = {S, A, E}

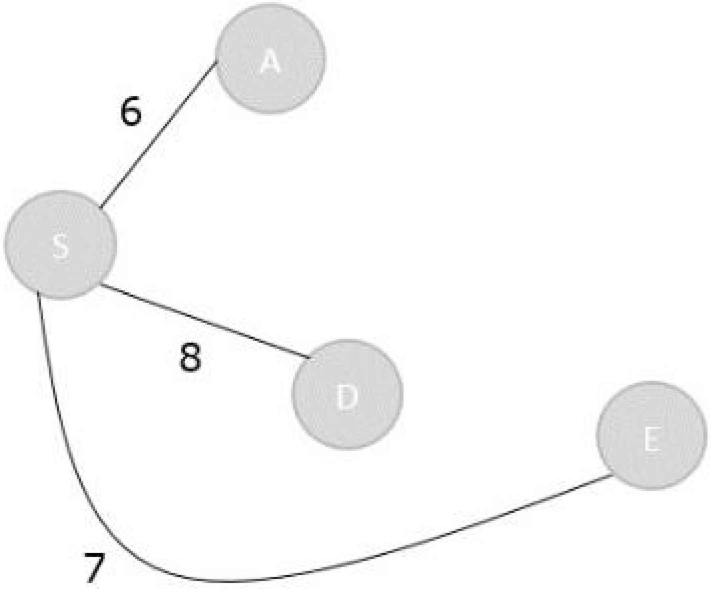


Step 4

Calculate the distances of the adjacent vertices – S, A, E – of all the visited arrays and select the vertex with minimum distance.
S → D = 8
S → B = 15
S → C = S → E + E → C = 7 + 5 = 12

Vertex	S	A	B	C	D	E
Distance	0	6	15	12	8	7

Visited = {S, A, E, D}

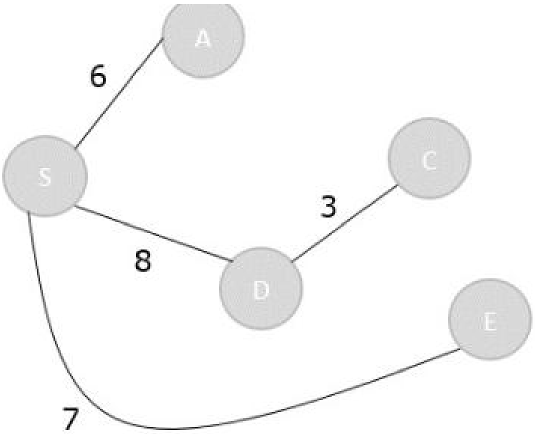


Step 5

Recalculate the distances of unvisited vertices and if the distances minimum than existing distance is found, replace the value in the distance array.
S → C = S → E + E → C = 7 + 5 = 12
S → C = S → D + D → C = 8 + 3 = 11
dist[C] = minimum (12, 11) = 11
S → B = S → A + A → B = 6 + 9 = 15
S → B = S → D + D → C + C → B = 8 + 3 + 12 = 23
dist[B] = minimum (15,23) = 15

Vertex	S	A	B	C	D	E
Distance	0	6	15	11	8	7

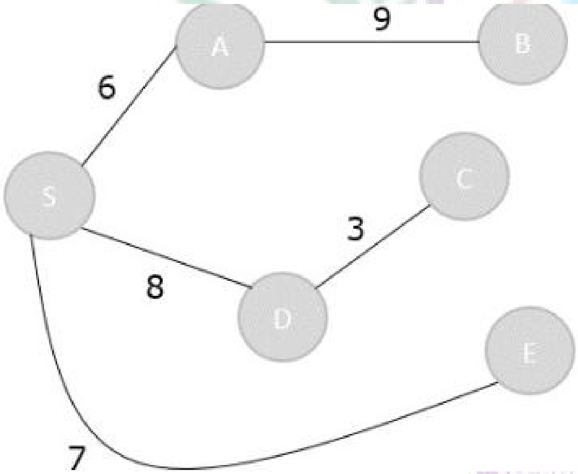
Visited = { S, A, E, D, C}



Step 6

The remaining unvisited vertex in the graph is B with the minimum distance 15, is added to the output spanning tree.

Visited = {S, A, E, D, C, B}



The shortest path spanning tree is obtained as an output using the dijkstra’s algorithm

Job Sequencing with Deadline:

Job scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits.

The greedy approach of the job scheduling algorithm states that, “Given ‘n’ number of jobs with a starting time and ending time, they need to be scheduled in such a way that maximum profit is received within the maximum deadline”.

Job Scheduling Algorithm

Set of jobs with deadlines and profits are taken as an input with the job scheduling algorithm and scheduled subset of jobs with maximum profit are obtained as the final output.

Algorithm

Find the maximum deadline value from the input set of jobs.

Once, the deadline is decided, arrange the jobs in descending order of their profits.

Selects the jobs with highest profits, their time periods not exceeding the maximum deadline.

The selected set of jobs are the output.

Examples

Consider the following tasks with their deadlines and profits. Schedule the tasks in such a way that they produce maximum profit after being executed –

S. No.	1	2	3	4	5
Jobs	J1	J2	J3	J4	J5
Deadlines	2	2	1	3	4
Profits	20	60	40	100	80

Step 1

Find the maximum deadline value, dm, from the deadlines given.

dm = 4.

Step 2

Arrange the jobs in descending order of their profits.

S. No.	1	2	3	4	5
Jobs	J4	J5	J2	J3	J1
Deadlines	3	4	2	1	2
Profits	100	80	60	40	20

The maximum deadline, dm, is 4. Therefore, all the tasks must end before 4.

Choose the job with highest profit, J4. It takes up 3 parts of the maximum deadline.

Therefore, the next job must have the time period 1.

Total Profit = 100.

Step 3

The next job with highest profit is J5. But the time taken by J5 is 4, which exceeds the deadline by 3. Therefore, it cannot be added to the output set.

Step 4

The next job with highest profit is J2. The time taken by J5 is 2, which also exceeds the deadline by 1. Therefore, it cannot be added to the output set.

Step 5

The next job with higher profit is J3. The time taken by J3 is 1, which does not exceed the given deadline. Therefore, J3 is added to the output set.

Total Profit: 100 + 40 = 140

Step 6

Since, the maximum deadline is met, the algorithm comes to an end. The output set of jobs scheduled within the deadline are {J4, J3} with the maximum profit of 140.

Optimal Storage on Tapes:

Optimal Storage on Tapes is one of the applicationof the Greedy Method.

- The objective is to find the Optimal retrieval timefor accessing programs that are stored on tape.Description
- There are n programs that are to be stored on a computertape of length L.
- Associated with each program i is a length l;
- Clearly, all programs can be stored on the tape if and only ifthe sum of the lengths of the programs is at most L.
- We shall assume that whenever a program is to beretrieved from this tape, the tape is initiallypositioned atthe front.
- Hence' if the programs are stored in the order I=i_1, i_2' i_3 …. i_n the time $t_j$needed to retrieve program i $_j$is proportional to l_{ik}.

$$t_j = \sum_{1 \le k \le j} l_{i_k}$$

- If all programs are retrieved equally often then the expected or mean retrieval time (MRT)

$$MRT = 1/n \sum_{1 \le j \le n} t_j$$

is

Example:

Example 1: Let n=3 and $(l_{1,}l_{2,}l_3)$=(5,10,3). There are n!=6 possible orderings. These orderings and their respective D values are:

Ordering I	D(I)
1,2,3	5+5+10+5+10+3=38
1,3,2	5+5+3+5+3+10=31
2,1,3	10+10+5+10+5+3=43
2,3,1	10+10+3+10+3+5=41
3,1,2	3+3+5+3+5+10=29
3,2,1	3+3+10+3+10+5=34

The optimal ordering is 3,1,2

Method

• The greedy method simply requires us to storethe programs in non-decreasing order of theirlengths.

• This ordering (sorting) can be carried out inO(n log n) time using an efficient sortingalgorithm

Optimal Merge Pattern:

Merge a set of sorted files of different lengths into a single sorted file. We need to find an optimal solution, where the resultant file will be generated in minimum time.

If the number of sorted files is given, there are many ways to merge them into a single sorted file. This merge can be performed pair wise. Hence, this type of merging is called a 2-way merge patterns.

As different pairings require different amounts of time, in this strategy we want to determine an optimal way of merging many files together. At each step, the two shortest sequences are merged.

To merge a p-record file and a q-record file requires possibly p + q record moves, the obvious choice being, merge the two smallest files together at each step.

Two-way merge patterns can be represented by binary merge trees. Let us consider a set of n sorted files {f1, f2, f3, …, fn}. Initially, each element of this is considered as a single node binary tree. To find this optimal solution, the following algorithm is used.

Algorithm: TREE (n)

```
for i := 1 to n – 1 do
  declare new node
```

```
node.leftchild := least (list)
node.rightchild := least (list)
node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)
   insert (list, node);
return least (list);
```

At the end of this algorithm, the weight of the root node represents the optimal cost.

Example

Let us consider the given files, f1, f2, f3, f4 and f5 with 20, 30, 10, 5 and 30 number of elements respectively.

If merge operations are performed according to the provided sequence, then

M1 = merge f1 and f2 => 20 + 30 = 50

M2 = merge M1 and f3 => 50 + 10 = 60

M3 = merge M2 and f4 => 60 + 5 = 65

M4 = merge M3 and f5 => 65 + 30 = 95

Hence, the total number of operations is

50 + 60 + 65 + 95 = 270

Now, the question arises is there any better solution?

Sorting the numbers according to their size in an ascending order, we get the following sequence

f4, f3, f1, f2, f5

Hence, merge operations can be performed on this sequence

M1 = merge f4 and f3 => 5 + 10 = 15

M2 = merge M1 and f1 => 15 + 20 = 35

M3 = merge M2 and f2 => 35 + 30 = 65

M4 = merge M3 and f5 => 65 + 30 = 95

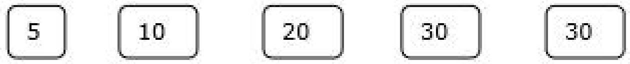
Therefore, the total number of operations is

15 + 35 + 65 + 95 = 210

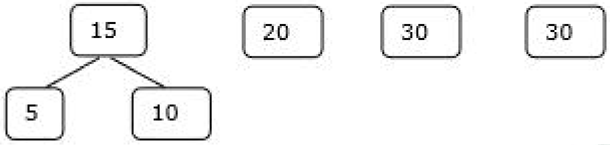
Obviously, this is better than the previous one.

In this context, we are now going to solve the problem using this algorithm.

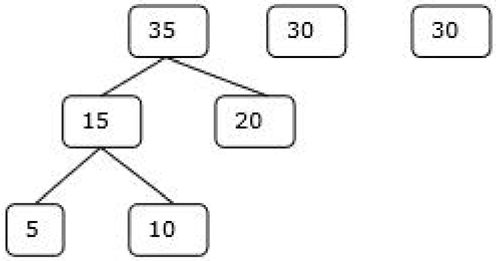
Initial Set:



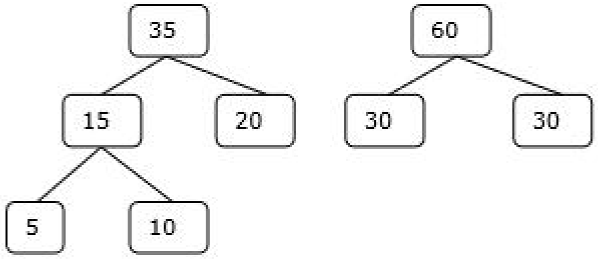
Step 1



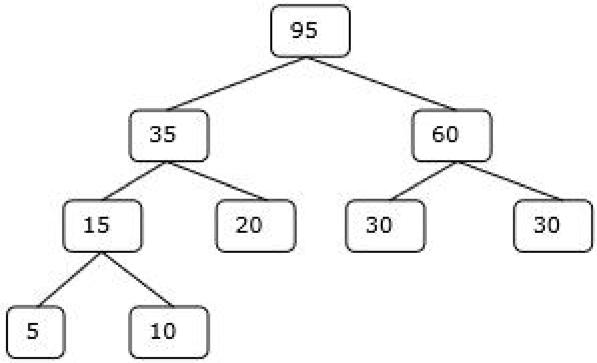
Step 2:



Step 3:



Step 4:



Hence, the solution takes 15 + 35 + 60 + 95 = 205 number of comparisons.

Dynamic Programming:

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Mostly, dynamic programming algorithms are used for solving optimization problems. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best optimal final solution.

All-Pairs Shortest Paths:

The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

Floyd-Warshall algorithm works on both directed and undirected weighted graphs unless these graphs do not contain any negative cycles in them. By negative cycles, it is meant that the sum of all the edges in the graph must not lead to a negative number.

Since, the algorithm deals with overlapping sub-problems – the path found by the vertices acting as pivot are stored for solving the next steps – it uses the dynamic programming approach.

Floyd-Warshall algorithm is one of the methods in All-pairs shortest path algorithms and it is solved using the Adjacency Matrix representation of graphs.

Floyd-Warshall Algorithm

Consider a graph, G = {V, E} where V is the set of all vertices present in the graph and E is the set of all the edges in the graph. The graph, G, is represented in the form of an adjacency matrix, A, that contains all the weights of every edge connecting two vertices.

Algorithm

Step 1 − Construct an adjacency matrix A with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as ∞.

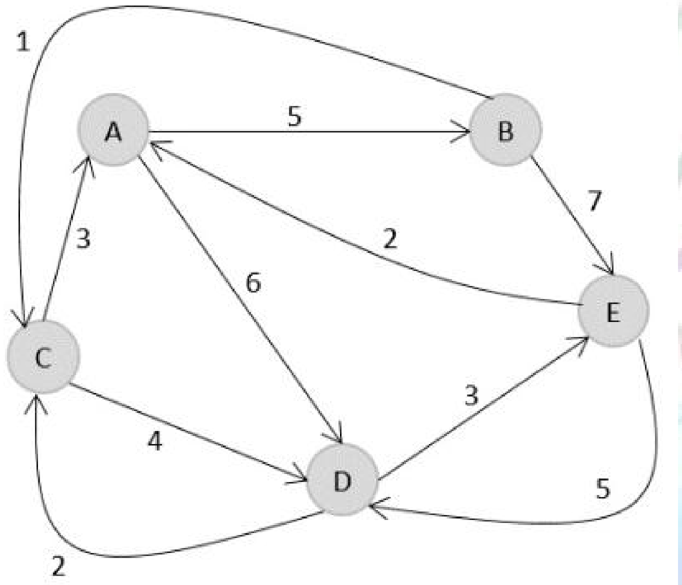
Step 2 − Derive another adjacency matrix A1 from A keeping the first row and first column of the original adjacency matrix intact in A1. And for the remaining values, say A1[i,j], if A[i,j]>A[i,k]+A[k,j] then replace A1[i,j] with A[i,k]+A[k,j]. Otherwise, do not change the values. Here, in this step, k = 1 (first vertex acting as pivot).

Step 3 − Repeat Step 2 for all the vertices in the graph by changing the k value for every pivot vertex until the final matrix is achieved.

Step 4 − The final adjacency matrix obtained is the final solution with all the shortest paths.

Example

Consider the following directed weighted graph G = {V, E}. Find the shortest paths between all the vertices of the graphs using the Floyd-Warshall algorithm.



Solution

Step 1

Construct an adjacency matrix A with all the distances as values.

A=0∞3∞250∞∞∞∞102∞6∞405∞7∞30

Step 2

Considering the above adjacency matrix as the input, derive another matrix A0 by keeping only first rows and columns intact. Take k = 1, and replace all the other values by A[i,k]+A[k,j].

A=0∞3∞25∞6∞

A1=0∞3∞2508∞7∞102∞6∞405∞7∞30

Step 3

Considering the above adjacency matrix as the input, derive another matrix A0 by keeping only first rows and columns intact. Take k = 1, and replace all the other values by A[i,k]+A[k,j].

A2=∞508∞71∞7

A2=0∞3∞2508∞7610286∞4051271530

Step 4

Considering the above adjacency matrix as the input, derive another matrix A0 by keeping only first rows and columns intact. Take k = 1, and replace all the other values by A[i,k]+A[k,j].

A3=3861028415

A3=0435250810761028654051271530

Step 5

Considering the above adjacency matrix as the input, derive another matrix A0 by keeping only first rows and columns intact. Take k = 1, and replace all the other values by A[i,k]+A[k,j].

A4=5102654053

A4=04352508107610276540597730

Step 6

Considering the above adjacency matrix as the input, derive another matrix A0 by keeping only first rows and columns intact. Take k = 1, and replace all the other values by A[i,k]+A[k,j].

A5=277597730

A5=04352508107610276540597730

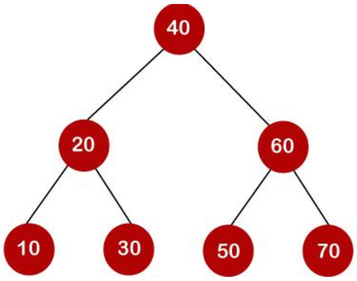
Optimal Binary Search Tree:

In binary search tree, the nodes in the left subtree have lesser value than the root node and the nodes in the right subtree have greater value than the root node.

We know the key values of each node in the tree, and we also know the frequencies of each node in terms of searching means how much time is required to search a node. The frequency and key-value determine the overall cost of searching a node. The cost of searching is a very important factor in various applications. The overall cost of searching a node should be less. The time required to search a node in BST is more than the balanced binary search tree as a balanced binary search tree contains a lesser number of levels than the BST. There is one way that can reduce the cost of a binary search tree is known as an optimal binary search tree.

Let's understand through an example.

If the keys are 10, 20, 30, 40, 50, 60, 70.



In the above tree, all the nodes on the left subtree are smaller than the value of the root node, and all the nodes on the right subtree are larger than the value of the root node. The maximum time required to search a node is equal to the minimum height of the tree, equal to logn.

Now we will see how many binary search trees can be made from the given number of keys.

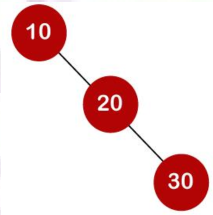
For example: 10, 20, 30 are the keys, and the following are the binary search trees that can be made out from these keys.

The Formula for calculating the number of trees:

$$\frac{{}^{2n}C_n}{n+1}$$

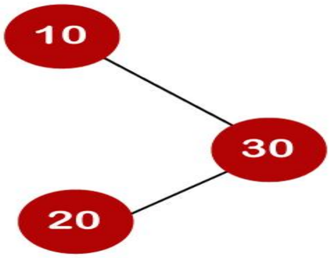
When we use the above formula, then it is found that total 5 number of trees can be created.

The cost required for searching an element depends on the comparisons to be made to search an element. Now, we will calculate the average cost of time of the above binary search trees.



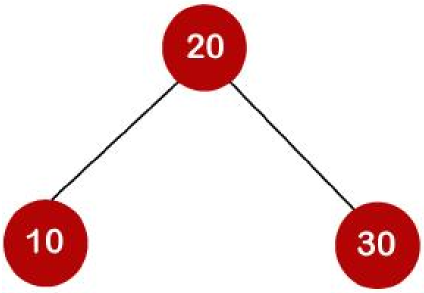
In the above tree, total number of 3 comparisons can be made. The average number of comparisons can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



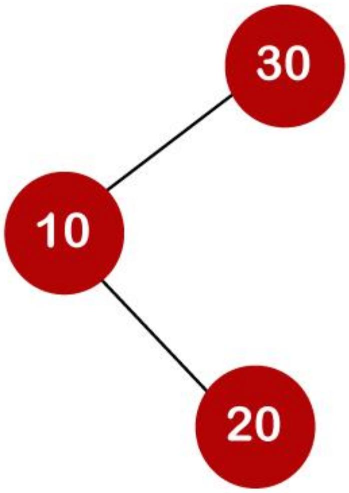
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



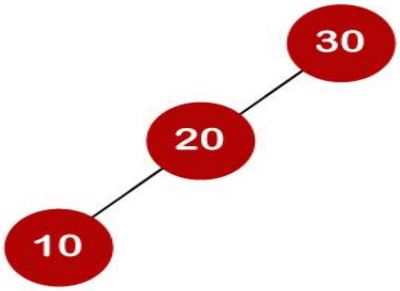
In the above tree, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+2}{3} = 5/3$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$



In the above tree, the total number of comparisons can be made as 3. Therefore, the average number of comparisons that can be made as:

$$average\ number\ of\ comparisons = \frac{1+2+3}{3} = 2$$

In the third case, the number of comparisons is less because the height of the tree is less, so it's a balanced binary search tree.

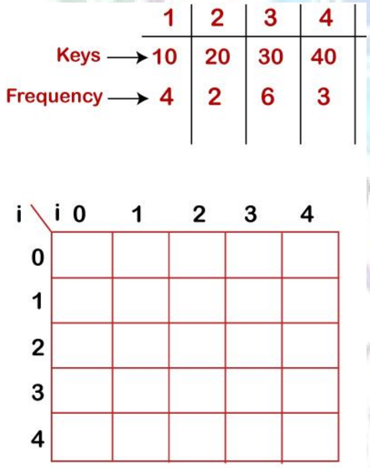
To find the optimal binary search tree, we will determine the frequency of searching a key.

Let's assume that frequencies associated with the keys 10, 20, 30 are 3, 2, 5.

The above trees have different frequencies. The tree with the lowest frequency would be considered the optimal binary search tree. The tree with the frequency 17 is the lowest, so it would be considered as the optimal binary search tree.

Dynamic Approach:

Consider the below table, which contains the keys and frequencies.



	1	2	3	4
Keys →	10	20	30	40
Frequency →	4	2	6	3

i \ i	0	1	2	3	4
0					
1					
2					
3					
4					

irst, we will calculate the values where j-i is equal to zero.

When i=0, j=0, then j-i = 0

When i = 1, j=1, then j-i = 0

When i = 2, j=2, then j-i = 0

When i = 3, j=3, then j-i = 0

When i = 4, j=4, then j-i = 0

Therefore, c[0, 0] = 0, c[1 , 1] = 0, c[2,2] = 0, c[3,3] = 0, c[4,4] = 0

Now we will calculate the values where j-i equal to 1.

When j=1, i=0 then j-i = 1

When j=2, i=1 then j-i = 1

When j=3, i=2 then j-i = 1

When j=4, i=3 then j-i = 1

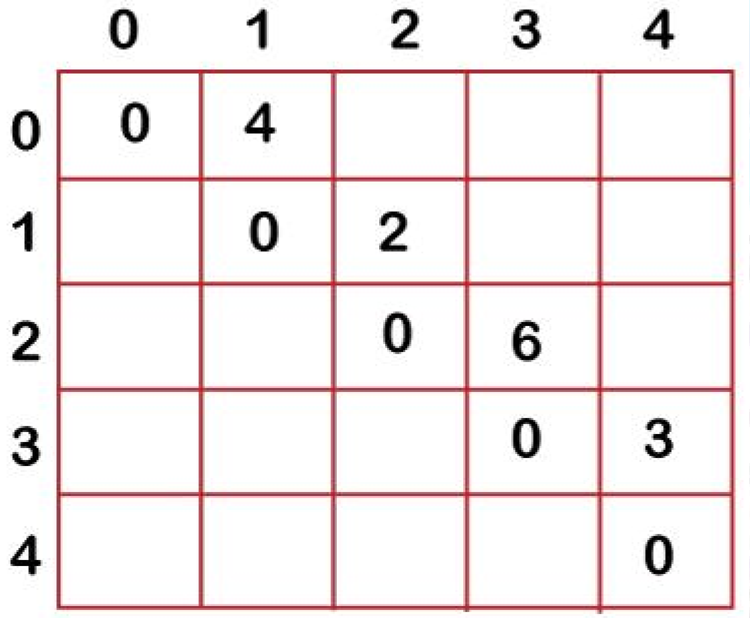
Now to calculate the cost, we will consider only the jth value.

The cost of c[0,1] is 4 (The key is 10, and the cost corresponding to key 10 is 4).

The cost of c[1,2] is 2 (The key is 20, and the cost corresponding to key 20 is 2).

The cost of c[2,3] is 6 (The key is 30, and the cost corresponding to key 30 is 6)

The cost of c[3,4] is 3 (The key is 40, and the cost corresponding to key 40 is 3)



	0	1	2	3	4
0	0	4			
1		0	2		
2			0	6	
3				0	3
4					0

Now we will calculate the values where j-i = 2

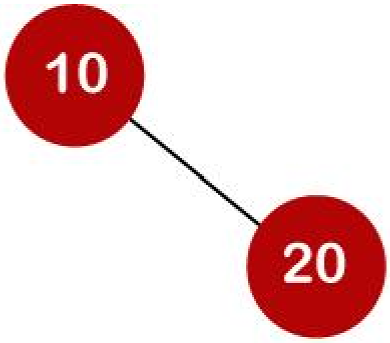
When j=2, i=0 then j-i = 2

When j=3, i=1 then j-i = 2

When j=4, i=2 then j-i = 2

In this case, we will consider two keys.

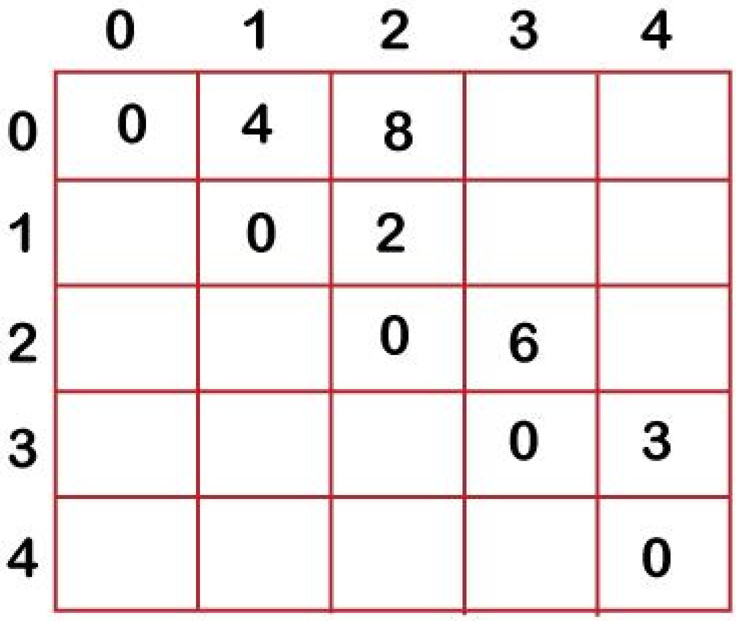
When i=0 and j=2, then keys 10 and 20. There are two possible trees that can be made out from these two keys shown below:



In the first binary tree, cost would be: 4*1 + 2*2 = 8

In the second binary tree, cost would be: 4*2 + 2*1 = 10

The minimum cost is 8; therefore, c[0,2] = 8



	0	1	2	3	4
0	0	4	8		
1		0	2		
2			0	6	
3				0	3
4					0

When i=1 and j=3, then keys 20 and 30. There are two possible trees that can be made out from these two keys shown below:

In the first binary tree, cost would be: 1*2 + 2*6 = 14

In the second binary tree, cost would be: 1*6 + 2*2 = 10

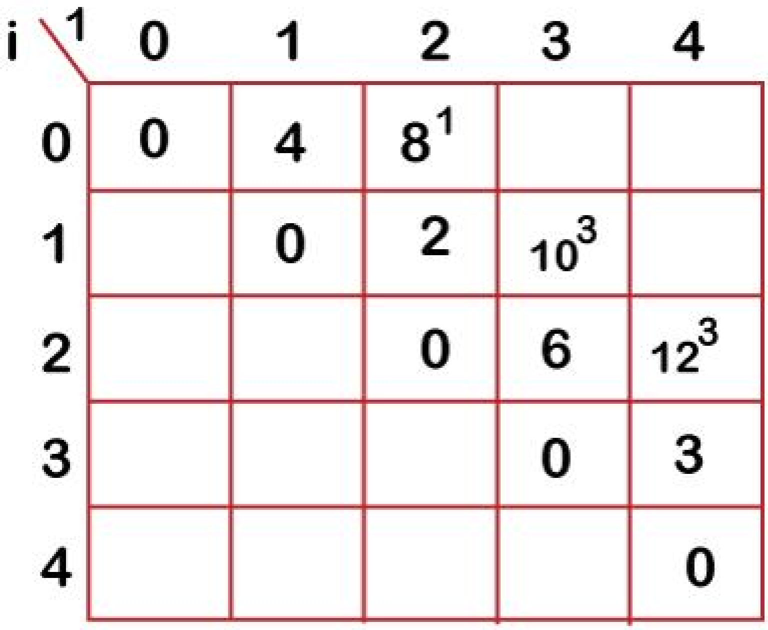
The minimum cost is 10; therefore, c[1,3] = 10

When i=2 and j=4, we will consider the keys at 3 and 4, i.e., 30 and 40. There are two possible trees that can be made out from these two keys shown as below:

In the first binary tree, cost would be: 1*6 + 2*3 = 12

In the second binary tree, cost would be: 1*3 + 2*6 = 15

The minimum cost is 12, therefore, c[2,4] = 12



i \ 1	0	1	2	3	4
0	0	4	8^1		
1		0	2	10^3	
2			0	6	12^3
3				0	3
4					0

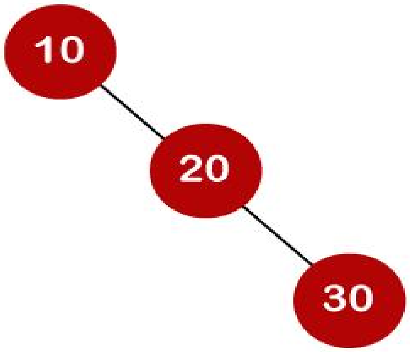
Now we will calculate the values when j-i = 3

When j=3, i=0 then j-i = 3

When j=4, i=1 then j-i = 3

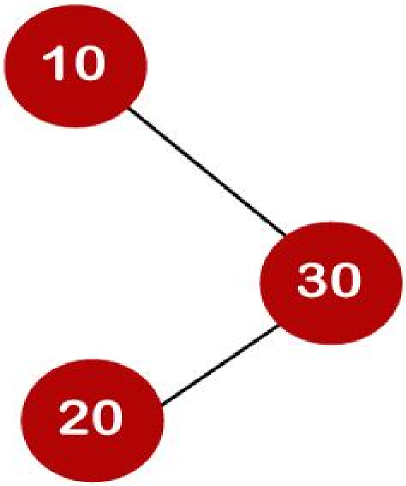
When i=0, j=3 then we will consider three keys, i.e., 10, 20, and 30.

The following are the trees that can be made if 10 is considered as a root node.



In the above tree, 10 is the root node, 20 is the right child of node 10, and 30 is the right child of node 20.

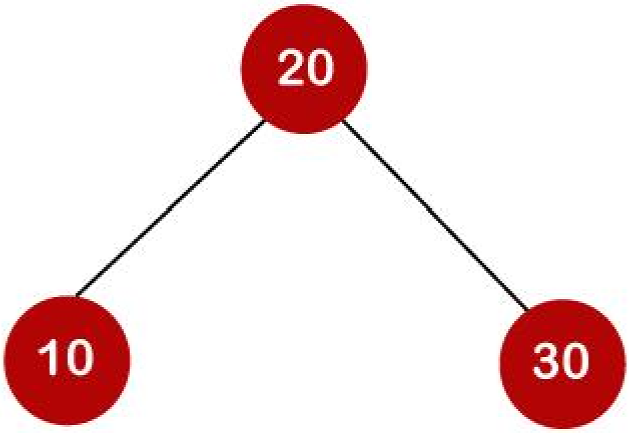
Cost would be: 1*4 + 2*2 + 3*6 = 26



In the above tree, 10 is the root node, 30 is the right child of node 10, and 20 is the left child of node 20.

Cost would be: 1*4 + 2*6 + 3*2 = 22

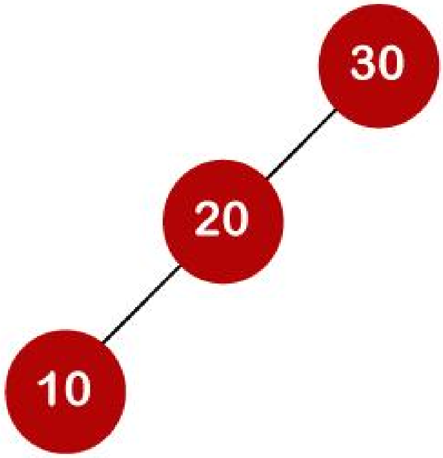
The following tree can be created if 20 is considered as the root node.



In the above tree, 20 is the root node, 30 is the right child of node 20, and 10 is the left child of node 20.

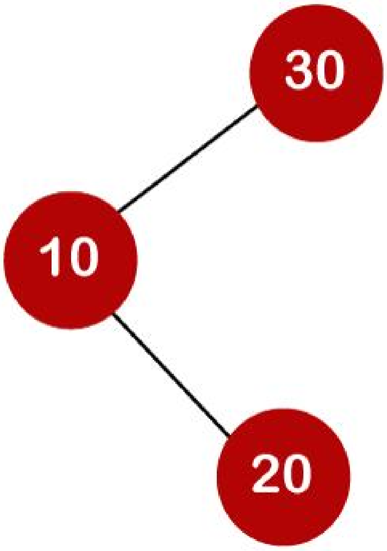
Cost would be: 1*2 + 4*2 + 6*2 = 22

The following are the trees that can be created if 30 is considered as the root node.



In the above tree, 30 is the root node, 20 is the left child of node 30, and 10 is the left child of node 20.

Cost would be: 1*6 + 2*2 + 3*4 = 22



In the above tree, 30 is the root node, 10 is the left child of node 30 and 20 is the right child of node 10.

Cost would be: 1*6 + 2*4 + 3*2 = 20

Therefore, the minimum cost is 20 which is the 3rd root. So, c[0,3] is equal to 20.

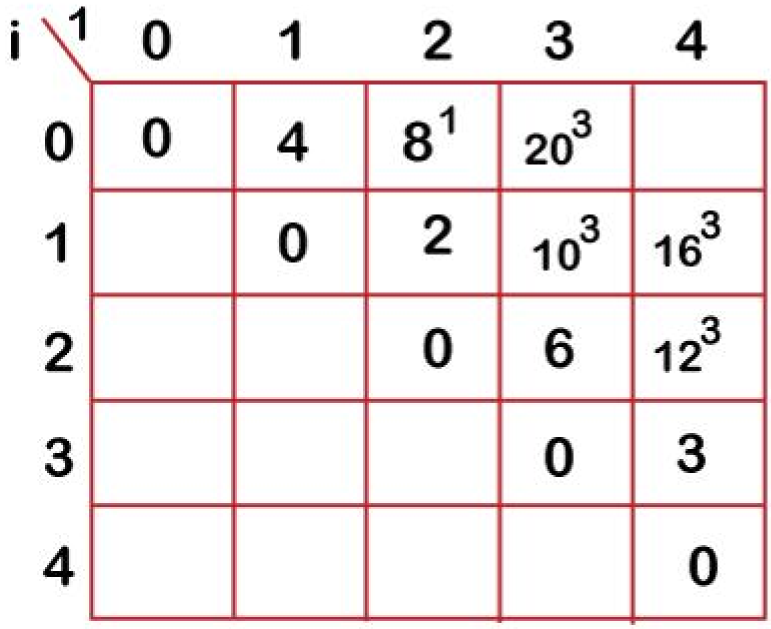
When i=1 and j=4 then we will consider the keys 20, 30, 40

c[1,4] = min{ c[1,1] + c[2,4], c[1,2] + c[3,4], c[1,3] + c[4,4] } + 11

= min{0+12, 2+3, 10+0}+ 11

= min{12, 5, 10} + 11

The minimum value is 5; therefore, c[1,4] = 5+11 = 16



i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

Now we will calculate the values when j-i = 4

When j=4 and i=0 then j-i = 4

In this case, we will consider four keys, i.e., 10, 20, 30 and 40. The frequencies of 10, 20, 30 and 40 are 4, 2, 6 and 3 respectively.

w[0, 4] = 4 + 2 + 6 + 3 = 15

If we consider 10 as the root node then

C[0, 4] = min {c[0,0] + c[1,4]}+ w[0,4]

= min {0 + 16} + 15= 31

If we consider 20 as the root node then

C[0,4] = min{c[0,1] + c[2,4]} + w[0,4]

= min{4 + 12} + 15

= 16 + 15 = 31

If we consider 30 as the root node then,

C[0,4] = min{c[0,2] + c[3,4]} +w[0,4]

= min {8 + 3} + 15

= 26

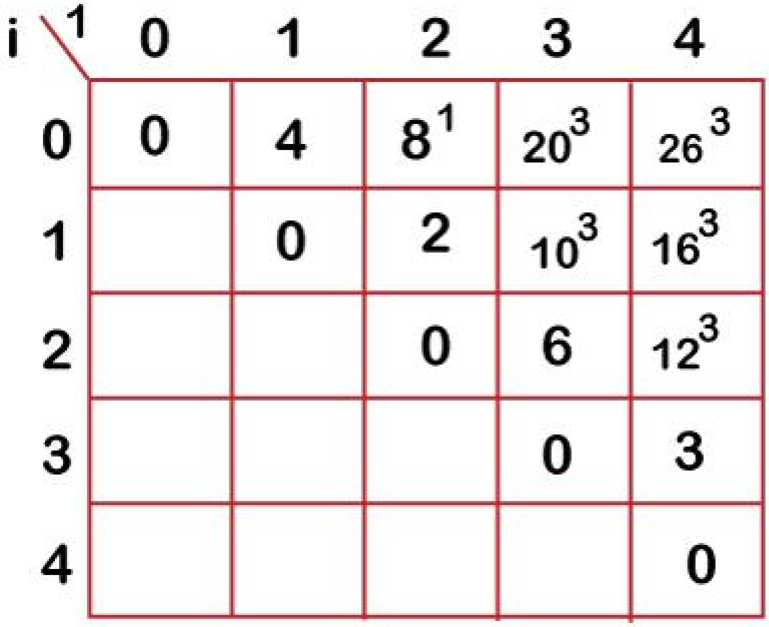
If we consider 40 as the root node then,

C[0,4] = min{c[0,3] + c[4,4]} + w[0,4]

= min{20 + 0} + 15

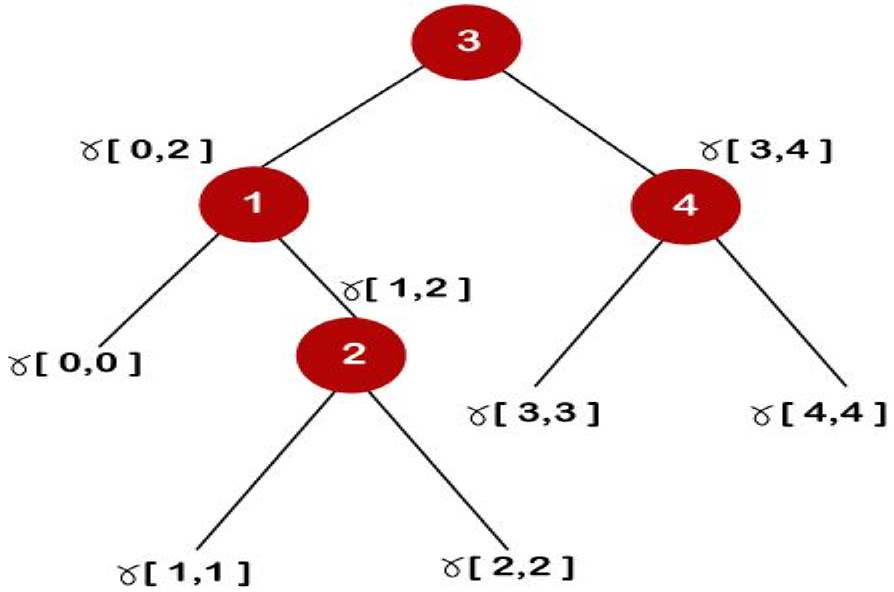
= 35

In the above cases, we have observed that 26 is the minimum cost; therefore, c[0,4] is equal to 26.



i \ 1	0	1	2	3	4
0	0	4	8^1	20^3	26^3
1		0	2	10^3	16^3
2			0	6	12^3
3				0	3
4					0

The optimal binary tree can be created as:



0-1 Knapsack:

in fractional knapsack, the items are always stored fully without using the fractional part of them. Its either the item is added to the knapsack or not. That is why, this method is known as the 0-1 Knapsack problem.

Hence, in case of 0-1 Knapsack, the value of xi can be either 0 or 1, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution in this method. In many instances, Greedy approach may give an optimal solution.

0-1 Knapsack Algorithm

Problem Statement − A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items and weight of ith item is wi and the profit of selecting this item is pi. What items should the thief take?

Let i be the highest-numbered item in an optimal solution S for W dollars. Then S’ = S − {i} is an optimal solution for W – wi dollars and the value to the solution S is Vi plus the value of the sub-problem.

We can express this fact in the following formula: define c[i, w] to be the solution for items 1,2, … , i and the maximum weight w.

The algorithm takes the following inputs

The maximum weight W

The number of items n

The two sequences v = <v1, v2, …, vn> and w = <w1, w2, …, wn>

The set of items to take can be deduced from the table, starting at c[n, w] and tracing backwards where the optimal values came from.

If c[i, w] = c[i-1, w], then item i is not part of the solution, and we continue tracing with c[i-1, w]. Otherwise, item i is part of the solution, and we continue tracing with c [i-1, w-W].

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
   c[0, w] = 0
for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if wi ≤ w then
         if vi + c[i-1, w-wi] then
            c[i, w] = vi + c[i-1, w-wi]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
```

The following examples will establish our statement.

Example

Let us consider that the capacity of the knapsack is W = 8 and the items are as shown in the following table.

Item	**A**	**B**	**C**	**D**
Profit	2	4	7	10
Weight	1	3	5	7

Solution

Using the greedy approach of 0-1 knapsack, the weight that's stored in the knapsack would be A+B = 4 with the maximum profit 2 + 4 = 6. But, that solution would not be the optimal solution.

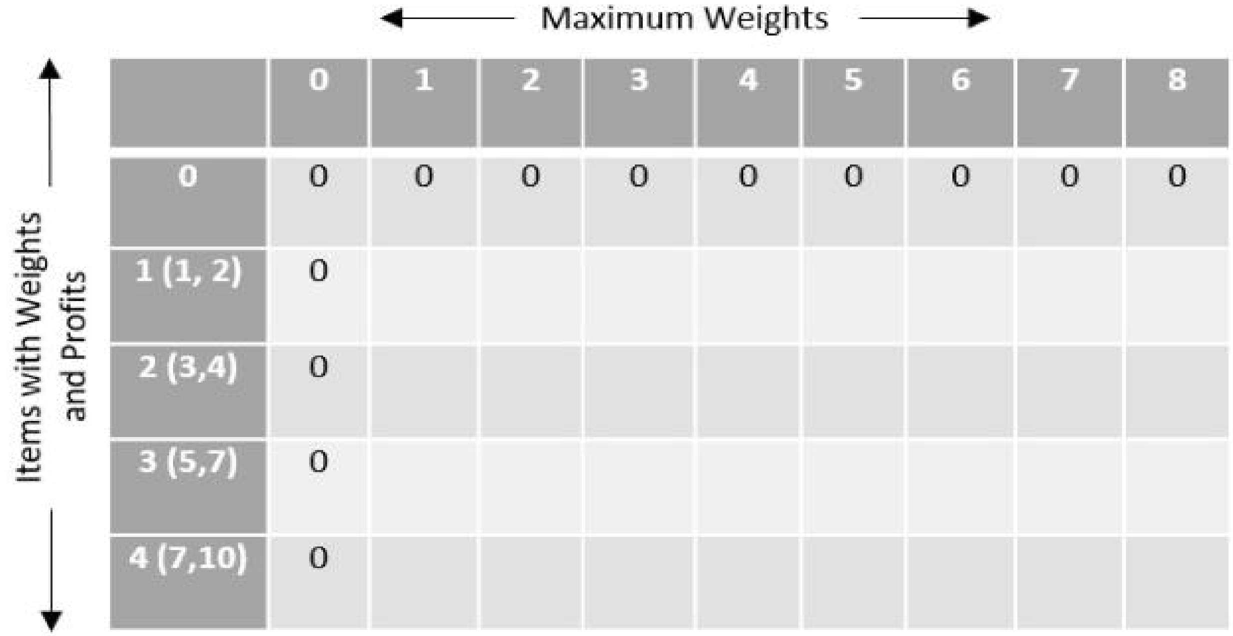
Therefore, dynamic programming must be adopted to solve 0-1 knapsack problems.

Step 1

Construct an adjacency table with maximum weight of knapsack as rows and items with respective weights and profits as columns.

Values to be stored in the table are cumulative profits of the items whose weights do not exceed the maximum weight of the knapsack (designated values of each row)

So we add zeroes to the 0th row and 0th column because if the weight of item is 0, then it weighs nothing; if the maximum weight of knapsack is 0, then no item can be added into the knapsack.


Maximum Weights
Items with Weights and Profits

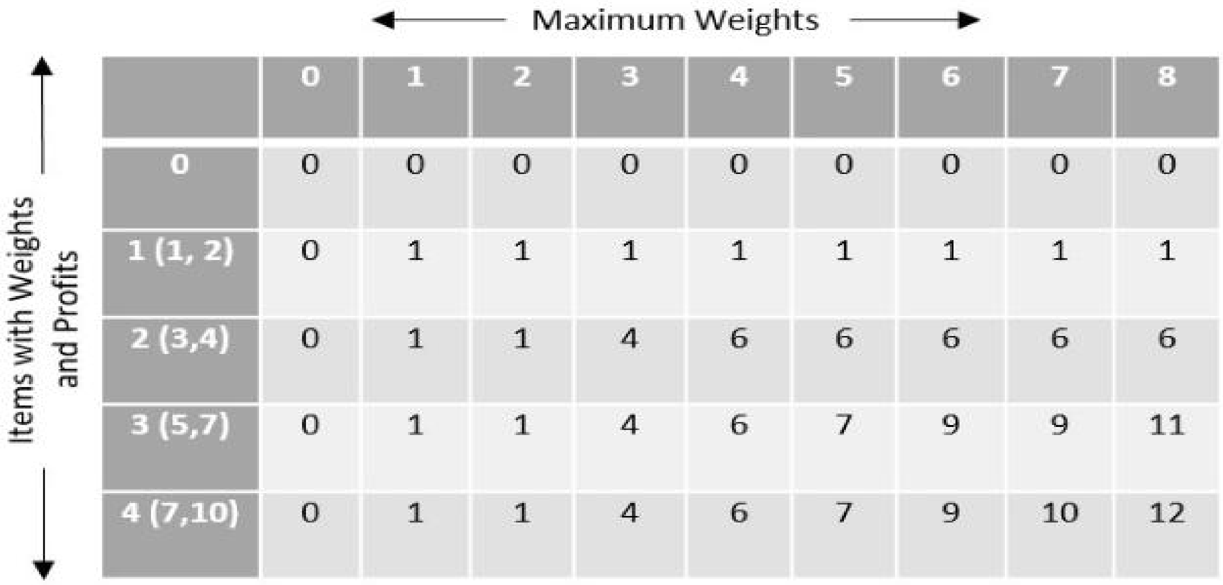
	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0								
2 (3,4)	0								
3 (5,7)	0								
4 (7,10)	0								

The remaining values are filled with the maximum profit achievable with respect to the items and weight per column that can be stored in the knapsack.

The formula to store the profit values is −

c[i,w]=max{c[i−1,w−w[i]]+P[i]}

By computing all the values using the formula, the table obtained would be –

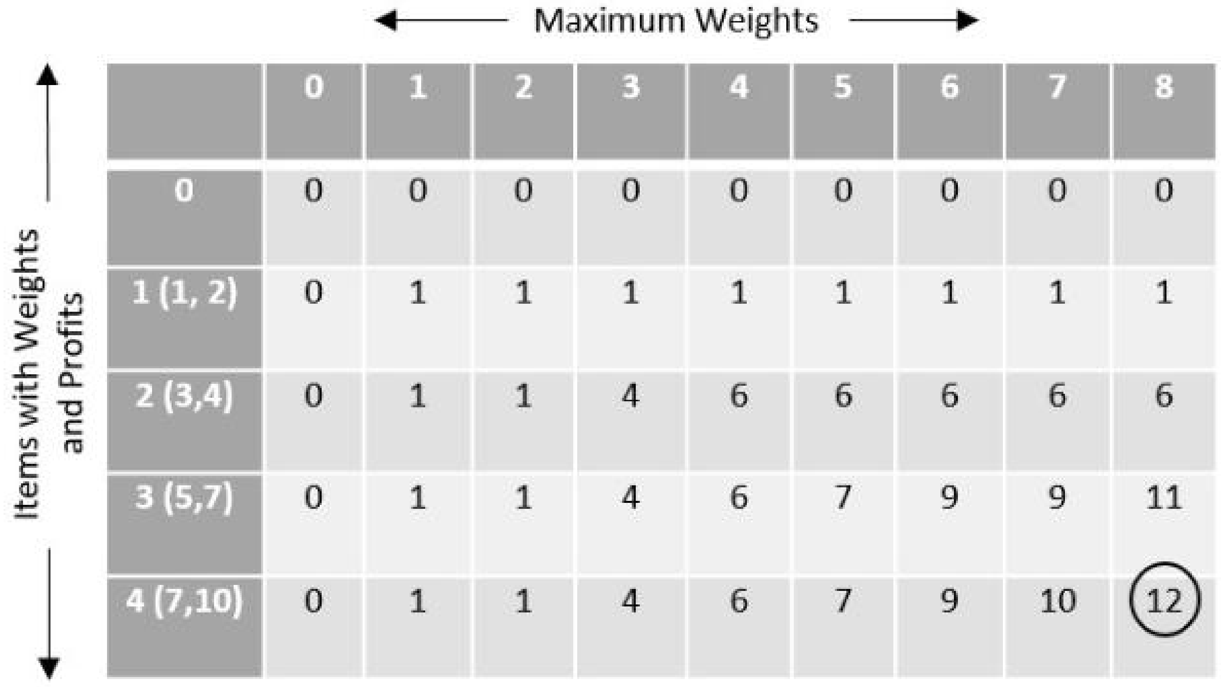


Maximum Weights

Items with Weights and Profits

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

To find the items to be added in the knapsack, recognize the maximum profit from the table and identify the items that make up the profit, in this example, its {1, 7}.



Maximum Weights

Items with Weights and Profits

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

The optimal solution is {1, 7} with the maximum profit is 12.

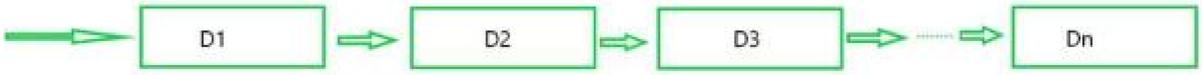
Analysis

This algorithm takes $\Theta(n.w)$ times as table c has $(n+1).(w+1)$ entries, where each entry requires $\Theta(1)$ time to compute.

Reliability Design:

The reliability design problem is the designing of a system composed of several devices connected in series or parallel. Reliability means the probability to get the success of the device.

Let's say, we have to set up a system consisting of D1, D2, D3, …………, and Dn devices, each device has some costs C1, C2, C3, …….., Cn. Each device has a reliability of 0.9 then the entire system has reliability which is equal to the product of the reliabilities of all devices i.e., $\pi ri = (0.9)4$.



It means that 35% of the system has a chance to fail, due to the failure of any one device. the problem is that we want to construct a system whose reliability is maximum. How it can be done so? we can think that we can take more than one copy of each device so that if one device fails we can use the copy of that device, which means we can connect the devices parallel.

When the same type of 3 devices is connected parallelly in stage 1 having a reliability 0.9 each then:

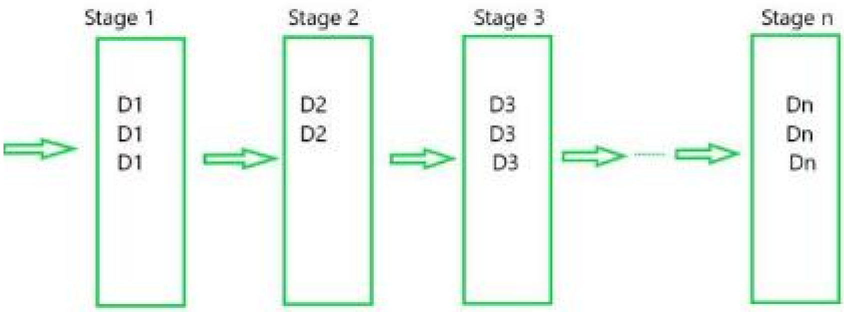
Reliability of device1, $r1= 0.9$

The probability that device does not work well $= 1 - r1 = 1 - 0.9 = 0.1$

The probability that all three copies failed $= (1-r1)3 = (0.1)3 = 0.001$

The Probability that all three copies work properly $= 1 - (1- r1)3 = 1- 0.001 = 0.999$

We can see that the system with multiple copies of the same device parallel may increase the reliability of the system.



Given a cost C and we have to set up a system by buying the devices and we need to find number of copies of each device under the cost such that reliability of a system is maximized.

We have to design a three-stage system with device types D1, D2, and D3. The costs are \$30, \$15, and \$20 respectively. The cost of the system is to be no more than \$105. The reliability of each device type is 0.9, 0.8, and 0.5 respectively.

P_i	C_i	r_i
P_1	30	0.9
P_2	15	0.8
P_3	20	0.5

Explanation:

Given that we have total cost C = 105,

sum of all Ci = 30 + 15 + 20 = 65, the remaining amount we can use to buy a copy of each device in such a way that the reliability of the system, may increase.

Remaining amount = C – sum of Ci = 105 – 65 = 40

Now, let us calculate how many copies of each device we can buy with \$40, If consume all \$40 in device1 then we can buy 40/30 = 1 and 1 copy we have already so overall 2 copies of device1. Now In general, we can have the formula to calculate the upper bond of each device:

Ui = floor(C – ∑Ci / Ci) + 1 (1 is added because we have one copy of each device before)

C1=30, C2=15, C3=20, C=105

r1=0.9, r2=0.8, r3=0.5

u1 = floor ((105+30-(30+15+20))/30) = 2

u2 = 3

u3 = 3

A tuple is just an ordered pair containing reliability and total cost up to a choice of mi's that has been made. we can make pair in of Reliability and Cost of each stage like copySdevice

S0 = {(1,0)}

Device 1:

Each Si from Si-1 by trying out all possible values for ri and combining the resulting tuples together.

let us consider P1 :

1S1 = {(0.9, 30)} where 0.9 is the reliability of stage1 with a copy of one device and 30 is the cost of P1.

Now, two copies of device1 so, we can take one more copy as:

2S1 = { (0.99, 60) } where 0.99 is the reliability of stage one with two copies of device, we can see that it will come as: 1 – (1 – r1)2 = 1 – (1 – 0.9)2 = 1 – 0.01 = 0.99 .

After combining both conditions of Stage1 i.e., with copy one and copy of 2 devices respectively.

S1 = { (0.9, 30), (0.99, 60) }

Device 2:

S2 will contain all reliability and cost pairs that we will get by taking all possible values for the stage2 in conjunction with possibilities calculated in S1.

First of all we will check the reliability at stage2 when we have 1, 2, and 3 as a copy of device. let us assume that Ei is the reliability of the particular stage with n number of devices, so for S2 we first calculate:

E2 (with copy 1) = 1 – (1 – r2)1 = 1 – (1 – 0.8) = 0.8

E2 (with copy 2) = 1 – (1 – r2)2 = 1 – (1 – 0.8)2 = 0.96

E2 (with copy 3) = 1 – (1 – r2)3 = 1 – (1 – 0.8)3 = 0.992

If we use 1 copy of P1 and 1 copy of P2 reliability will be 0.9*0.8 and the cost will be 30+15

One Copy of Device two , 1S2 = { (0.8, 15) } Conjunction with S1 (0.9, 30) = { (0.72,45) }

Similarly, we can calculate other pairs as S2 = (0.72, 45), (0.792, 75), (0.864, 60), (0.98,90) }

We get ordered pair (0.98,90) in S2 when we take 2 copies of Device1and 2 copies of Device2 However, with the remaining cost of 15 (105 – 90), we cannot use device Device3 (we need a minimum of 1 copy of every device in any stage), therefore (0.792, 75) should be discarded and other ordered pairs like it. We get S2 = { (0.72, 45), (0.864, 60), (0.98,90) }. There are other possible ordered pairs too, but all of them exceed cost limitations.

Up to this point we got ordered pairs:

S1 = { (0.9, 30), (0.99, 60) }

S2 = { (0.72, 45), (0.864, 60), (0.98,90)}

Device 3:

First of all we will check the reliability at stage3 when we have 1, 2, and 3 as a copy of device. Ei is the reliability of the particular stage with n number of devices, so for S3 we first calculate:

E3 (with copy 1) = 1 – (1 – r3)1 = 1 – (1 – 0.5) = 0.5

E3 (with copy 2) = 1 – (1 – r3)2 = 1 – (1 – 0.5)2 = 0.75

E3 (with copy 3) = 1 – (1 – r3)3 = 1 – (1 – 0.5)3 = 0.875

Now, possible ordered pairs of device three are as- S3 = { (0.36, 65), (0.396, 95), (0.432, 80), (0.54, 85), (0.648, 100), (0.63, 105) }

(0.648,100) is the solution pair, 0.648 is the maximum reliability we can get under the cost constraint of 105.

Travelling Salesman Problem:

Travelling salesman problem is the most notorious computational problem. We can use brute-force approach to evaluate every possible tour and select the best one. For n number of vertices in a graph, there are (n−1)! number of possibilities. Thus, maintaining a higher complexity.

However, instead of using brute-force, using the dynamic programming approach will obtain the solution in lesser time, though there is no polynomial time algorithm.

Travelling Salesman Dynamic Programming Algorithm

Let us consider a graph G = (V,E), where V is a set of cities and E is a set of weighted edges. An edge e(u, v) represents that vertices u and v are connected. Distance between vertex u and v is d(u, v), which should be non-negative.

Suppose we have started at city 1 and after visiting some cities now we are in city j. Hence, this is a partial tour. We certainly need to know j, since this will determine which cities are most convenient to visit next. We also need to know all the cities visited so far, so that we don't repeat any of them. Hence, this is an appropriate sub-problem.

For a subset of cities S ϵ

{1,2,3,...,n} that includes 1, and j ϵ

S, let C(S, j) be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j.

When |S|> 1 , we define C(S,1)= ∝

since the path cannot start and end at 1.

Now, let express C(S, j) in terms of smaller sub-problems. We need to start at 1 and end at j. We should select the next city in such a way that

$C(S,j)=\min C(S-\{j\},i)+d(i,j) where i \epsilon S and i \neq j$

Algorithm: Traveling-Salesman-Problem

```
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S ϵ {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
```

for all j є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i є S and i ≠ j}

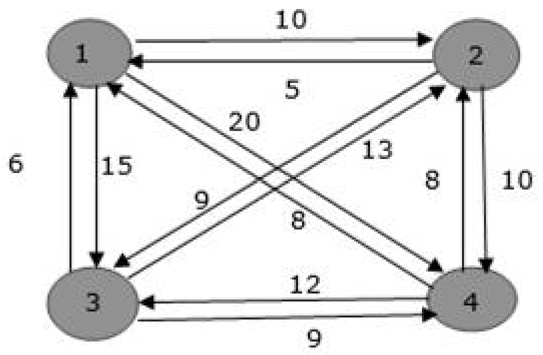
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

Analysis

There are at the most 2n.n sub-problems and each one takes linear time to solve. Therefore, the total running time is O(2n.n2).

Example

In the following example, we will illustrate the steps to solve the travelling salesman problem.



From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

S = Φ

Cost(2,Φ,1)=d(2,1)=5

Cost(3,Φ,1)=d(3,1)=6

Cost(4,Φ,1)=d(4,1)=8

S = 1

Cost(i,s)=min{Cos(j,s−(j))+d[i,j]}

Cost(2,{3},1)=d[2,3]+Cost(3,Φ,1)=9+6=15

Cost(2,{4},1)=d[2,4]+Cost(4,Φ,1)=10+8=18

Cost(3,{2},1)=d[3,2]+Cost(2,Φ,1)=13+5=18

Cost(3,{4},1)=d[3,4]+Cost(4,Φ,1)=12+8=20

Cost(4,{3},1)=d[4,3]+Cost(3,Φ,1)=9+6=15

Cost(4,{2},1)=d[4,2]+Cost(2,Φ,1)=8+5=13

S = 2

Cost(2,{3,4},1)=min{d[2,3]+Cost(3,{4},1)=9+20=29d[2,4]+Cost(4,{3},1)=10+15=25=25

Cost(3,{2,4},1)=min{d[3,2]+Cost(2,{4},1)=13+18=31d[3,4]+Cost(4,{2},1)=12+13=25=25

Cost(4,{2,3},1)=min{d[4,2]+Cost(2,{3},1)=8+15=23d[4,3]+Cost(3,{2},1)=9+18=27=23

S = 3

Cost(1,{2,3,4},1)=min⌈ ⌊ { | | d[1,2]+Cost(2,{3,4},1)=10+25=35d[1,3]+Cost(3,{2,4},1)=15+25=40d[1,4]+Cost(4,{2,3},1)=20+23=43=35

The minimum cost path is 35.

UNIT–IV

Back tracking: N-queens problem, Graph coloring, Hamiltonian cycles
Branch-and-bound: FIFO & LC branch and Bound methods, 0/1 Knapsack problem, Traveling salesperson
Course Outcome:
- Describe the backtracking paradigm and when this approach is used

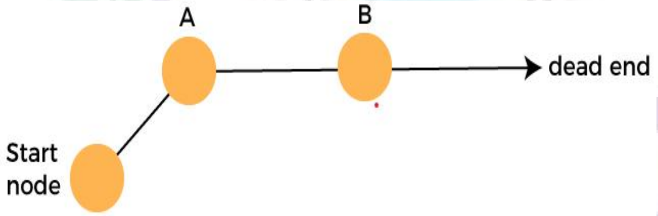
Backtracking:

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

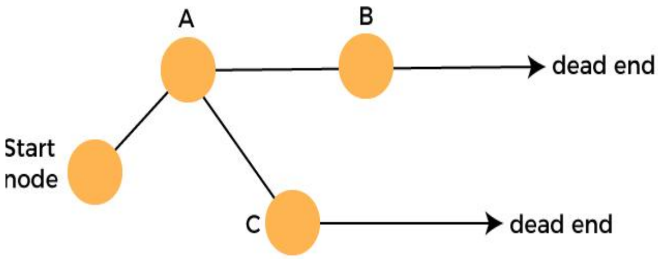
Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria. When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions. Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.
Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

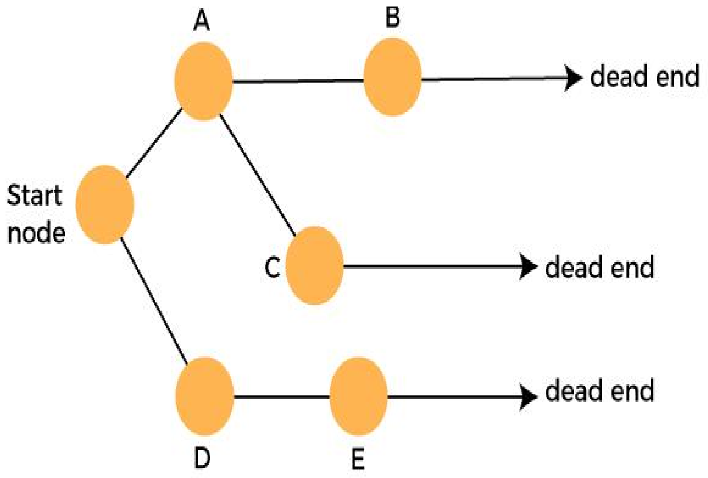
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



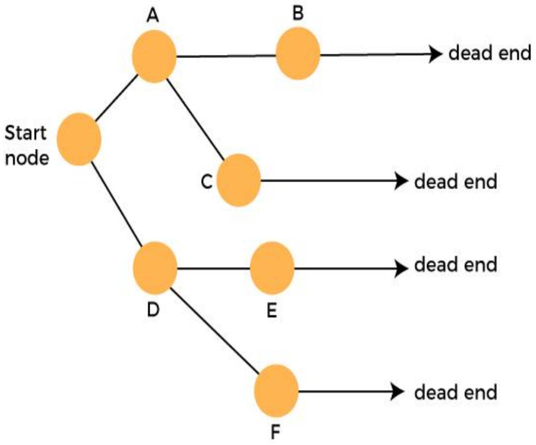
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



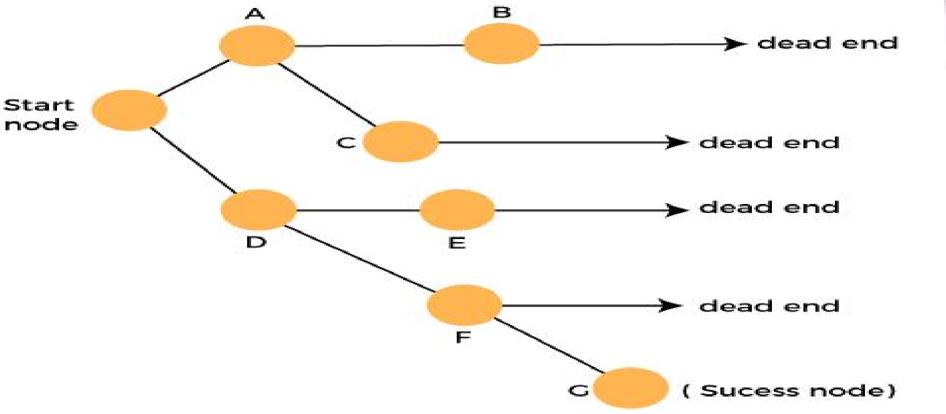
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



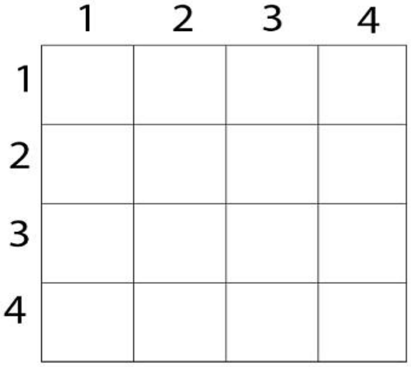
The terms related to the backtracking are:

Live node: The nodes that can be further generated are known as live nodes.
E node: The nodes whose children are being generated and become a success node.
Success node: The node is said to be a success node if it provides a feasible solution.
Dead node: The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.
Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:
Implicit constraint: It is a rule in which how each element in a tuple is related.
Explicit constraint: The rules that restrict each element to be chosen from the given set.
Applications of Backtracking

- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

N-Queens Problem:

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.
It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queen's problem.
Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



4x4 chessboard

Since, we have to place 4 queens such as q1 q2 q3 and q4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."
Now, we place queen q1 in the very first acceptable position (1, 1). Next, we put queen q2 so that both these queens do not attack each other. We find that if we place q2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q3' safely. So we backtrack one step and place the queen 'q2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q4' can be placed safely. Then we have to backtrack till 'q1' and place it to (1, 2) and then all other queens are placed safely by moving q2 to (2, 4), q3 to (3, 1) and q4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

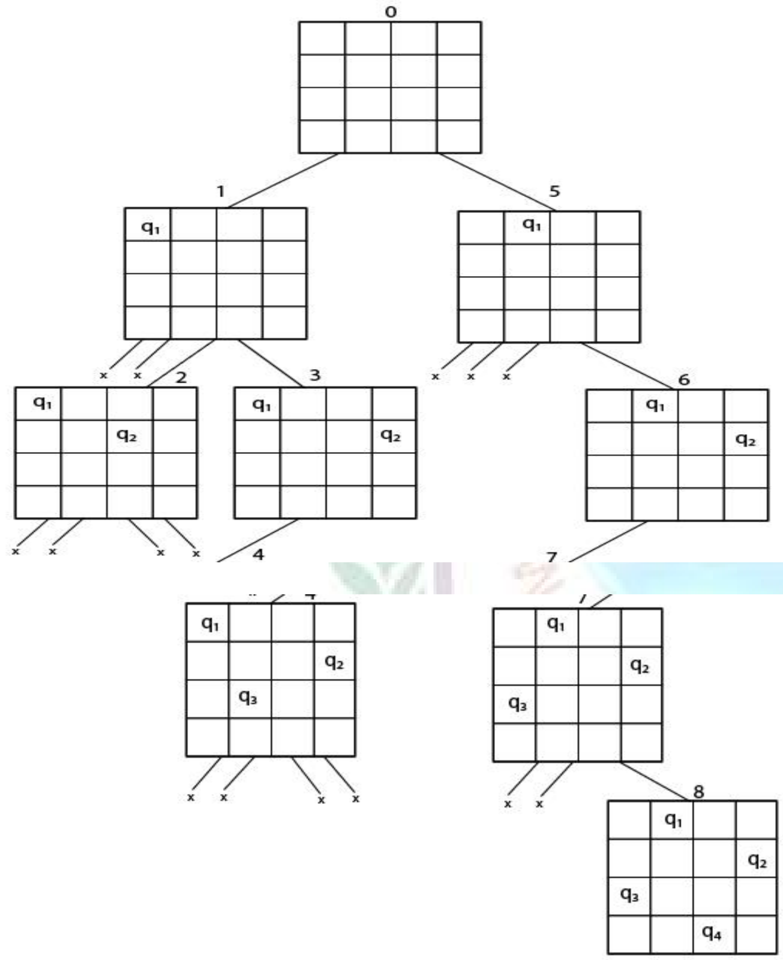
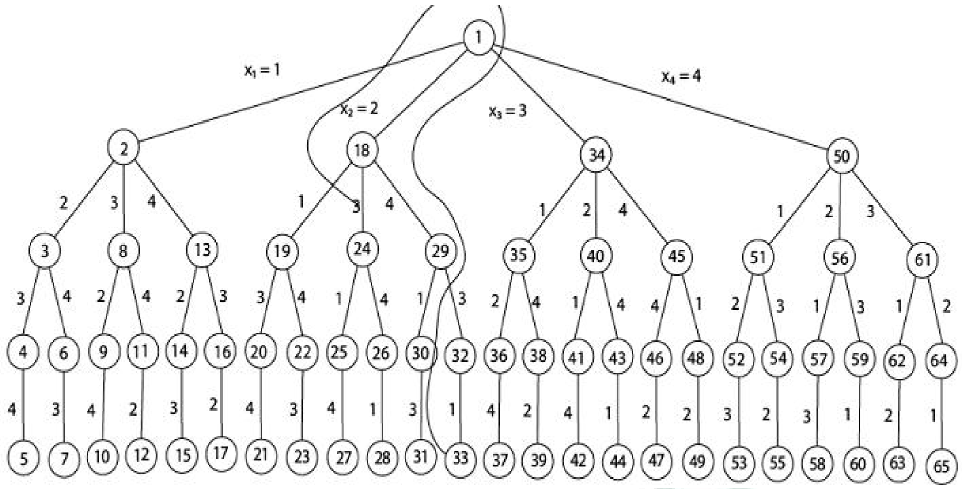


Fig shows the complete state space for 4 - queen's problem.

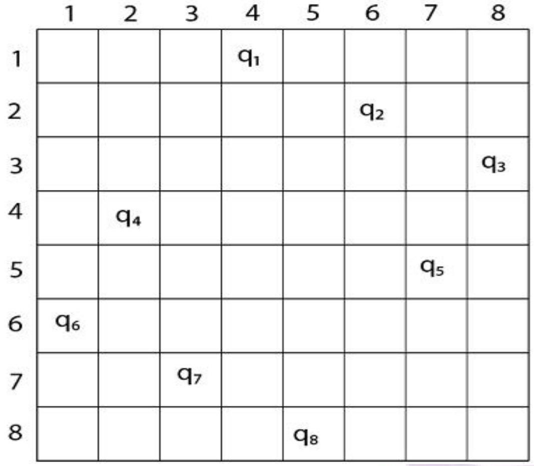
But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x1, x2, x3, x4) where xi represents the column on which queen "qi" is placed.

One possible solution for 8 queens problem is shown in fig:



	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

Graph Coloring Problem:

Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the vertex coloring problem.

If coloring is done using at most k colors, it is called k-coloring.

The smallest number of colors required for coloring graph is called its chromatic number.

The chromatic number is denoted by X(G). Finding the chromatic number for the graph is NP-complete problem.

Graph coloring problem is both, decision problem as well as an optimization problem. A decision problem is stated as, “With given M colors and graph G, whether such color scheme is possible or not?”.

The optimization problem is stated as, “Given M colors and graph G, find the minimum number of colors required for graph coloring.”

Graph coloring problem is a very interesting problem of graph theory and it has many diverse applications. Few of them are listed below.

Applications of Graph Coloring Problem

Design a timetable.

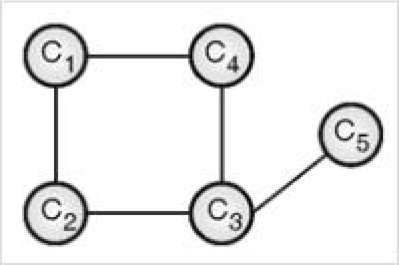
Sudoku

Register allocation in the compiler

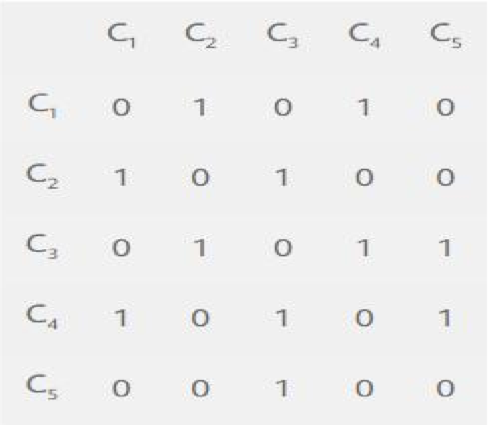
Map coloring

Mobile radio frequency assignment:

The input to the graph is an adjacency matrix representation of the graph. Value M(i, j) = 1 in the matrix represents there exists an edge between vertex i and j. A graph and its adjacency matrix representation are shown in Figure (a)



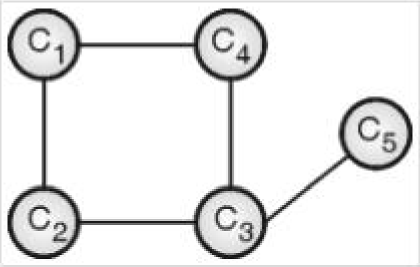
(a). Graph G



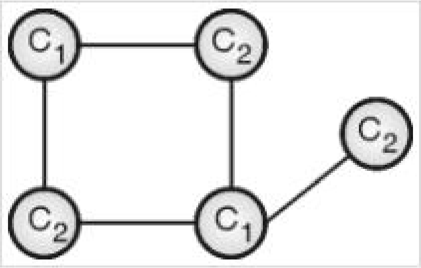
	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

Adjacency matrix for graph G

The problem can be solved simply by assigning a unique color to each vertex, but this solution is not optimal. It may be possible to color the graph with colors less than |V|. Figure (b) and figure (c) demonstrate both such instances. Let Ci denote the ith color.



(b). Nonoptimal solution (uses 5 colors)



(c). Optimal solution (uses 2 colors)

This problem can be solved using backtracking algorithms as follows:

List down all the vertices and colors in two lists

Assign color 1 to vertex 1

If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.

Repeat the process until all vertices are colored.

Algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects next color i + 1 and test is repeated. Consider the graph shown in
Figure (d)

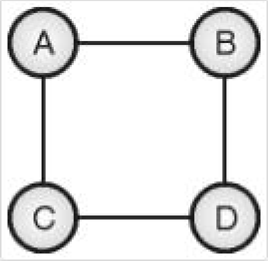


Figure (d)

If we assign color 1 to vertex A, the same color cannot be assigned to vertex B or C. In the next step, B is assigned some different colors 2. Vertex A is already colored and vertex D is a neighbor of B, so D cannot be assigned color 2. The process goes on. State-space tree is shown in Figure (e)

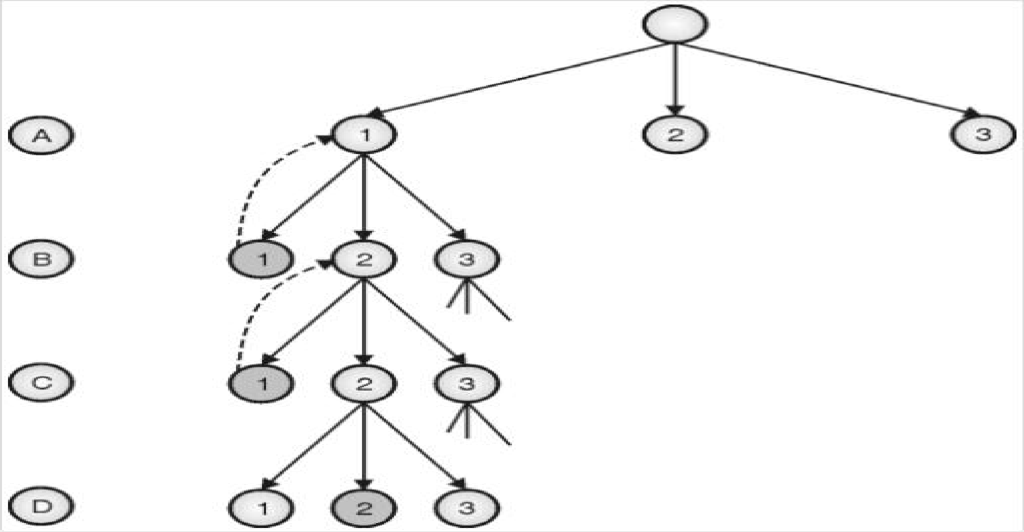
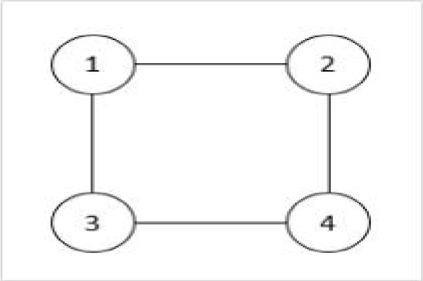


Figure (e): State-space tree of the graph of Figure (d)

Thus, vertices A and C will be colored with color 1, and vertices B and D will be colored with color 2.
Complexity Analysis
The number of anode increases exponentially at every level in state space tree. With M colors and n vertices, total number of nodes in state space tree would be
1 + M + M2 + M3 + …. + Mn

Examples on Graph Coloring Problem

Example: Apply backtrack on the following instance of graph coloring problem of 4 nodes and 3 colors



This problem can be solved using backtracking algorithms. The formal idea is to list down all the vertices and colors in two lists. Assign color 1 to vertex 1. If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2. The process is repeated until all vertices are colored. The algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects the next color i + 1 and the test is repeated. This graph can be colored with 3 colors.

The solution tree is shown in Fig. (i):

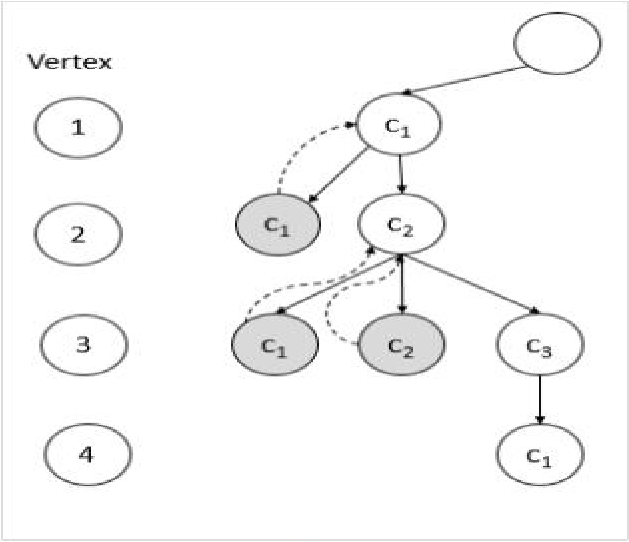


Fig. (i): Solution tree

Vertex	Assigned Color
1	c1
2	c2
3	c3
4	c1

<u>Hamiltonian Cycle using Backtracking:</u>

The Hamiltonian cycle is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges

The Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.

The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – E – D – C – A forms a Hamiltonian cycle. It visits all the vertices exactly once, but does not visit the edges <B, D>.

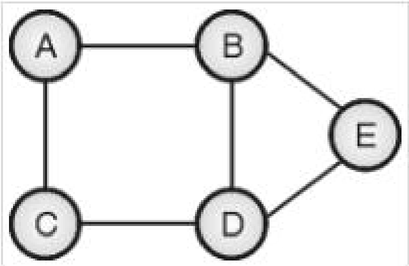


Figure (a)

The Hamiltonian cycle problem is also both, decision problem and an optimization problem. A decision problem is stated as, “Given a path, is it a Hamiltonian cycle of the graph?”.

The optimization problem is stated as, “Given graph G, find the Hamiltonian cycle for the graph.”

We can define the constraint for the Hamiltonian cycle problem as follows:

In any path, vertex i and (i + 1) must be adjacent.

1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).

Vertex i must not appear in the first (i – 1) vertices of any path.

With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

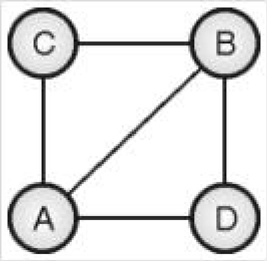
Complexity Analysis

Looking at the state space graph, in worst case, total number of nodes in tree would be,

T(n) = 1 + (n – 1) + (n – 1)2 + (n – 1)3 + … + (n – 1)n – 1

T(n) = O(nn). Thus, the Hamiltonian cycle algorithm runs in exponential time.

Example: Show that given graph has a Hamiltonian cycle



Solution:

We can start with any random vertex. Let us start with vertex A. Neighbors of A are {B, C, D}. The inclusion of B does not lead to a complete solution. So explore it as shown in Figure (c).

Adjacent vertices to B are {C, D}. The inclusion of C does not lead to a complete solution. All adjacent vertices of C are already members of the path formed so far. So it leads to a dead end.

Backtrack and go to B and explore its next neighbor i.e. D.

The inclusion of D does not lead to a complete solution, and all adjacent vertices of D are already a member of the path formed so far. So it leads to a dead end.

Figure: (b) initial tree

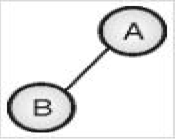


Figure (c): Added node B

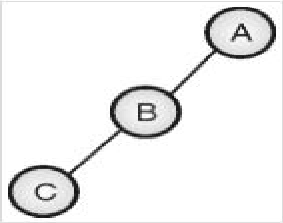


Figure (d): Node C added, dead-end

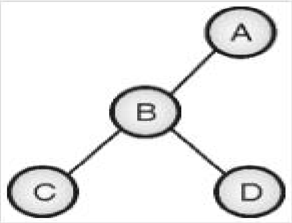
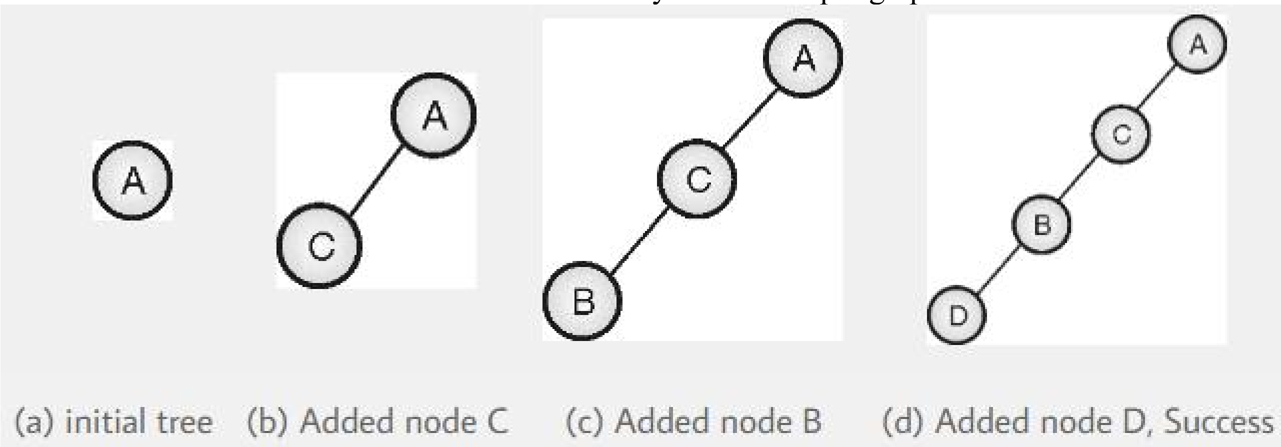
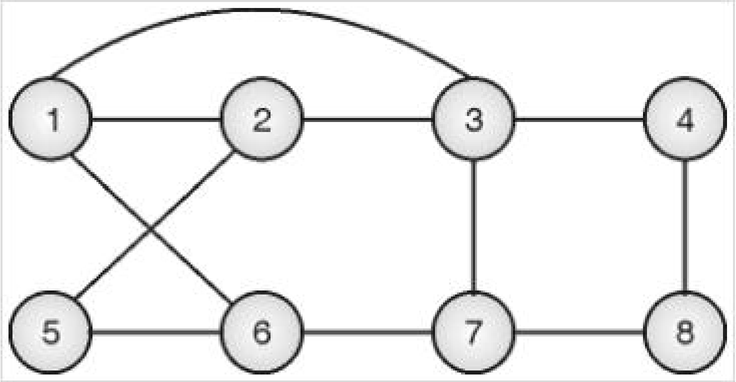


Figure (e): Node D added, dead-end

Backtrack and go to B. Now B does not have any more neighbors, so backtrack and go to A. Explore the next neighbor of A i.e. C. By repeating the same procedure, we get the state space trees as shown below. And path A – C – B – D – A is detected as the Hamiltonian cycle of the input graph



Explain how to find Hamiltonian Cycle by using Backtracking in a given graph



Solution:

The backtracking approach uses a state-space tree to check if there exists a Hamiltonian cycle in the graph. Figure (f) shows the simulation of the Hamiltonian cycle algorithm. For simplicity, we have not explored all possible paths, the concept is self-explanatory.

Step 1: Tour is started from vertex 1. There is no path from 5 to 1. So it's the dead-end state.

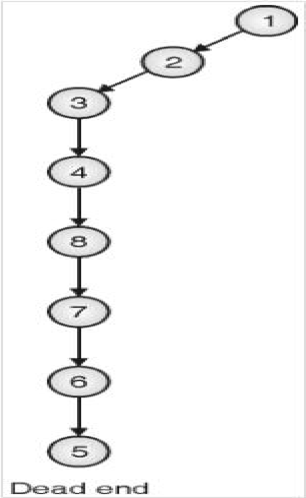
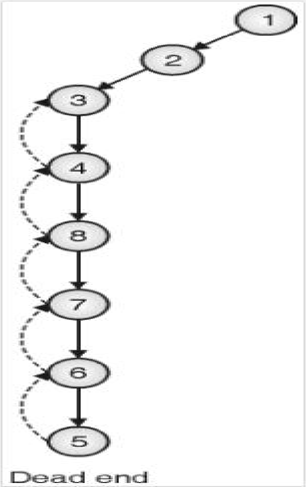
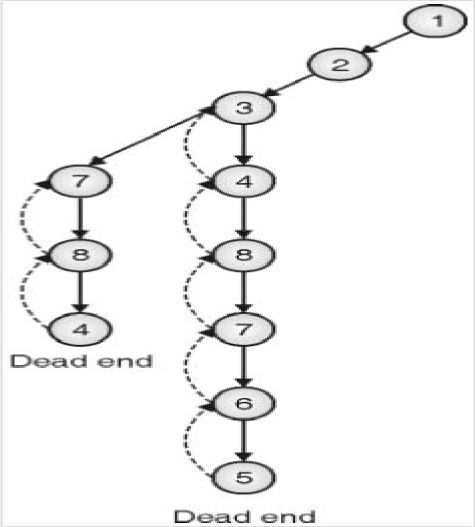


Figure (f)

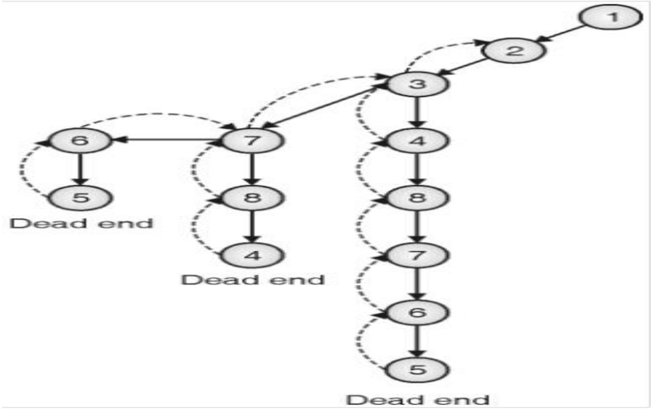
Step 2: Backtrack to the node from where the new path can be explored, that is 3 here



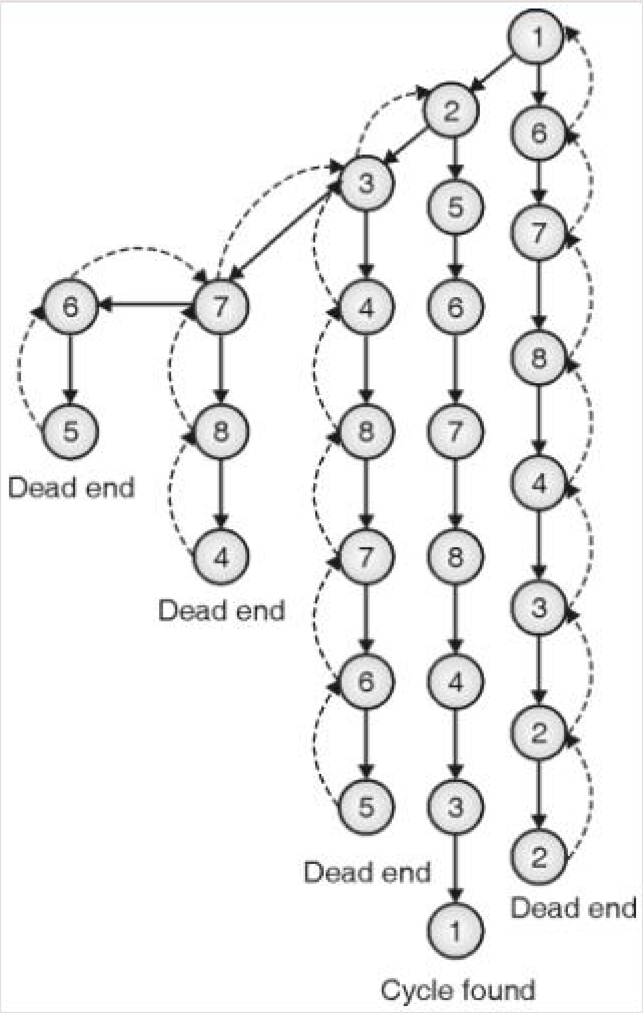
Step 3: New path also leads to a dead end so backtrack and explore all possible paths



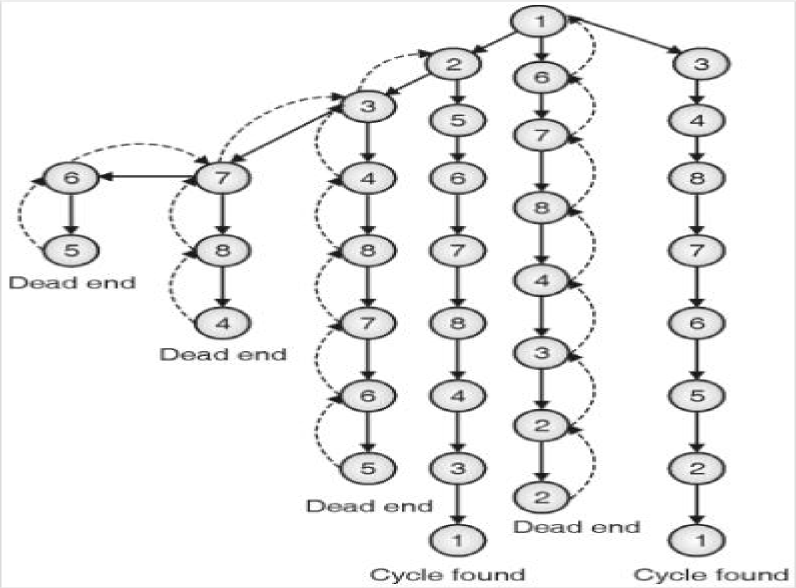
Step 4: Next path is also leading to a dead-end so keep backtracking until we get some node that can generate a new path, i.e .vertex 2 here



Step 5: One path leads to Hamiltonian cycle, next leads to a dead end so backtrack and explore all possible paths at each vertex

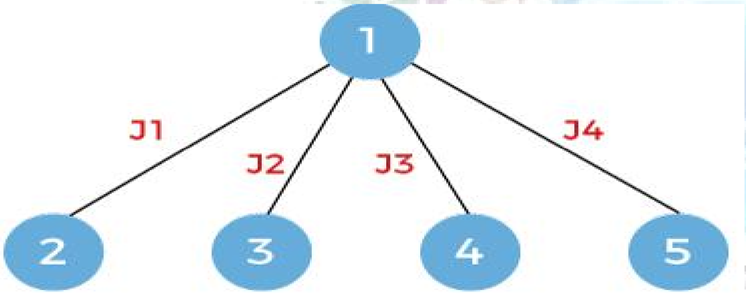


Step 6: Total two Hamiltonian cycles are detected in a given graph

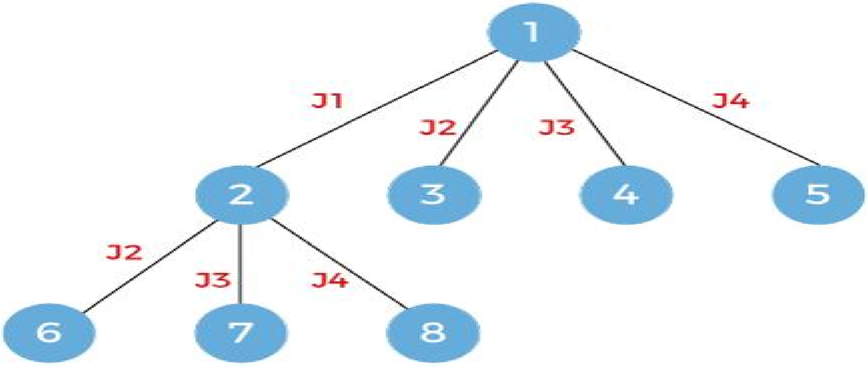


Branch and bound:

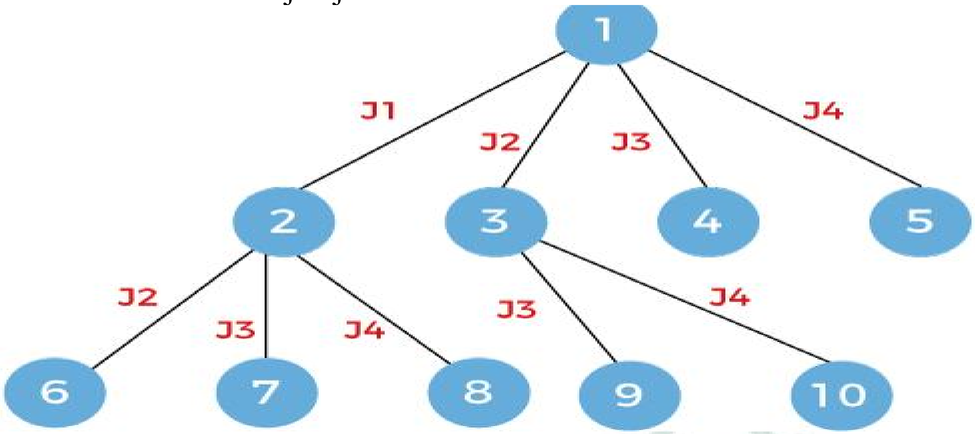
Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.
Jobs = {j1, j2, j3, j4}
P = {10, 5, 8, 3}
d = {1, 2, 1, 2}
The above are jobs, problems and problems given. We can write the solutions in two ways which are given below:
Suppose we want to perform the jobs j1 and j2 then the solution can be represented in two ways:
The first way of representing the solutions is the subsets of jobs.
S1 = {j1, j4}
The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.
S2 = {1, 0, 0, 1}
The solution s1 is the variable-size solution while the solution s2 is the fixed-size solution.
First, we will see the subset method where we will see the variable size.
<u>**First method:**</u>



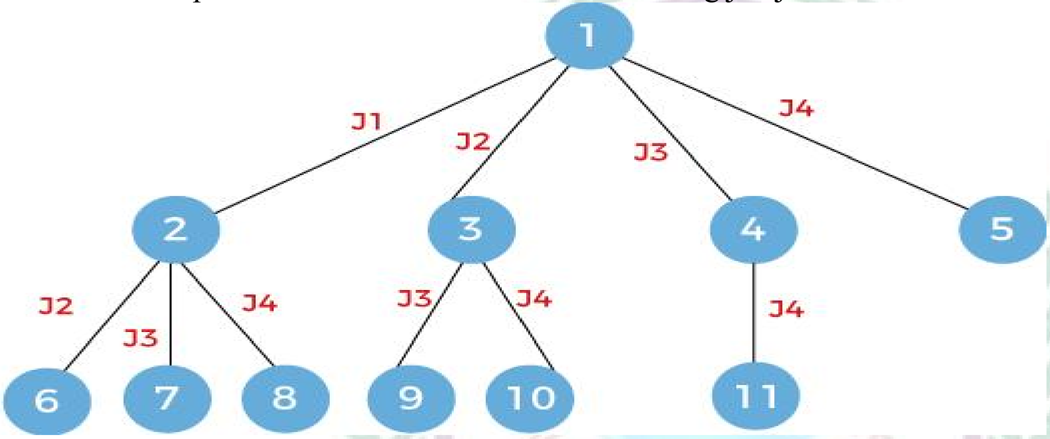
In this case, we first consider the first job, then second job, then third job and finally we consider the last job.
As we can observe in the above figure that the breadth first search is performed but not the depth first search. Here we move breadth wise for exploring the solutions. In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.
Now one level is completed. Once I take first job, then we can consider either j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.



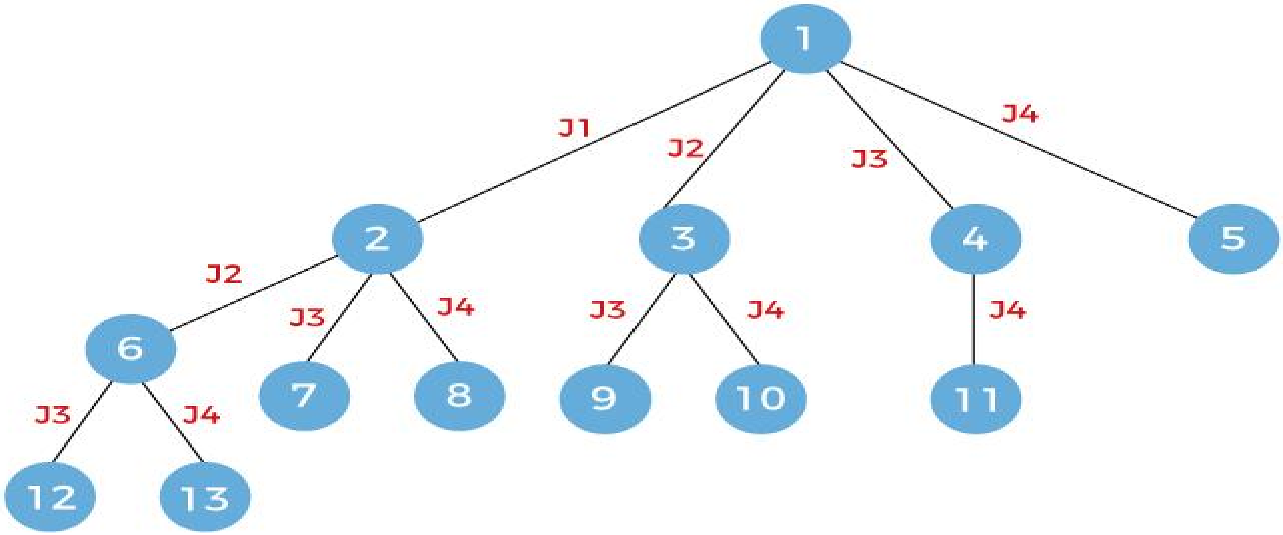
Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.



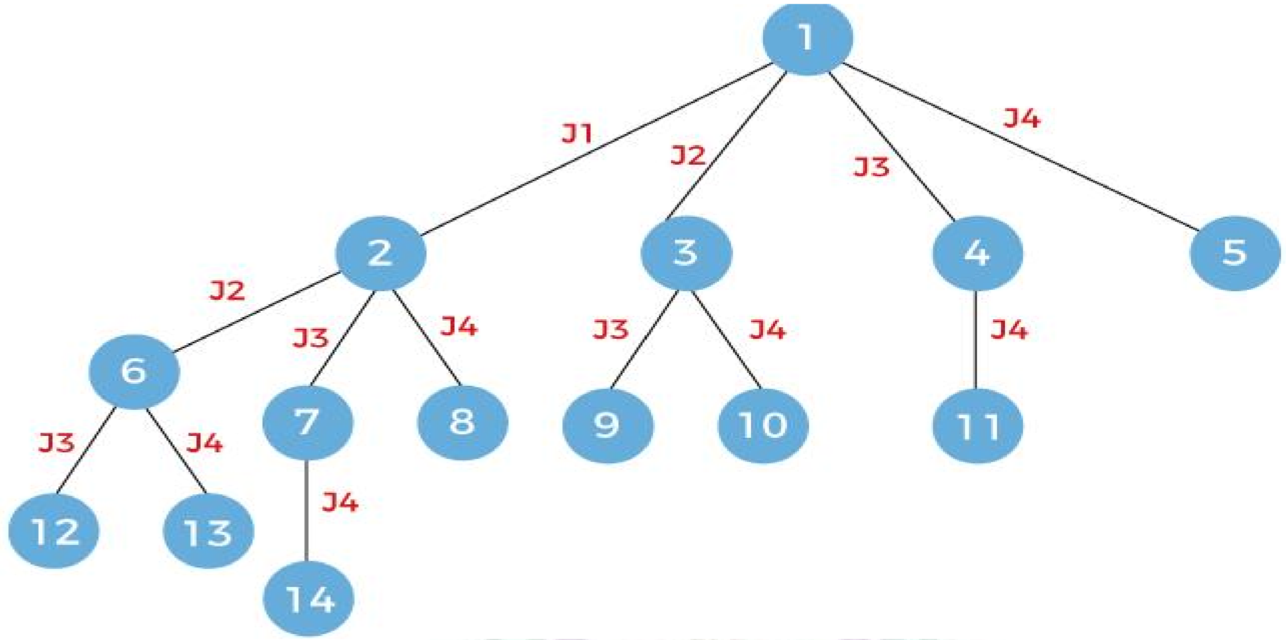
Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.



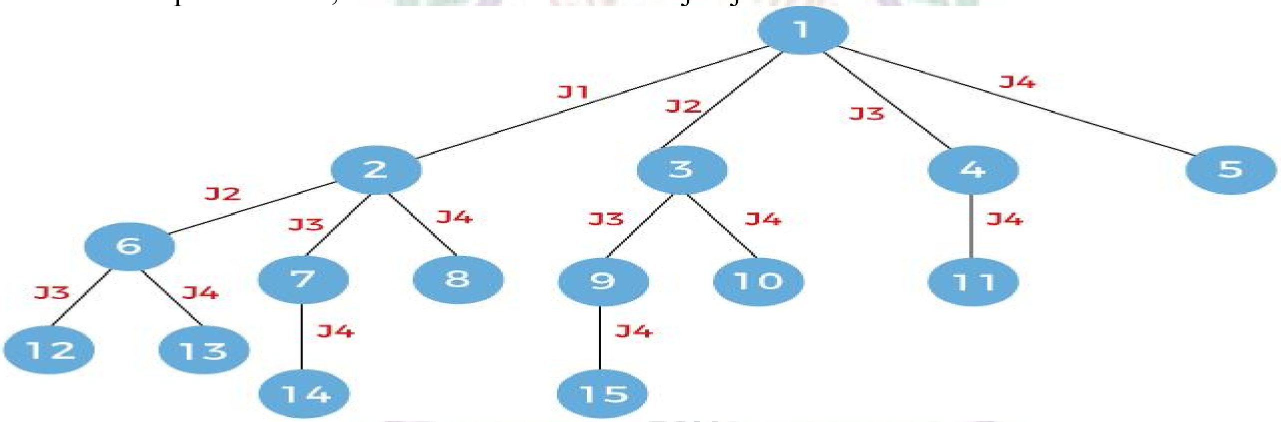
Now we will expand node 6, and here we will consider the jobs j3 and j4.



Now we will expand node 7 and here we will consider job j4.



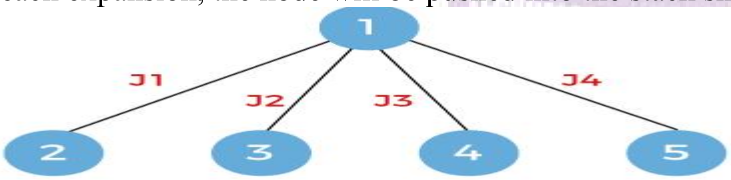
Now we will expand node 9, and here we will consider job j4.

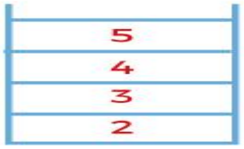


The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.
The above is the state space tree for the solution s1 = {j1, j4}

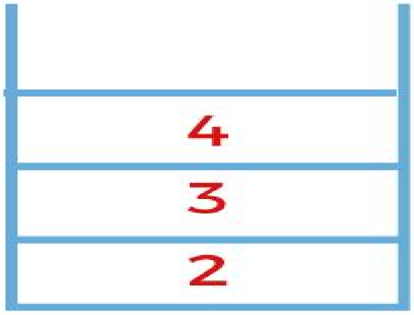
Second method:

We will see another way to solve the problem to achieve the solution s1.
First, we consider the node 1 shown as below:
Now, we will expand the node 1. After expansion, the state space tree would be appeared as:
On each expansion, the node will be pushed into the stack shown as below:

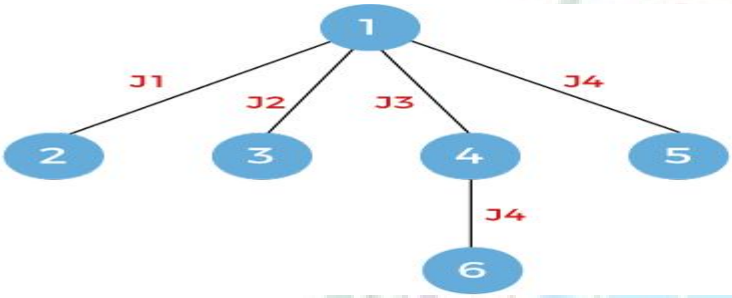


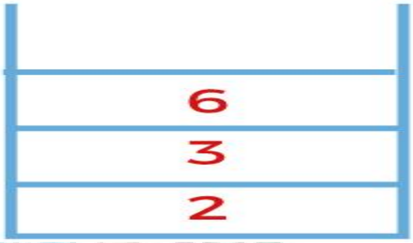


Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.

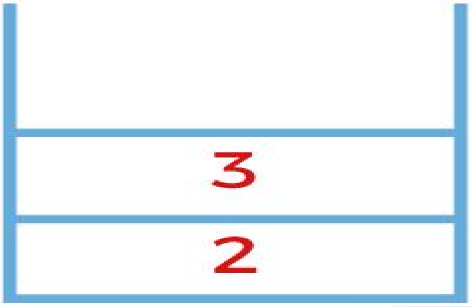


The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:

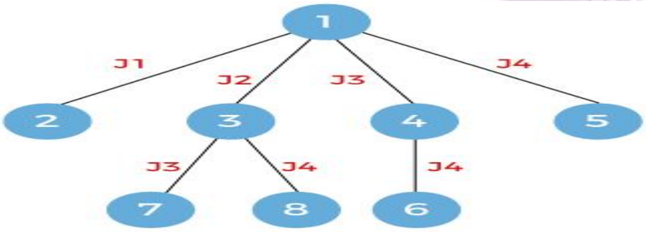


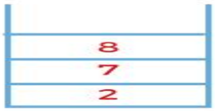


The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j4 so there is no further scope of expansion.

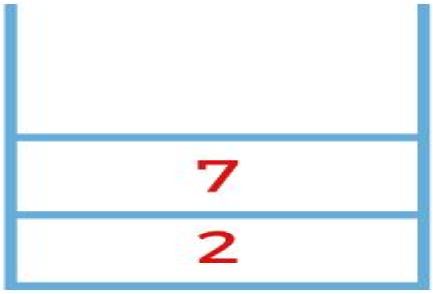


The next node to be expanded is node 3. Since the node 3 works on the job j2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:

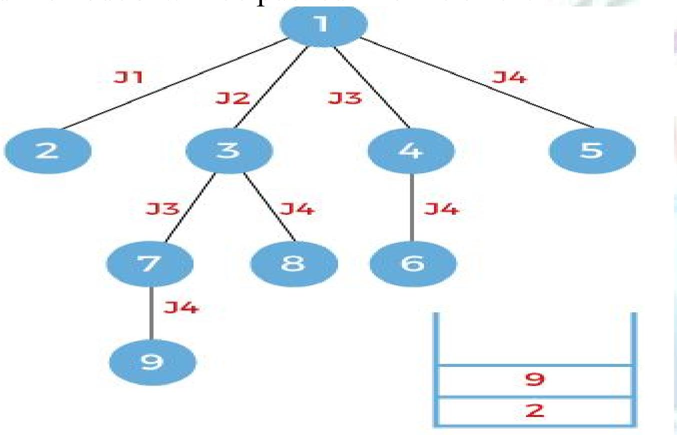




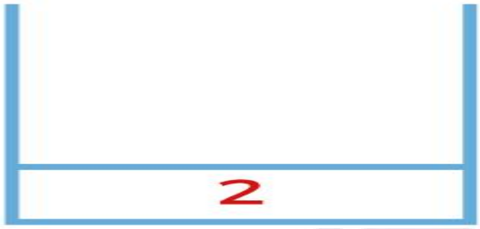
The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j3 so node 7 will be further expanded to node 9 that works on the job j4 as shown as below and the node 9 will be pushed into the stack.

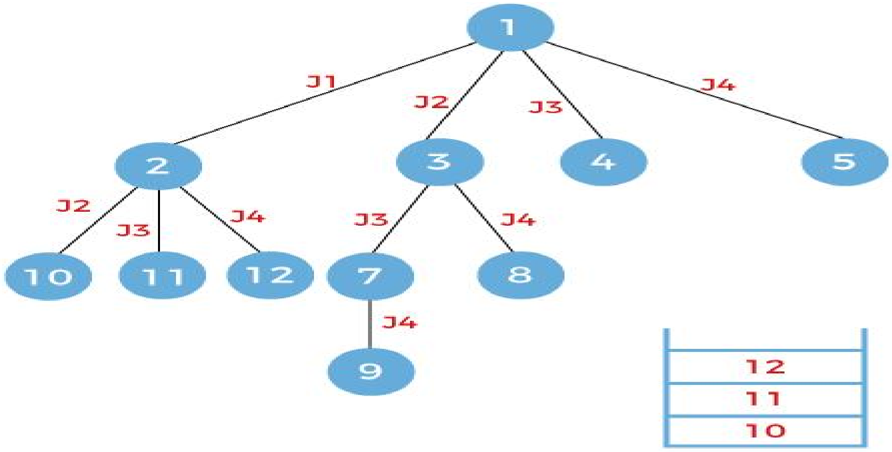


The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j2, j3, and j4 respectively.

There newly nodes will be pushed into the stack shown as below:



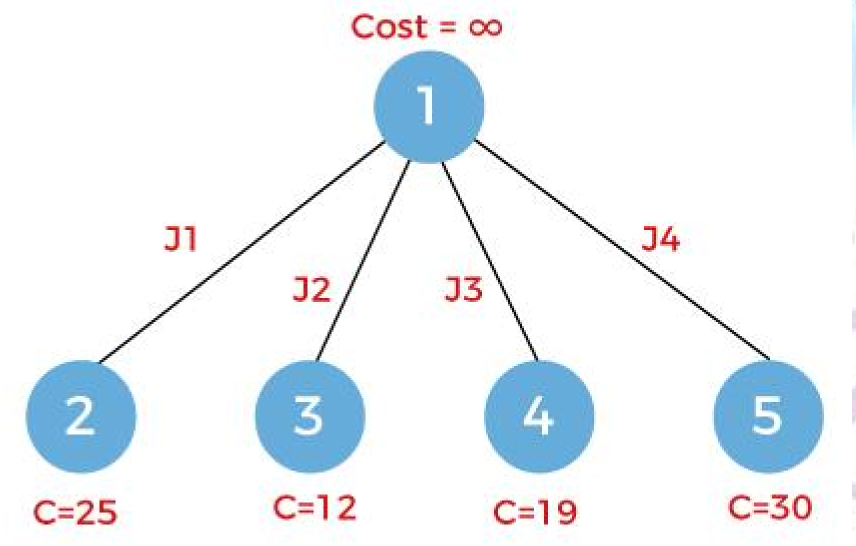
In the above method, we explored all the nodes using the stack that follows the LIFO principle.

<u>Third method:</u>

There is one more method that can be used to find the solution and that method is Least cost branch and bound. In this technique, nodes are explored based on the cost of the node. The cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.

Let's first consider the node 1 having cost infinity shown as below:

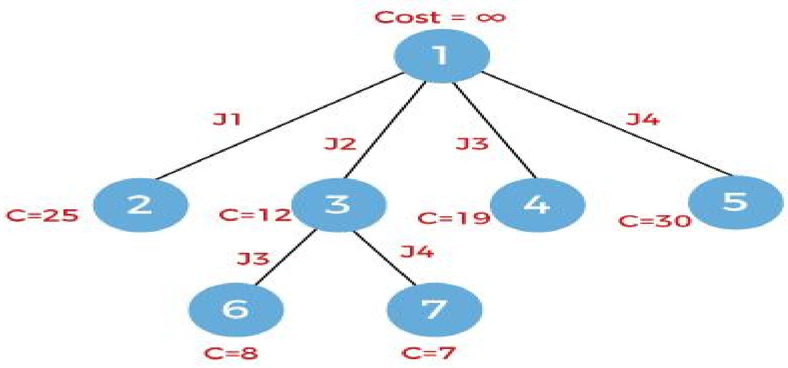
Now we will expand the node 1. The node 1 will be expanded into four nodes named as 2, 3, 4 and 5 shown as below:



Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.

Since it is the least cost branch n bound, so we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 3. So, we will explore the node 3 having cost 12.

Since the node 3 works on the job j2 so it will be expanded into two nodes named as 6 and 7 shown as below:



The node 6 works on job j3 while the node 7 works on job j4. The cost of the node 6 is 8 and the cost of the node 7 is 7. Now we have to select the node which is having the minimum cost. The node 7 has the minimum cost so we will explore the node 7. Since the node 7 already works on the job j4 so there is no further scope for the expansion.

FIFO Branch and Bound solution

FIFO Branch and Bound solution is one of the methods of branch and bound.

Branch and Bound is the state space search method where all the children E-node that is generated before the live node, becomes an E- node.

FIFO branch and bound search is the one that follows the BFS like method. It does so as the list follows first in and first out.

Some key terms to keep in mind while proceeding further:

What is a live node?

A live node is the one that has been generated but its children are not yet generated.

What is an E node?

An E node is the one that is being explored at the moment.

What is a dead node?

A dead node is one that is not being generated or explored any further. The children of the dead node have already been expanded to their full capacity.

To proceed further with the FIFO branch and bound we use a queue.

To begin with, we keep the queue empty. Then we assume a node 1.

In FIFO search the E node is assumed as node 1. After that, we need to generate children of Node 1. The children are the live nodes, and we place them in the queue accordingly. Then we delete node 2 from the queue and generate children of node 2.

Next, we delete another element from the queue and assume that as the E node. We then generate all its children.

In the same process, we delete the next element and generate their children. We keep repeating the process till the queue is covered and we find a node that satisfies the conditions of the problem. When we have found the answer node, the process terminates.

Let us understand it through an example:

We start by taking an empty queue:

In the given diagram we have assumed that node 12 is the answer node.

As mentioned we start by assuming node 1 as the E node and generate all its children.

2	3	4

Then we delete node 1 from the queue, take node 2 as the E node, and generate all its children:

3	4	5	6

We delete the next element and generate children for the next node, assuming it to be the E node.

4	5	6

We repeat the same process. The generated children of 4, that is node 9 are killed by the boundary function.

5	6

Similarly, we proceed further, but the children of nodes 5, meaning the nodes 10, and 11 are also generated and killed by boundary function. The last standing node in the queue is 6, the children of node 6 are 12, and it satisfies all the conditions for the problem, therefore it is the answer node.

Least Cost (LC) Search:

The selection rule for the next E-node in FIFO or LIFO branch and bound is sometimes “blind”. i.e., the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can often be speeded by using an “intelligent” ranking function. It is also called an approximate cost function “Ĉ”.

Expended node (E-node) is the live node with the best Ĉ value.

Branching: A set of solutions, which is represented by a node, can be partitioned into mutually (jointly or commonly) exclusive (special) sets. Each subset in the partition is represented by a child of the original node.

Lower bounding: An algorithm is available for calculating a lower bound on the cost of any solution in a given subset.

Each node X in the search tree is associated with a cost: Ĉ(X)

C=cost of reaching the current node, X(E-node) form the root + The cost of reaching an answer node form X.

Ĉ=g(X)+H(X).

Example:

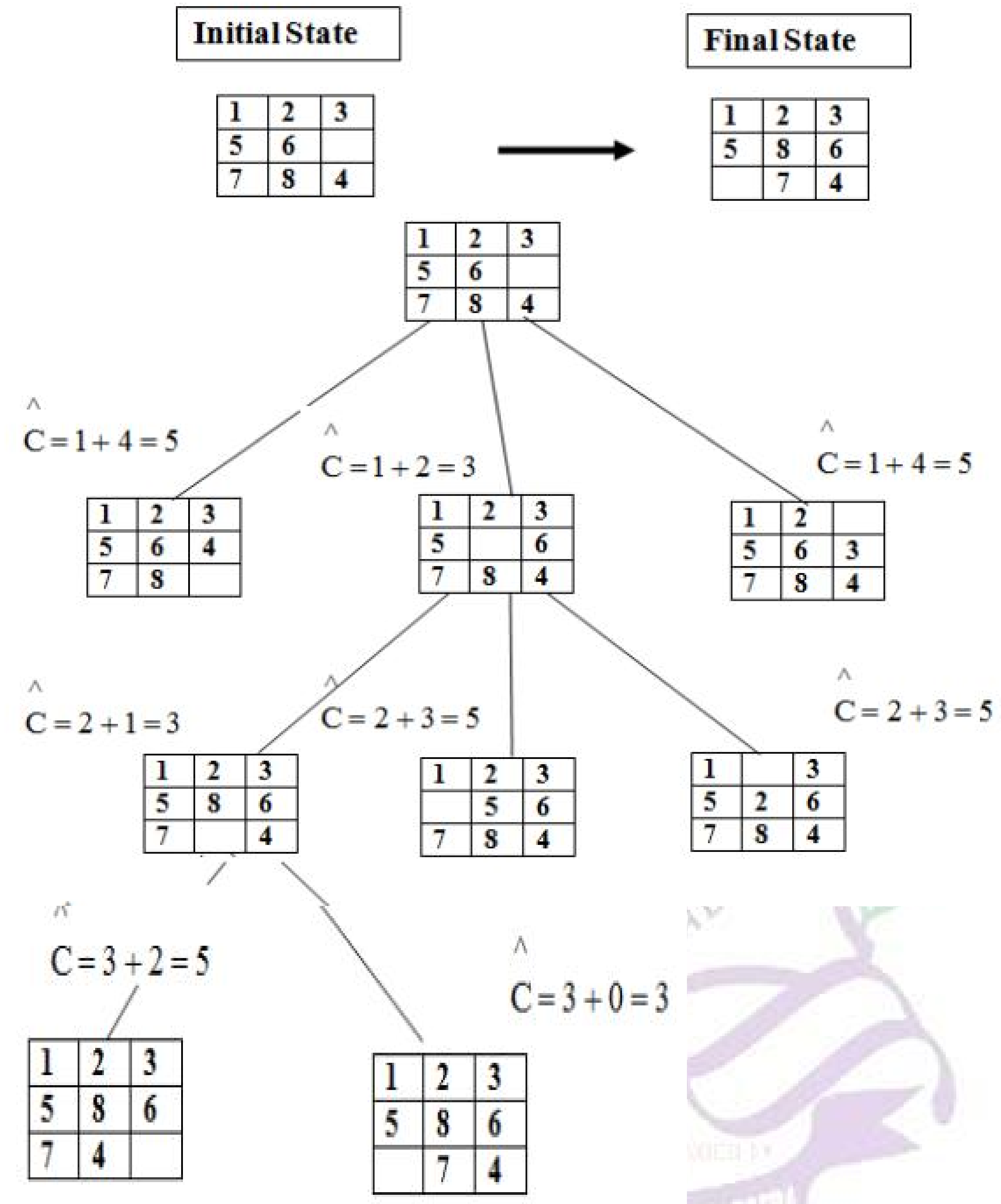
8-puzzle

Cost function: Ĉ = g(x) +h(x)

where h(x) = the number of misplaced tiles

and g(x) = the number of moves so far

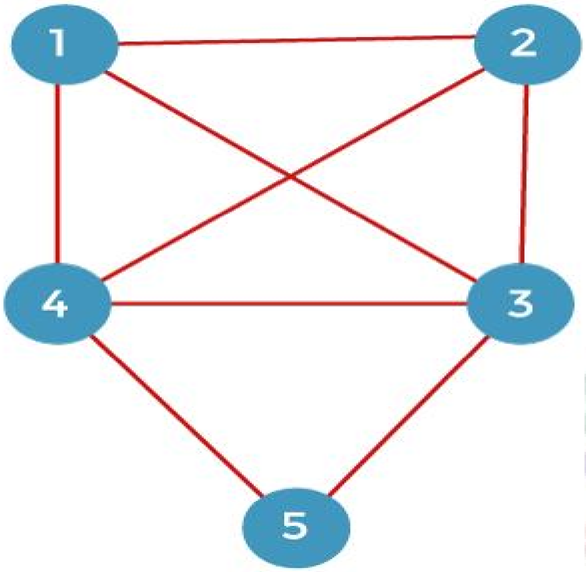
Assumption: move one tile in any direction cost 1.



Note: In case of tie, choose the leftmost node.

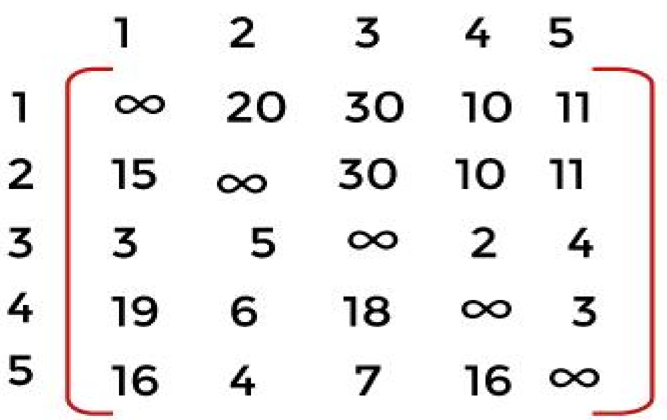
Traveling Salesperson problem using branch and bound:

Given the vertices, the problem here is that we have to travel each vertex exactly once and reach back to the starting point. Consider the below graph:



As we can observe in the above graph that there are 5 vertices given in the graph. We have to find the shortest path that goes through all the vertices once and returns back to the starting vertex. We mainly consider the starting vertex as 1, then traverse through the vertices 2, 3, 4, and 5, and finally return to vertex 1.

The adjacent matrix of the problem is given below:

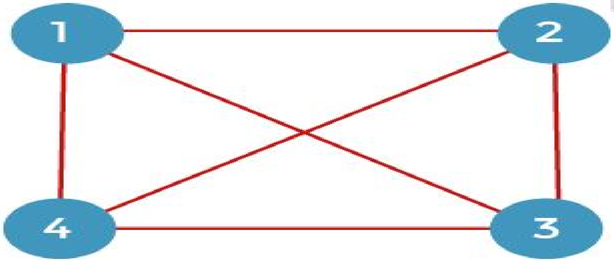


$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 20 & 30 & 10 & 11 \\ 2 & 15 & \infty & 30 & 10 & 11 \\ 3 & 3 & 5 & \infty & 2 & 4 \\ 4 & 19 & 6 & 18 & \infty & 3 \\ 5 & 16 & 4 & 7 & 16 & \infty \end{array}$$

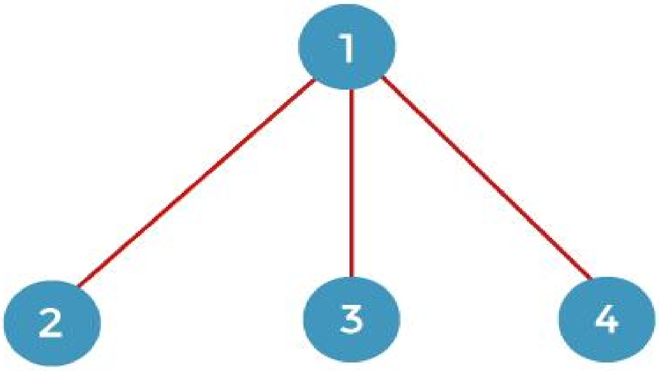
Now we look at how this problem can be solved using the branch n bound.

Let's first understand the approach then we solve the above problem.

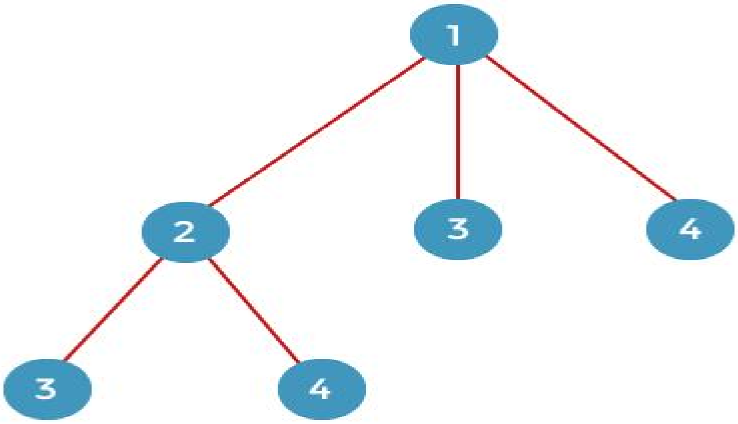
The graph is given below, which has four vertices:



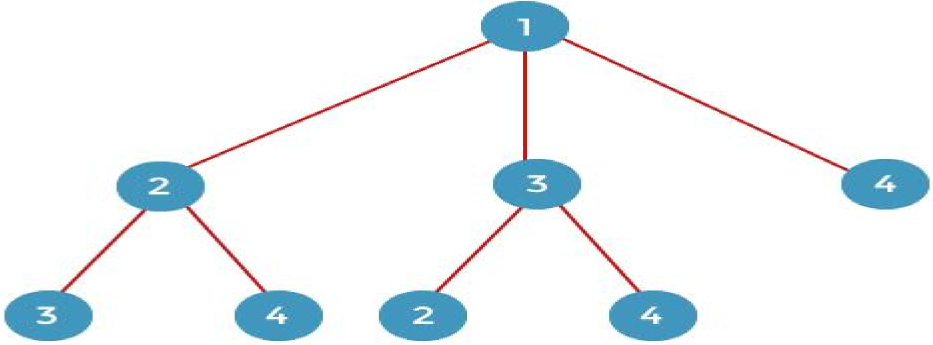
Suppose we start travelling from vertex 1 and return back to vertex 1. There are various ways to travel through all the vertices and returns to vertex 1. We require some tools that can be used to minimize the overall cost. To solve this problem, we make a state space tree. From the starting vertex 1, we can go to either vertices 2, 3, or 4, as shown in the below diagram.



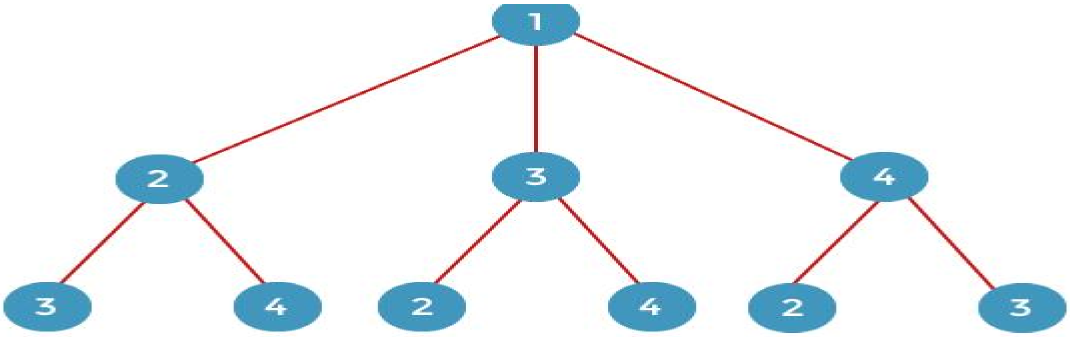
From vertex 2, we can go either to vertex 3 or 4. If we consider vertex 3, we move to the remaining vertex, i.e., 4. If we consider the vertex 4 shown in the below diagram:



From vertex 3, we can go to the remaining vertices, i.e., 2 or 4. If we consider the vertex 2, then we move to remaining vertex 4, and if we consider the vertex 4 then we move to the remaining vertex, i.e., 3 shown in the below diagram:

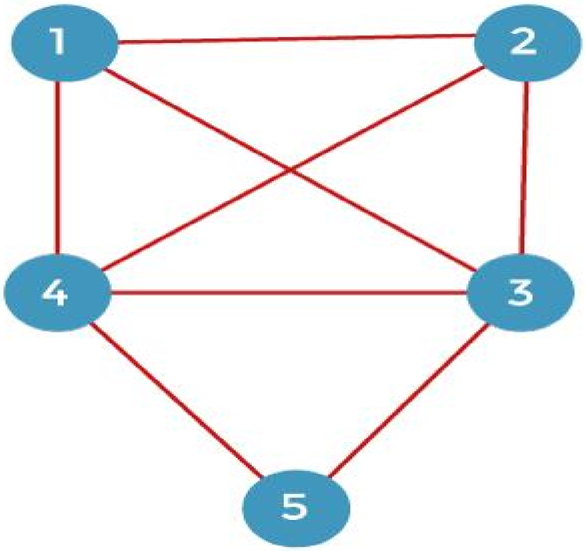


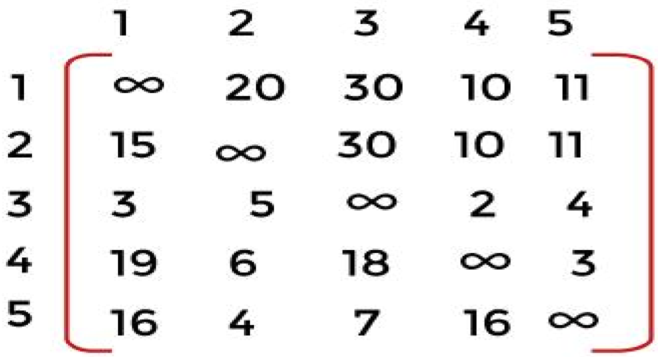
From vertex 4, we can go to the remaining vertices, i.e., 2 or 3. If we consider vertex 2, then we move to the remaining vertex, i.e., 3, and if we consider the vertex 3, then we move to the remaining vertex, i.e., 2 shown in the below diagram:



The above is the complete state space tree. The state space tree shows all the possibilities. Backtracking and branch n bound both use the state space tree, but their approach to solve the problem is different. Branch n bound is a better approach than backtracking as it is more efficient. In order to solve the problem using branch n bound, we use a level order. First, we will observe in which order, the nodes are generated. While creating the node, we will calculate the cost of the node simultaneously. If we find the cost of any node greater than the upper bound, we will remove that node. So, in this case, we will generate only useful nodes but not all the nodes.

Let's consider the above problem.:





$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 20 & 30 & 10 & 11 \\ 2 & 15 & \infty & 30 & 10 & 11 \\ 3 & 3 & 5 & \infty & 2 & 4 \\ 4 & 19 & 6 & 18 & \infty & 3 \\ 5 & 16 & 4 & 7 & 16 & \infty \end{array}$$

As we can observe in the above adjacent matrix that 10 is the minimum value in the first row, 2 is the minimum value in the second row, 2 is the minimum value in the third row, 3 is the minimum value in the third row, 3 is the minimum value in the fourth row, and 4 is the minimum value in the fifth row.

Now, we will reduce the matrix. We will subtract the minimum value with all the elements of a row. First, we evaluate the first row. Let's assume two variables, i.e., i and j, where 'i' represents the rows, and 'j' represents the columns.

When i = 0, j =0

M[0][0] = ∞-10= ∞

When i = 0, j = 1

M[0][1] = 20 - 10 = 10

When i = 0, j = 2

M[0][2] = 30 - 10 = 20

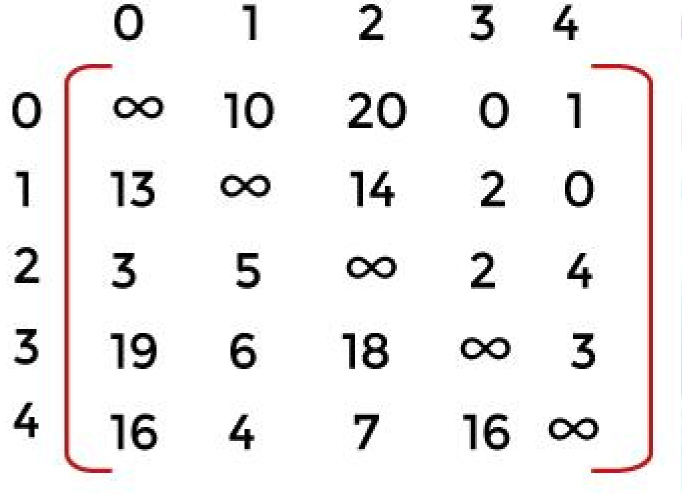
When i = 0, j = 3

M[0][3] = 10 - 10 = 0

When i = 0, j = 4

M[0][4] = 11 - 10 = 1

The matrix is shown below after the evaluation of the first row:



$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \infty & 10 & 20 & 0 & 1 \\ 1 & 13 & \infty & 14 & 2 & 0 \\ 2 & 3 & 5 & \infty & 2 & 4 \\ 3 & 19 & 6 & 18 & \infty & 3 \\ 4 & 16 & 4 & 7 & 16 & \infty \end{array}$$

Consider the second row.

When i = 1, j =0

M[1][0] = 15-2= 13

When i = 1, j = 1

M[1][1] = ∞ - 2= ∞

When i = 1, j = 2

M[1][2] = 16 - 2 = 14

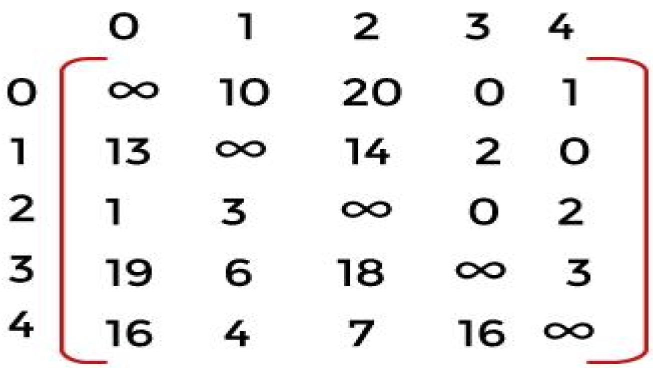
When i = 1, j = 3

M[1][3] = 4 - 2 = 2

When i = 1, j = 4

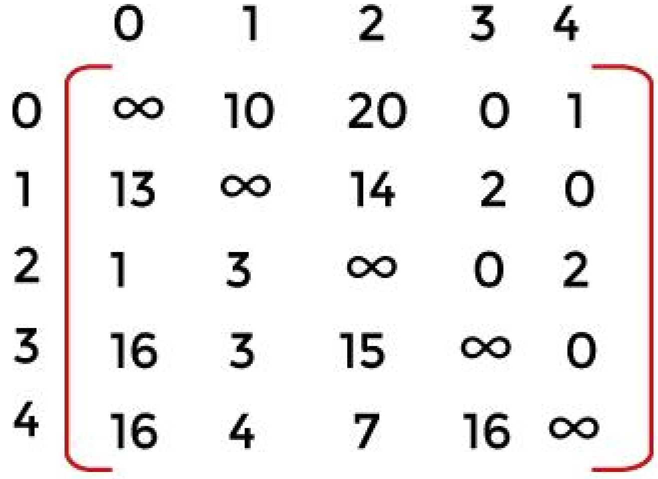
M[1][4] = 2 - 2 = 0

The matrix is shown below after the evaluation of the second row:



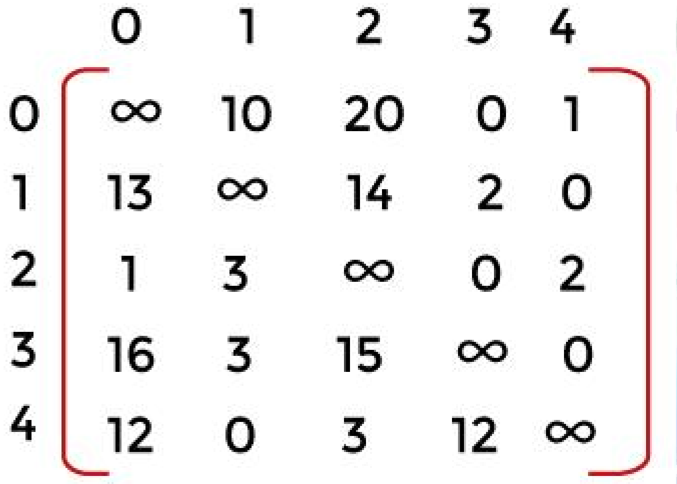
$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \infty & 10 & 20 & 0 & 1 \\ 1 & 13 & \infty & 14 & 2 & 0 \\ 2 & 1 & 3 & \infty & 0 & 2 \\ 3 & 19 & 6 & 18 & \infty & 3 \\ 4 & 16 & 4 & 7 & 16 & \infty \end{array}$$

Consider the third row:
When i = 2, j =0
M[2][0] = 3-2= 1
When i = 2, j = 1
M[2][1] = 5 - 2= 3
When i = 2, j = 2
M[2][2] = ∞ - 2 = ∞
When i = 2, j = 3
M[2][3] = 2 - 2 = 0
When i = 2, j = 4
M[2][4] = 4 - 2 = 2
The matrix is shown below after the evaluation of the third row:
Consider the fourth row:
When i = 3, j =0
M[3][0] = 19-3= 16
When i = 3, j = 1
M[3][1] = 6 - 3= 3
When i = 3, j = 2
M[3][2] = 18 - 3 = 15
When i = 3, j = 3
M[3][3] = ∞ - 3 = ∞
When i = 3, j = 4
M[3][4] = 3 - 3 = 0
The matrix is shown below after the evaluation of the fourth row:



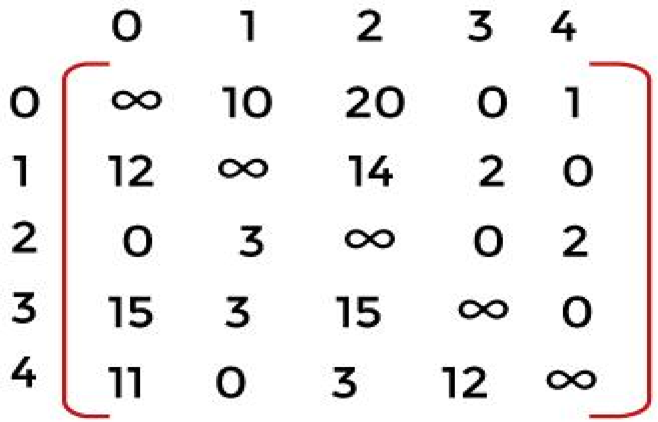
$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \infty & 10 & 20 & 0 & 1 \\ 1 & 13 & \infty & 14 & 2 & 0 \\ 2 & 1 & 3 & \infty & 0 & 2 \\ 3 & 16 & 3 & 15 & \infty & 0 \\ 4 & 16 & 4 & 7 & 16 & \infty \end{array}$$

Consider the fifth row:
When i = 4, j =0
M[4][0] = 16-4= 12
When i = 4, j = 1
M[4][1] = 4 - 4= 0
When i = 4, j = 2
M[4][2] = 7 - 4 = 3
When i = 4, j = 3
M[4][3] = 16 - 4 = 12
When i = 4, j = 4
M[4][4] = ∞ - 4 = ∞
The matrix is shown below after the evaluation of the fifth row:



$$\begin{array}{c|ccccc} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \infty & 10 & 20 & 0 & 1 \\ 1 & 13 & \infty & 14 & 2 & 0 \\ 2 & 1 & 3 & \infty & 0 & 2 \\ 3 & 16 & 3 & 15 & \infty & 0 \\ 4 & 12 & 0 & 3 & 12 & \infty \end{array}$$

The above matrix is the reduced matrix with respect to the rows.
Now we reduce the matrix with respect to the columns. Before reducing the matrix, we first find the minimum value of all the columns. The minimum value of first column is 1, the minimum value of the second column is 0, the minimum value of the third column is 3, the minimum value of the fourth column is 0, and the minimum value of the fifth column is 0, as shown in the below matrix:
Now we reduce the matrix.
Consider the first column.
When i = 0, j =0
M[0][0] = ∞-1= ∞
When i = 1, j = 0
M[1][0] = 13 - 1= 12
When i = 2, j = 0
M[2][0] = 1 - 1 = 0
When i = 3, j = 0
M[3][0] = 16 - 1 = 15
When i = 4, j = 0
M[4][0] = 12 - 1 = 11
The matrix is shown below after the evaluation of the first column:



$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \infty & 10 & 20 & 0 & 1 \\ 1 & 12 & \infty & 14 & 2 & 0 \\ 2 & 0 & 3 & \infty & 0 & 2 \\ 3 & 15 & 3 & 15 & \infty & 0 \\ 4 & 11 & 0 & 3 & 12 & \infty \end{array}$$

Since the minimum value of the first and the third columns is non-zero, we will evaluate only first and third columns. We have evaluated the first column. Now we will evaluate the third column.

Consider the third column.

When i = 0, j =2

M[0][2] = 20-3= 17

When i = 1, j = 2

M[1][2] = 13 - 1= 12

When i = 2, j = 2

M[2][2] = 1 - 1 = 0

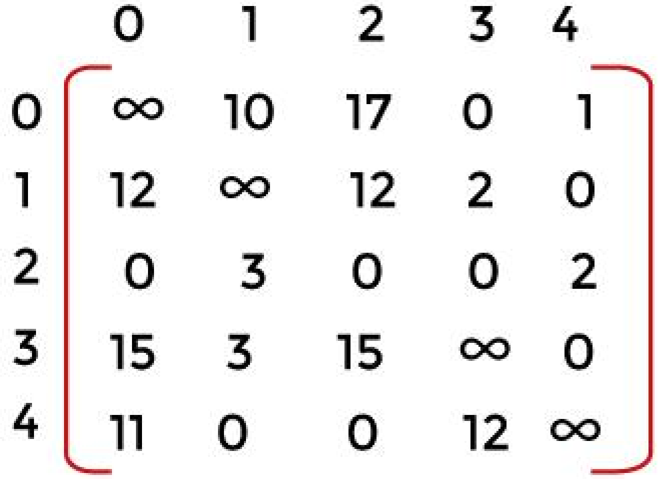
When i = 3, j = 2

M[3][2] = 16 - 1 = 15

When i = 4, j = 2

M[4][2] = 12 - 1 = 11

The matrix is shown below after the evaluation of the third column:

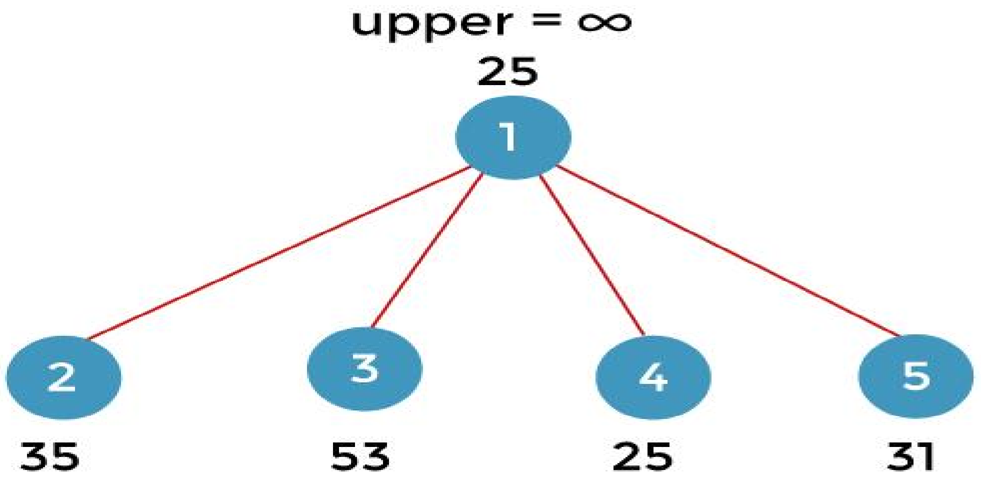


$$\begin{array}{c|ccccc|} & 0 & 1 & 2 & 3 & 4 \\ \hline 0 & \infty & 10 & 17 & 0 & 1 \\ 1 & 12 & \infty & 12 & 2 & 0 \\ 2 & 0 & 3 & 0 & 0 & 2 \\ 3 & 15 & 3 & 15 & \infty & 0 \\ 4 & 11 & 0 & 0 & 12 & \infty \end{array}$$

The above is the reduced matrix. The minimum value of rows is 21, and the columns is 4. Therefore, the total minimum value is (21 + 4) equals to 25.

Let's understand that how to solve this problem using branch and bound with the help of a state-space tree.

To make a state-space tree, first, we consider node 1. From node 1, we can go either to nodes 2, 3, 4, or 5 as shown in the below image. The cost of node 1 would be the cost which we achieved in the above-reduced matrix, i.e.., 25. Here, we will also maintain the upper bound. Initially, the upper bound would-be infinity.



Now, consider node 2. It means that we are moving from node 1 to node 2. Make the first row and second column as infinity shown in the below matrix:

Once we move from node 1 to node 2, we cannot move back to node 1. Therefore, we have to make 2 to 1 as infinity shown in the below matrix:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & 12 & 2 & 0 \\ 3 & 0 & \infty & 0 & 0 & 2 \\ 4 & 15 & \infty & 15 & \infty & 0 \\ 5 & 11 & \infty & 0 & 12 & \infty \end{array}$$

Since each row and column contains at least one zero value; therefore, we can say that above matrix has been reduced. The cost of reduction of node 2 is c(1, 2) + r + r` = 10 + 25 + 0 = 35.

Now we will find the minimum value of each column of the new reduced matrix. The minimum value of the first column is 11 and the minimum value of other three columns is 0.

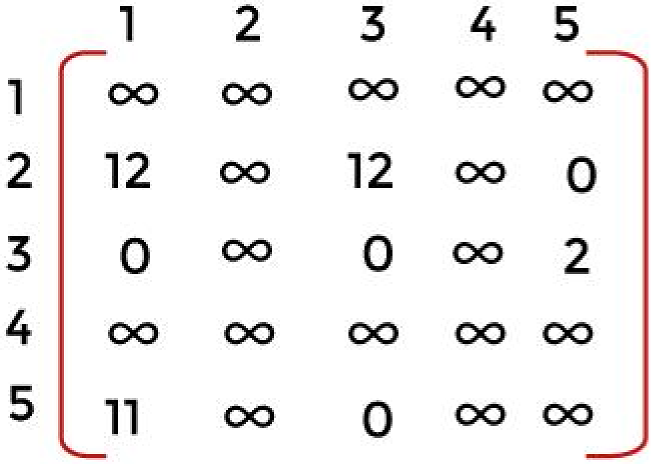
Now, consider the node 3. It means that we are moving from the node 1 to node 3. Make the first row and third column as infinity shown in the below matrix:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 12 & \infty & \infty & 2 & 0 \\ 3 & 0 & 3 & \infty & 0 & 2 \\ 4 & 15 & 3 & \infty & \infty & 0 \\ 5 & 11 & 0 & \infty & 12 & \infty \end{array}$$

Once we move from the node 1 to node 3, we cannot move back to the node 1. Therefore, we have to make 3 to 1 as infinity shown in the below matrix:

Since each row and column contains atleast one zero value; therefore, we can say that above matrix has been reduced. The cost of reduction of node 3 is c(1, 3) + r + r` = 17 + 25 + 11= 53.

Now, consider the node 4. It means that we are moving from the node 1 to node 4. Make the first row and forth column as infinity shown in the below matrix:



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & 12 & \infty & 12 & \infty & 0 \\ 3 & 0 & \infty & 0 & \infty & 2 \\ 4 & \infty & \infty & \infty & \infty & \infty \\ 5 & 11 & \infty & 0 & \infty & \infty \end{array}$$

Once we move from the node 1 to node 4, we cannot move back to the node 1. Therefore, we have to make 4 to 1 as infinity shown in the below matrix:

Since each row and column contains atleast one zero value; therefore, we can say that above matrix has been reduced. The cost of reduction of node 4 is c(1, 4) + r + r` = 0 + 25 + 0 = 25.

Now, consider the node 5. It means that we are moving from the node 1 to node 5. Make the first row and fifth column as infinity shown in the below matrix:

Once we move from the node 1 to node 5, we cannot move back to the node 1. Therefore, we have to make 5 to 1 as infinity shown in the below matrix:

Since each row and column contains atleast one zero value; therefore, we can say that above matrix has been reduced. In this case, second and third rows are non-zero. Therefore, we have to first find the minimum values of both the rows. The minimum value of second row is 2; therefore, we subtract 2 from all the elements of the second row. The elements of second row would be:

A[1][0] = 12-2 = 10

A[1][1] = ∞

A[1][2] = 11 - 2 = 9

A[1][3] = 2 - 2 = 0

A[1][4] = ∞ - 2 = ∞

As we can observe now that the second row contains one zero value.

The cost of reduction of node 5 is c(1, 5) + r + r` = 1 + 25 + 5 = 31

Since the node 4 has the minimum cost, i.e., 25. So we will explore the node 4 first. From the vertex 4, we can go either to the vertex 2, 3 or 5 as shown in the below image:

Now we have to calculate the cost of the path from the vertex 4 to 2, vertex 4 to 3, and vertex 4 to 5. Here, we will use the matrix of node 4 to find the cost of all the nodes.

First, we consider the path from the vertex 4 to the vertex 2. We make fourth row as ∞ and second column as ∞. Since we cannot move back from 2 to 1, so 1 to 2 is also infinity as shown in the below matrix:

Since all the rows and columns have atleast one zero value. Therefore, we can say that this matrix is already reduced. So, there would be no reduction cost. The cost of reduction of node 2 is c(4, 2) + r + r` = 3 + 25 + 0 = 28

Now we have to calculate the cost of the path from the vertex 4 to the vertex 3. We make fourth row and third column as infinity as shown in the below matrix. Since we cannot move from the vertex 3 to 1, so we make 3 to 1 as infinity shown in the below matrix:

Now we will check whether each row and column contain atleast one zero value or not. First, we observe all the rows. Since the third row does not have a zero value, so we first find the minimum value of the third row. The minimum value of the third row is 2, so we subtract 2 from all the elements of the third row. The elements of third row would be:

A[2][0] = ∞ - 2 = ∞
A[2][1] = 3 - 2 = 1
A[2][2] = ∞ - 2 = ∞
A[2][3] = ∞ - 2 = ∞
A[2][4] = 2 - 2 = 0

As we can observe now that the third row contains one zero value.

The first column does not contain the zero value. The minimum value of the first column is 11. We subtract 11 from all the elements of the first column. The elements of first column would be:

A[0][0] = ∞ - 11 = ∞
A[1][0] = 12 - 11 = 1
A[2][0] = ∞ - 11= ∞
A[3][0] = ∞ - 11= ∞
A[4][0] = 11 - 11 = 0

As we can observe now that the first column contains one zero value. The total minimum cost is 11 +2 equals to 13. The cost of reduction of node 3 is c(4, 3) + r + r` = 12 + 25 + 13 = 50.

Now we will calculate the cost of the path from the vertex 4 to 5. We make fourth row and fifth column as infinity. Since we cannot move back from the node 5 to 1, so we also make 1 to 5 as infinity shown in the below matrix:

Now we will check whether each row and column contain atleast one zero value or not. First, we observe all the rows. The second row does not contain the zero value, so we find the minimum value of the second row. The minimum value is 11 so we subtract 11 from all the elements of the second row. The elements of second row would be:

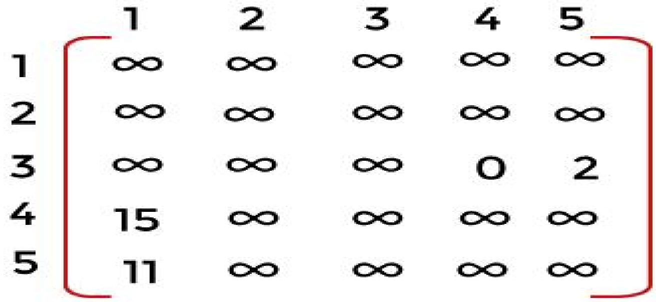
A[1][0] = 12 - 11 = 1
A[1][1] = ∞ - 11 = ∞
A[1][2] = 11 - 11 = 0
A[1][3] = ∞ - 11 = ∞
A[1][4] = ∞ - 11 = ∞

As we can observe now that the second row contains one zero value. The cost of reduction of node 5 is c(4, 5) + r + r` = 0 + 25 + 11 = 36.

Now we will compare the cost of all the leaf nodes. The node with a cost 28 is minimum so we will explore this node. The node with a cost 28 can be further expanded to the nodes 3 and 5 as shown in the below figure:

Now we have to calculate the cost of both the nodes, i.e., 3 and 5. First we consider the path from node 2 to node 3. Consider the matrix of node 2 which is shown below:

We make second row and third column as infinity. Also, we cannot move back from the node 3 to node 1 so we make 3 to 1 as infinity as shown in the below matrix:



$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & \infty & \infty & \infty & \infty \\ 2 & \infty & \infty & \infty & \infty & \infty \\ 3 & \infty & \infty & \infty & 0 & 2 \\ 4 & 15 & \infty & \infty & \infty & \infty \\ 5 & 11 & \infty & \infty & \infty & \infty \end{array}$$

Now we will check whether any row contains zero value or not. Since third row does not contain any zero value so we will find the minimum value of the third row. The minimum value of the third row is 2 so we subtract 2 from all the elements of the third row. The elements of third row would be:

A[2][0] = ∞ - 2 = ∞
A[2][1] = ∞ - 2 = ∞
A[2][2] = ∞ - 2 = ∞
A[2][3] = ∞ - 2 = ∞
A[2][4] = 2 - 2 = 0

Since fifth row does not contain any zero value so we will find the minimum value of the fifth row. The minimum value of the fifth row is 11 so we subtract 11 from all the elements of the fifth row.

A[4][0] = 11 - 11 = 0
A[4][1] = ∞ - 11 = ∞
A[4][2] = ∞ - 11 = ∞
A[4][3] = ∞ - 11 = ∞
A[4][4] = ∞ - 11 = ∞

The total minimum cost is (11 + 2) equals to 13. The cost of reduction of node 3 is c(2, 3) + r + r` = 11 + 28 + 13 = 52.

Consider the path from node 2 to node 5. Make the fourth row and third column as infinity. Since we cannot move back from the node 5 to node 1 so make 1 to 5 also as infinity as shown in the below matrix:

Now we will check whether any row contains zero value or not. Since every row and column contains a zero value; therefore, the above matrix is the reduced matrix.

The cost of reduction of node 5 is c(2, 5) + r + r` = 0 + 28 + 0 = 28

Now we will find out the leaf node with a minimum cost. The node 5 with a cost 28 is minimum so we select node 5 for the further exploration. The node 5 can be further expanded to the node 3 as shown in the below figure:

Here, we will use the matrix of node 5 having cost 28 as shown below:

Consider the path from node 5 to node 3. Make the fifth row and third column as infinity. Since we cannot move back from the node 3 to node 1 so make 1 to 5 also as infinity as shown in the below matrix:

Now we will check whether any row contains zero value or not. Since every row and column contains a zero value; therefore, the above matrix is the reduced matrix.

The cost of reduction of node 3 is c(5, 3) + r + r` = 0 + 28 + 0 = 28.

Finally, we traverse all the nodes. The upper value is updated from infinity to 28. We will check whether any leaf node has a value less than 28. Since no leaf node contains the value less than 28 so we discard all the leaf nodes from the tree as shown below:

The path of the tour would be 1->4->2->5->3.

O/1 Knapsack Problem

What is Knapsack Problem: Knapsack problem is a problem in combinatorial optimization, Given a set of items, each with a mass & a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit & the total value is as large as possible.

O-1 Knapsack Problem can formulate as. Let there be n items, Z1 to Zn where Zi has value Pi & weight wi

The maximum weight that can carry in the bag is m. All values and weights are non negative. Maximize the sum of the values of the items in the knapsack, so that sum of the weights must be less than the knapsack's capacity m.

The formula can be stated as

Xi=0 or 1 $1 \leq i \leq n$

To solve o/1 knapsack problem using B&B:

Knapsack is a maximization problem

Replace the objective function by the function to make it into a minimization problem

The modified knapsack problem is stated as

Fixed tuple size solution space:

o Every leaf node in state space tree represents an answer for which is an answer node; other leaf nodes are infeasible

o For optimal solution, define for every answer node x

For infeasible leaf nodes,

For non leaf nodes

c(x) = min{c(lchild(x)), c(rchild(x))}

Define two functions c^(x) and u(x) such that for every node x,

$c\hat{}(x) \leq c(x) \leq u(x)$

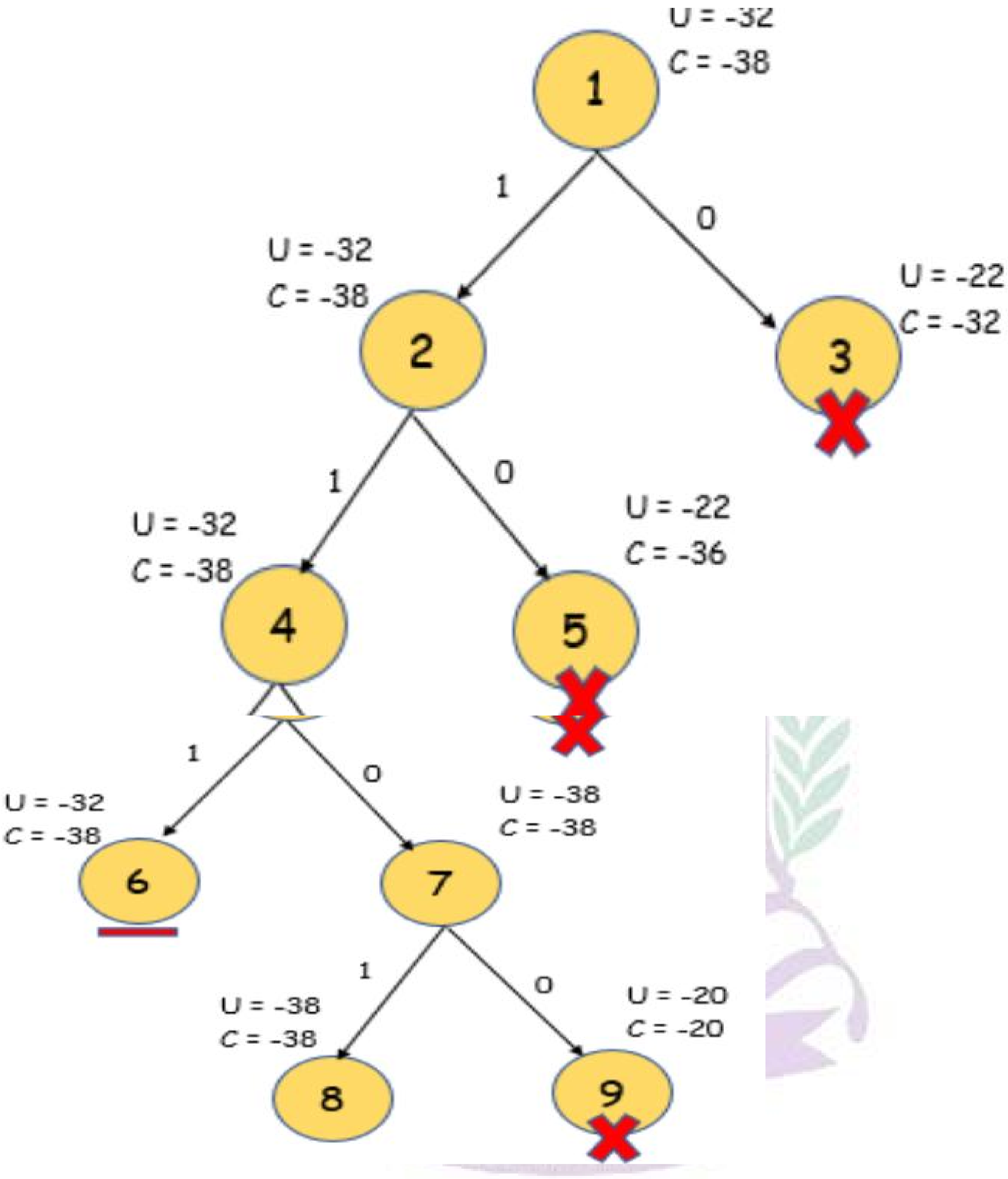
Computing c^(·) and u(·)

```
Algorithm ubound ( cp, cw, k, m )
{
// Input: cp: Current profit total
// Input: cw: Current weight total
// Input: k: Index of last removed item
// Input: m: Knapsack capacity
b=cp; c=cw;
for i:=k+1 to n do{
if(c+w[i] ≤ m) then {
c:=c+w[i]; b=b-p[i];
}
}
return b;
}
```

Solving an Example

Consider the problem with n =4, V = {10, 10, 12, 18}, w = {2, 4, 6, 9} and W = 15. Here, we calculate the initital upper bound to be U = 10 + 10 + 12 = 32. Note that the 4th object cannot be included here, since that would exceed W. For the cost, we add 3/9 th of the final value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.

After calculating the cost at each node, kill nodes that do not need exploring. Hence, the final state space tree will be as follows (Here, the number of the node denotes the order in which the state space tree was explored):



Note here that node 3 and node 5 have been killed after updating U at node 7. Also, node 6 is not explored further, since adding any more weight exceeds the threshold. At the end, only nodes 6 and 8 remain. Since the value of U is less for node 8, we select this node. Hence the solution is {1, 1, 0, 1}, and we can see here that the total weight is exactly equal to the threshold value in this case.

UNIT-V

NP-hard and NP-complete problems: Non Deterministic algorithms, The classes: P, NP, NP Complete, NP Hard, Satisfiability problem, Proofs for NP Complete Problems: Clique, Vertex Cover.
Parallel Algorithms: Introduction, models for parallel computing
Course Outcome:

- Describe the decidability and undecidabilty of problems, and understanding P,NP,NP complete and NP Hard problems

Non-deterministic algorithms

Non-deterministic algorithms, also known as randomized algorithms, are algorithms that use randomness in their computation. Unlike deterministic algorithms, which produce the same output for a given input every time they are executed, non-deterministic algorithms can produce different outputs or behaviors each time they are run, even for the same input.

The design and analysis of non-deterministic algorithms involve considering the average-case behavior or expected performance of the algorithm, rather than focusing solely on the worst-case scenario. Non-deterministic algorithms make use of random numbers or random choices during their execution to achieve certain properties, such as improved efficiency, simplicity, or probabilistic correctness.

Here are a few examples of non-deterministic algorithms and their applications:

- Randomized Quicksort: Quicksort is a widely used sorting algorithm. Randomized Quicksort improves upon the worst-case performance of the deterministic version by randomizing the choice of the pivot element. By choosing a random pivot, the algorithm achieves expected O(n log n) time complexity, even for certain inputs that would otherwise result in worst-case behavior.
- Monte Carlo Algorithms: These algorithms use random numbers to approximate solutions to problems that are otherwise computationally expensive to solve exactly. Monte Carlo methods are used in various fields, such as computational physics, finance, and optimization problems. The accuracy of the approximation improves as more random samples are taken.
- Randomized Prim's Algorithm: Prim's algorithm is used to find a minimum spanning tree in a connected weighted graph. In its randomized version, the algorithm selects edges randomly instead of always choosing the minimum-weight edge. Although the deterministic version guarantees correctness, the randomized version provides a faster average-case running time.
- Las Vegas Algorithms: These algorithms use randomness to improve efficiency by allowing the algorithm to terminate early in certain cases. Las Vegas algorithms guarantee correctness but have a variable running time. An example is the randomized algorithm for solving the k-SAT problem, which is a well-known NP-complete problem.
- Skip Lists: Skip lists are a probabilistic data structure that allows for efficient searching, insertion, and deletion operations in sorted lists. They are an alternative to balanced search trees and offer expected O(log n) time complexity for these operations.

When analyzing non-deterministic algorithms, a common approach is to consider their expected behavior, average-case complexity, or probabilistic guarantees. This involves analyzing the algorithm's performance over a range of possible inputs and taking into account the probabilities of different outcomes. Randomized algorithms provide a powerful toolset for solving complex problems efficiently and often yield practical solutions in real-world scenarios.

What are the major differences between deterministic and non-deterministic algorithms?

Deterministic algorithm	Non-deterministic algorithm
In a deterministic algorithm, the computer	In a non-deterministic algorithm, the computer

produces the same output while going through different steps.	may produce different outputs for the same input while going through different runs.
The deterministic algorithm can determine the next step.	The non-deterministic algorithm cannot solve problems in polynomial time, and cannot determine the next step.
The output is not random.	The output has a certain degree of randomness to it.
In a deterministic algorithm, the next step can be determined.	In a non-deterministic algorithm, the next step cannot be determined.

The classes : P, NP, NP complete, NP Hard

In the study of computational complexity, several classes of problems are defined based on the resources required to solve them. Here are the definitions of the classes P, NP, NP-complete, and NP-hard:

1. P (Polynomial Time): The class P represents the set of decision problems that can be solved by a deterministic Turing machine in polynomial time. In other words, these are the problems for which an algorithm exists that can solve them efficiently, with a time complexity bound of O(n^k), where n is the size of the input and k is a constant. Examples of problems in P include sorting, searching, and basic arithmetic operations.
2. NP (Non-deterministic Polynomial Time): The class NP represents the set of decision problems for which a potential solution can be verified in polynomial time by a deterministic Turing machine. In other words, given a solution, it can be verified efficiently. However, finding the solution itself may not be efficient. This class includes many important problems, such as the traveling salesman problem, the knapsack problem, and the graph coloring problem.
3. NP-complete (Nondeterministic Polynomial-time complete): A problem is NP-complete if it is both in NP and every problem in NP can be reduced to it in polynomial time. In simpler terms, an NP-complete problem is one where a solution can be verified in polynomial time and any other problem in NP can be transformed into it efficiently. The classic example of an NP-complete problem is the Boolean satisfiability problem (SAT).
4. NP-hard (Nondeterministic Polynomial-time hard): The class NP-hard contains problems that are at least as hard as the hardest problems in NP. Unlike NP-complete problems, NP-hard problems may not be in NP themselves. These problems do not necessarily have to be decision problems. Informally, an NP-hard problem is one that is "as hard as" or "harder than" the NP-complete problems in terms of computational complexity.

In summary, the class P represents problems that can be solved efficiently, NP represents problems that can be verified efficiently, NP-complete represents the hardest problems in NP, and NP-hard represents problems that are at least as hard as the hardest problems in NP. The relationship between these classes is that all NP-complete problems are NP-hard, but it is unknown whether P = NP (i.e., whether NP-complete problems can be solved in polynomial time or not).

The Classes NP-Hard & NP-Complete:

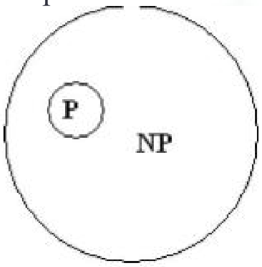
For measuring the complexity of an algorithm, we use the input length as the parameter. For example, An algorithm A is of polynomial complexity p() such that the computing time of A is O(p(n)) for every input of size n.

Decision problem/ Decision algorithm: Any problem for which the answer is either zero or one is decision problem. Any algorithm for a decision problem is termed a decision algorithm.

Optimization problem/ Optimization algorithm: Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

P-) is the set of all decision problems solvable by deterministic algorithms in polynomial time.

NP-) is the set of all decision problems solvable by nondeterministic algorithms in polynomial time. Since deterministic algorithms are just a special case of nondeterministic, by this we concluded that P ⊆ NP



Commonly believed relationship between P & NP

The most famous unsolvable problems in Computer Science is Whether P=NP or P≠NP In considering this problem, s.cook formulated the following question.

If there any single problem in NP, such that if we showed it to be in 'P' then that would imply that P=NP.

Cook answered this question with Theorem:

Satisfiability is in P if and only if (iff) P=NP -)Notation of Reducibility

Let L1 and L2 be problems, Problem L1 reduces to L2 (written L1 α L2) iff there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time

This implies that, if we have a polynomial time algorithm for L2, Then we can solve L1 in polynomial time.

Here α-) is a transitive relation i.e., **L1 α L2 and L2 α L3 then L1 α L3**

A problem L is NP-Hard if and only if (iff) satisfiability reduces to L ie., **Statisfiability α L**

A problem L is NP-Complete if and only if (iff) L is NP-Hard and **L Є NP**

Cook's Theorem:

States that satisfiability is in P if and only if P=NP If P=NP then satisfiability is in P If satisfiability is in P, then P=NP To do this >

- A-) Any polynomial time nondeterministic decision algorithm. I-)Input of that algorithm Then formula Q(A, I), Such that Q is satisfiable iff 'A' has a successful termination with Input I.

- If the length of ‘I’ is ‘n’ and the time complexity of A is p(n) for some polynomial p() then length of Q is O(p3 (n) log n)=O(p4 (n)) The time needed to construct Q is also O(p3 (n) log n).
- A deterministic algorithm ‘Z’ to determine the outcome of ‘A’ on any input ‘I’ Algorithm Z computes ‘Q’ and then uses a deterministic algorithm for the satisfiability problem to determine whether ‘Q’ is satisfiable.
- If O(q(m)) is the time needed to determine whether a formula of length ‘m’ is satisfiable then the complexity of ‘Z’ is O(p3 (n) log n + q(p3 (n)log n)). > If satisfiability is ‘p’, then ‘q(m)’ is a polynomial function of ‘m’ and the complexity of ‘Z’ becomes ‘O(r(n))’ for some polynomial ‘r()’.
- Hence, if satisfiability is in p, then for every nondeterministic algorithm A in NP, we can obtain a deterministic Z in p.

By this we shows that satisfiability is in p then P=NP

Satisfiability (SAT) Problem

- Boolean formula f(a_1, a_2, …a_n) is satisfiable if there is a way to assign values to a variable of the formula such that the function evaluates to *true*. If this is the case, the function is ***satisfiable***.
- On the other hand, if no such assignment is possible, the function is *unsatisfiable*.
- The expression (A ^ ¬B) is satisfiable for A = 1 and B = 0, but the expression (A ^ ¬A) is unsatisfiable.
- No known algorithm exists that solves the problem efficiently.

Example: f(x, y, z) = (x ∨ (y ∧ z)) ∧ (x ∧ z)

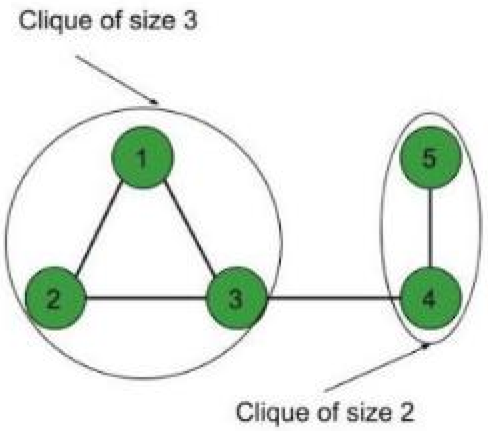
X	**y**	**z**	x ∨ (y ∧ z)	x ∧ z	**f(x, y, z)**
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	0	0
1	1	1	1	1	1

- The function is true for the combinations (x = 1, y = 0, z = 1) and (x = y = z = 1), hence it is satisfiable.
- Given the input sequence, it can be verified in linear time. But with n input n variables, there exist 2^n instances. Testing each of them takes O(n.2^n) time, which speaks SAT ∈ NP.

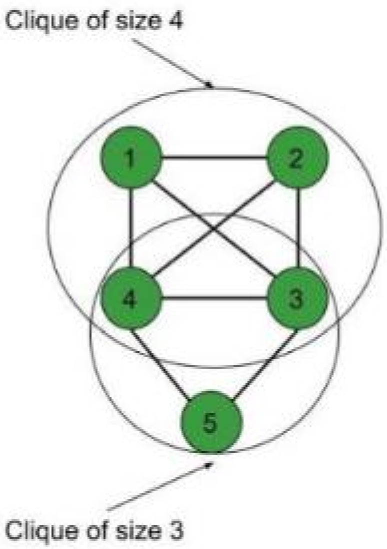
Proofs for NP complete Problems:

Clique :

A clique is a subgraph of a graph such that all the vertices in this subgraph are connected with each other that is the subgraph is a complete graph. The Maximal Clique Problem is to find the maximum sized clique of a given graph G, that is a complete graph which is a subgraph of G and contains the maximum number of vertices. This is an optimization problem. Correspondingly, the Clique Decision Problem is to find if a clique of size k exists in the given graph or not.



The above graph contains a maximum clique of size 3



The above graph contains a maximum clique of size 4

To prove that a problem is NP-Complete, we have to show that it belongs to both NP and NP-Hard Classes. (Since NP-Complete problems are NP-Hard problems which also belong to NP)

The Clique Decision Problem belongs to NP – If a problem belongs to the NP class, then it should have polynomial-time verifiability, that is given a certificate, we should be able to verify in polynomial time if it is a solution to the problem.

Proof:

1. Certificate – Let the certificate be a set S consisting of nodes in the clique and S is a subgraph of G.
2. Verification – We have to check if there exists a clique of size k in the graph. Hence, verifying if number of nodes in S equals k, takes O(1) time. Verifying whether each vertex has an out-degree of (k-1) takes $O(k^2)$ time. (Since in a complete graph, each vertex is connected to every other vertex through an edge. Hence the total number of edges in a complete graph = ${}^kC_2$ = k*(k-1)/2). Therefore, to check if the graph formed by the k nodes in S is complete or not, it takes $O(k^2) = O(n^2)$ time (since k<=n, where n is number of vertices in G).

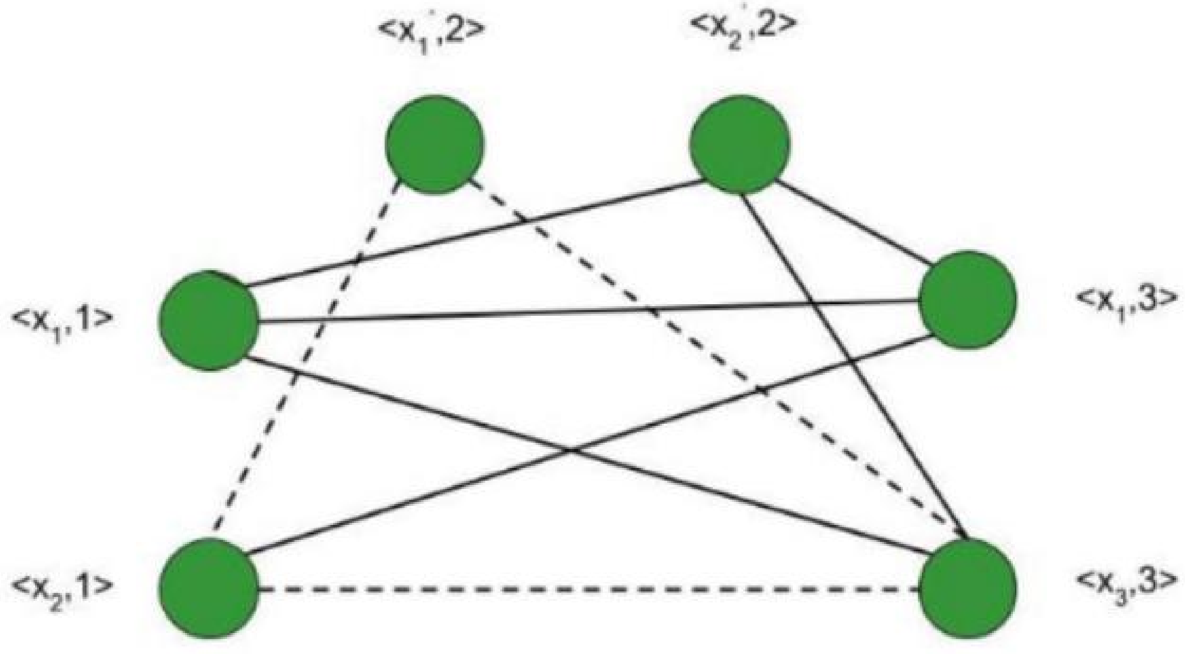
Therefore, the Clique Decision Problem has polynomial time verifiability and hence belongs to the NP Class.

The Clique Decision Problem belongs to NP-Hard – A problem L belongs to NP-Hard if every NP problem is reducible to L in polynomial time. Now, let the Clique Decision Problem by C. To prove that C is NP-Hard, we take an already known NP-Hard problem, say S, and reduce it to C for a particular instance. If this reduction can be done in polynomial time, then C is also an NP-Hard problem. The Boolean Satisfiability Problem (S) is an NP-Complete problem as proved by the Cook's theorem. Therefore, every problem in NP can be reduced to S in polynomial time. Thus, if S is reducible to C in polynomial time, every NP problem can be reduced to C in polynomial time, thereby proving C to be NP-Hard.

Proof that the Boolean Satisfiability problem reduces to the Clique Decision Problem

Let the boolean expression be – F = (x_1 v x_2) ^ (x_1‘ v x_2‘) ^ (x_1 v x_3) where x_1, x_2, x_3 are the variables, ‘^’ denotes logical ‘and’, ‘v’ denotes logical ‘or’ and x’ denotes the complement of x. Let the expression within each parentheses be a clause. Hence we have three clauses – C_1, C_2 and C_3. Consider the vertices as – <x_1, 1>; <x_2, 1>; <x_1’, 2>; <x_2’, 2>; <x_1, 3>; <x_3, 3> where the second term in each vertex denotes the clause number they belong to. We connect these vertices such that –

1. No two vertices belonging to the same clause are connected.
2. No variable is connected to its complement



Thus, the graph G (V, E) is constructed such that – V = { <a, i> | a belongs to C_i } and E = { (<a, i>, <b, j>) | i is not equal to j ; b is not equal to a' } Consider the subgraph of G with the vertices <x_2, 1>; <x_1', 2>; <x_3, 3>. It forms a clique of size 3 (Depicted by dotted line in above figure) . Corresponding to this, for the assignment – <x_1, x_2, x_3> = <0, 1, 1> F evaluates to true. Therefore, if we have k clauses in our satisfiability expression, we get a max clique of size k and for the corresponding assignment of values, the satisfiability expression evaluates to true. Hence, for a particular instance, the satisfiability problem is reduced to the clique decision problem. Therefore, the Clique Decision Problem is NP-Hard.

Vertex cover:

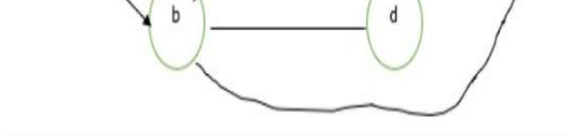
To prove VC is NP-complete we have to prove the following −

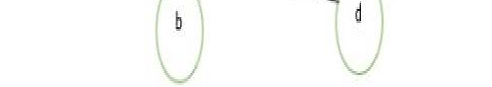
- VC is Non-deterministic Polynomial (NP).
- A NPC problem can be reduced into VC.

To prove VC is NP, find a verifier which is a subset of vertices which is VC and that can be verified in polynomial time. For a graph of n vertices it can be proved in O(n2). Thus, VC is NP.

Now consider the "clique" problem which is NPC and reduce it into VC to prove NPC. Clique of a graph G is a subset of vertices such that these vertices form a complete subgraph in the given graph G.

The two graphs titles (a) and (b) are given below for the VC problem –





(a) (b)

Consider graph (a), here clique is {a,b,c,d}.
Now calculate a graph as shown in (b) which is as follows −
Completed graph of all the vertices in (a) – (a)
For the graph (b), we can say that the vertex cover is {s,t} which covers all the edges of the (b). This {s,t} = {a,b,c,d,s,t} – {Clique of (a)} Thus conversely, we can say that we can reduce clique into the VC problem and conversely can find both VC and clique of a given undirected graph. This means VC is NP-complete reducible. Hence proved that VC is an NPC.

Parallel Algorithms:

Parallel algorithms play a crucial role in the field of Design and Analysis of Algorithms (DAA) as they focus on designing algorithms that can be executed on parallel computing architectures, where multiple processors or cores work together to solve a problem more efficiently. Parallel algorithms aim to exploit the available parallelism to achieve faster execution times and improved scalability.
Here are some common techniques and examples of parallel algorithms in DAA:

1. Parallel Divide and Conquer: The divide and conquer technique can be parallelized by dividing the problem into independent subproblems that can be solved concurrently on different processors. The results from the subproblems are then combined to obtain the final solution. Examples include parallel merge sort, parallel quicksort, and parallel matrix multiplication.
2. Parallel Prefix Sum: The prefix sum, also known as the scan operation, calculates the cumulative sum of elements in an array. The parallel prefix sum algorithm divides the array into smaller segments and performs prefix sum computations concurrently. This technique is useful in parallelizing algorithms like parallel sorting, parallel graph algorithms, and parallel dynamic programming.
3. Parallel Graph Algorithms: Many graph algorithms can be parallelized to take advantage of the inherent parallelism in graphs. For example, parallel breadth-first search (BFS) and parallel depth-first search (DFS) algorithms can explore different parts of the graph simultaneously on different processors. Other graph algorithms, such as minimum spanning tree, shortest path, and maximum flow algorithms, can also be parallelized.
4. Parallel Dynamic Programming: Dynamic programming algorithms, which solve problems by breaking them down into overlapping subproblems, can be parallelized to exploit parallelism. The key is to identify independent subproblems that can be solved concurrently. Parallel dynamic programming can be applied to problems such as sequence alignment, matrix chain multiplication, and knapsack problems.
5. Parallel Monte Carlo Algorithms: Monte Carlo methods involve generating random samples to approximate solutions to complex problems. Parallelization can be achieved by distributing the sampling and computation across multiple processors. Applications of parallel Monte Carlo algorithms include simulations, optimization problems, and computational physics.
6. Parallel Sorting: Sorting algorithms, such as parallel merge sort and parallel quicksort, can be designed to utilize multiple processors. The idea is to divide the input into smaller subproblems that can be sorted independently and then merge the sorted subproblems in parallel to obtain the final sorted result.

These are just a few examples of parallel algorithms in DAA. The design and analysis of parallel algorithms require careful consideration of load balancing, communication overhead, synchronization, and scalability issues. Performance metrics, such as speedup, efficiency, and scalability, are often used to evaluate the effectiveness of parallel algorithms in exploiting parallel computing resources.

Models of Parallel Computing

The model of a parallel algorithm is developed by considering a strategy for dividing the data and processing method and applying a suitable strategy to reduce interactions. In this chapter, we will discuss the following Parallel Algorithm Models −

- Data parallel model
- Task graph model
- Work pool model
- Master slave model
- Producer consumer or pipeline model
- Hybrid model

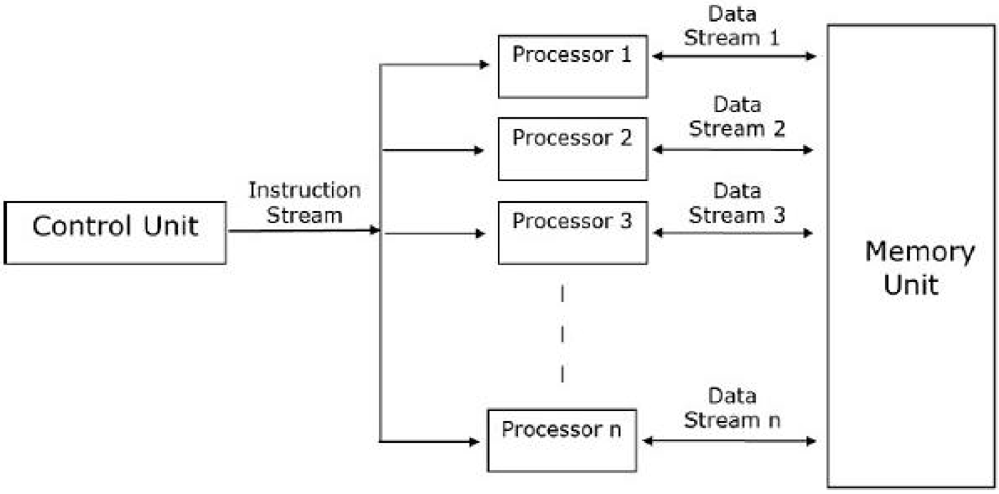
Data Parallel

In data parallel model, tasks are assigned to processes and each task performs similar types of operations on different data. Data parallelism is a consequence of single operations that is being applied on multiple data items.

Data-parallel model can be applied on shared-address spaces and message-passing paradigms. In data-parallel model, interaction overheads can be reduced by selecting a locality preserving decomposition, by using optimized collective interaction routines, or by overlapping computation and interaction.

The primary characteristic of data-parallel model problems is that the intensity of data parallelism increases with the size of the problem, which in turn makes it possible to use more processes to solve larger problems.

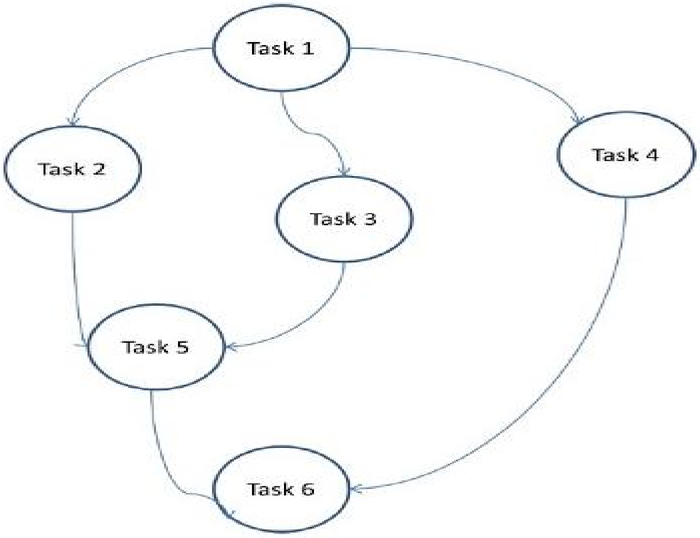
Example − Dense matrix multiplication.



Task Graph Model

In the task graph model, parallelism is expressed by a task graph. A task graph can be either trivial or nontrivial. In this model, the correlation among the tasks are utilized to promote locality or to minimize interaction costs. This model is enforced to solve problems in which the quantity of data associated with the tasks is huge compared to the number of computation associated with them. The tasks are assigned to help improve the cost of data movement among the tasks.

Examples − Parallel quick sort, sparse matrix factorization, and parallel algorithms derived via divide-and-conquer approach.



Here, problems are divided into atomic tasks and implemented as a graph. Each task is an independent unit of job that has dependencies on one or more antecedent task. After the completion of a task, the output of an antecedent task is passed to the dependent task. A task with antecedent task starts execution only when its entire antecedent task is completed. The final output of the graph is received when the last dependent task is completed (Task 6 in the above figure).

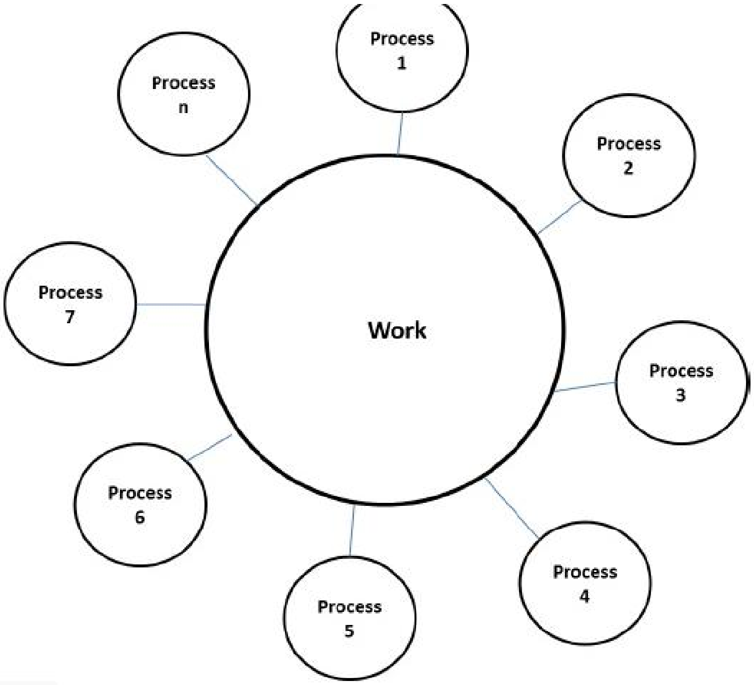
Work Pool Model

In work pool model, tasks are dynamically assigned to the processes for balancing the load. Therefore, any process may potentially execute any task. This model is used when the quantity of data associated with tasks is comparatively smaller than the computation associated with the tasks.

There is no desired pre-assigning of tasks onto the processes. Assigning of tasks is centralized or decentralized. Pointers to the tasks are saved in a physically shared list, in a priority queue, or in a hash table or tree, or they could be saved in a physically distributed data structure.

The task may be available in the beginning, or may be generated dynamically. If the task is generated dynamically and a decentralized assigning of task is done, then a termination detection algorithm is required so that all the processes can actually detect the completion of the entire program and stop looking for more tasks.

Example − Parallel tree search



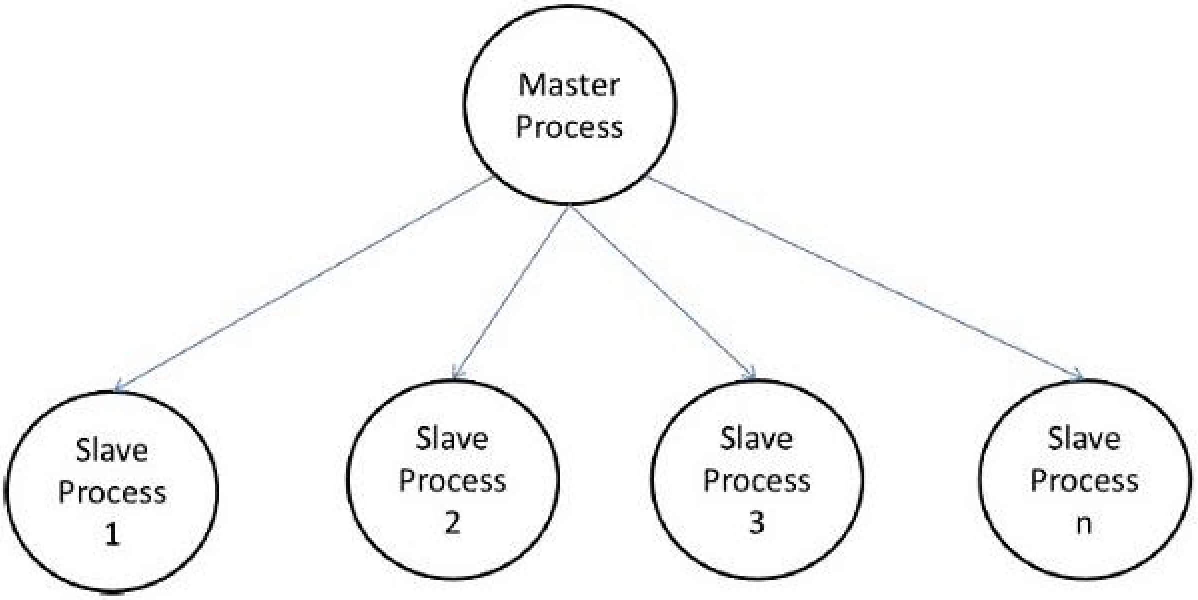
Master-Slave Model

In the master-slave model, one or more master processes generate task and allocate it to slave processes. The tasks may be allocated beforehand if −

- the master can estimate the volume of the tasks, or
- a random assigning can do a satisfactory job of balancing load, or
- slaves are assigned smaller pieces of task at different times.

This model is generally equally suitable to shared-address-space or message-passing paradigms, since the interaction is naturally two ways.

In some cases, a task may need to be completed in phases, and the task in each phase must be completed before the task in the next phases can be generated. The master-slave model can be generalized to hierarchical or multi-level master-slave model in which the top level master feeds the large portion of tasks to the second-level master, who further subdivides the tasks among its own slaves and may perform a part of the task itself.



Precautions in using the master-slave model

Care should be taken to assure that the master does not become a congestion point. It may happen if the tasks are too small or the workers are comparatively fast.

The tasks should be selected in a way that the cost of performing a task dominates the cost of communication and the cost of synchronization.

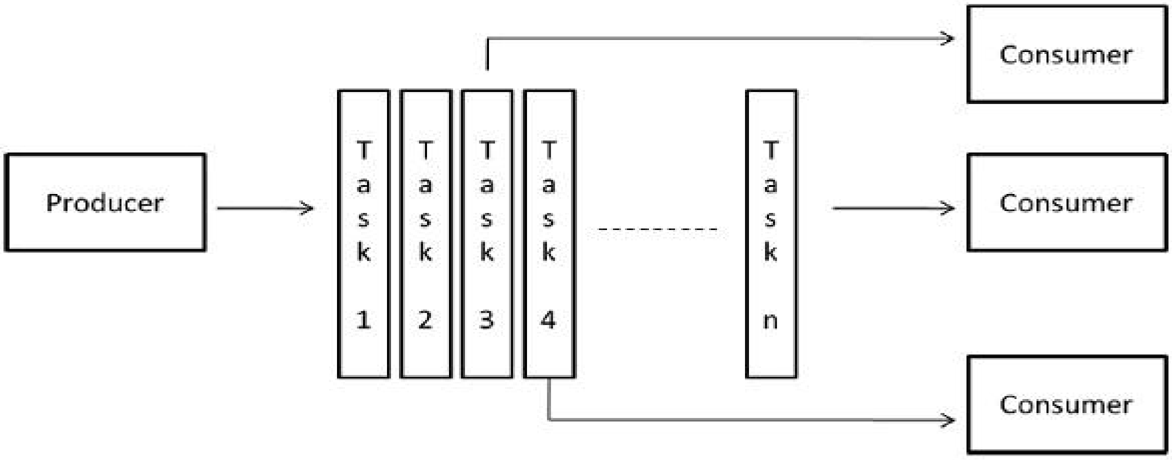
Asynchronous interaction may help overlap interaction and the computation associated with work generation by the master.

Pipeline Model

It is also known as the producer-consumer model. Here a set of data is passed on through a series of processes, each of which performs some task on it. Here, the arrival of new data generates the execution of a new task by a process in the queue. The processes could form a queue in the shape of linear or multidimensional arrays, trees, or general graphs with or without cycles.

This model is a chain of producers and consumers. Each process in the queue can be considered as a consumer of a sequence of data items for the process preceding it in the queue and as a producer of data for the process following it in the queue. The queue does not need to be a linear chain; it can be a directed graph. The most common interaction minimization technique applicable to this model is overlapping interaction with computation.

Example − Parallel LU factorization algorithm.



Hybrid Models

A hybrid algorithm model is required when more than one model may be needed to solve a problem.

A hybrid model may be composed of either multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.

Example − Parallel quick sort