IITT 63: DESIGN AND ANALYSIS OF ALGORITHMS

Unit - I

Introduction – Performance Analysis. Divide and conquer Method: Binary Search, Finding Maximum and Minimum, Merge Sort and Quick Sort.

Unit - II

Greedy Methods: Knapsack Problem, Minimum Cost Spanning Trees, Optimal Storage on Tapes and Single Source Shortest Path Problem.

Unit - III

Dynamic Programming: Multistage Graphs, 0/1 knapsack and Traveling Salesman Problem. Basic Traversal and Search Techniques: Techniques for Binary Tree, Techniques for Graphs: Depth First Search and Breadth First Search - Connected Components and Spanning Tree - Biconnected Components and DFS.

Unit - IV

Backtracking: 8 Queens Problems, Sum of Subsets, Graph Colouring, Hamiltonian Cycle and Knapsack Problem.

Unit - V

Branch and Bound: Least Cost Search. Bounding: FIFO Branch and Bound and LC Branch and Bound. 0/1 Knapsack Problem, Travelling Salesman Problem.

Text Books and References

1. E.Horowitz, S.Sahni and Sanguthevar Rajasekaran, Fundamentals of Computer Algorithms , Second edition, Universities Press.
2. S. K. Basu, Design Methods and Analysis of Algorithms , PHI, 2005.
3. Goodman and S. T. Hedetniem, Introduction to the Design and Analysis of Algorithms , MGH, 1977
4. A.V. Aho, J.D. Ullman and J.E.Hospcraft, The Design and Analysis of Computer Algorithms Pearson Education.

IITT 63: DESIGN AND ANALYSIS OF ALGORITHMS

Unit - I

Introduction – Performance Analysis. Divide and conquer Method: Binary Search, Finding Maximum and Minimum, Merge Sort and Quick Sort.

Introduction

What is an Algorithm?

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria,

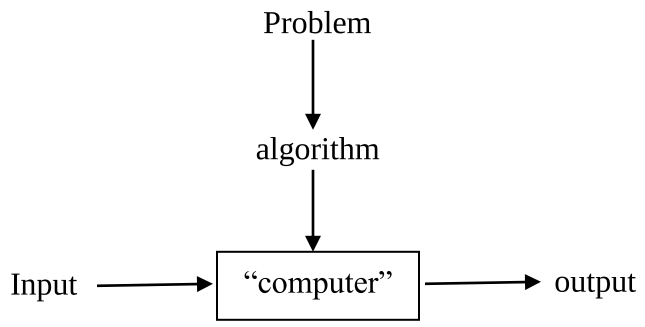
1. **Input:** zero or more quantities are externally supplied.
2. **Output:** At least one quantity is produced
3. **Definiteness (confidence):** Each instruction is clear and unambiguous (clear cut).
4. **Finiteness:** The algorithm must terminate after a finite number of steps.
5. **Effectiveness:** Each instruction is very basic, so that can be carried out in a finite amount of time and it must be feasible (sufficient).

Algorithms that are definite and effective are also called as ***computational procedures***.

For example: Operating system of a digital computer is one of the best examples of computational procedure. This procedure is designed to control the execution of jobs, in such a way that no jobs are available it does not terminate but continues in a waiting state until a new job is entered.

A program is the expression of an algorithm in a programming language.

Algorithms are written in programming languages to achieve the criterion of definiteness.



Notion(idea) of algorithm

Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program.

There are two approaches available to determine the performance of a program, are

- Analytical
- Experimental.

In performance analysis analytical methods are used, while in performance measurement experiments are conducted.

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 needed to execute the program successfully.

Actually the execution time of a program depends on a variety of backgrounds: like

- complexity of the problem
- the speed of the Computer
- the language in which the algorithm is implemented,
- the compiler/interpreter
- skill of the programmers etc.

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.

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.

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.

Algorithm Specification

Pseudocode Convention

Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm.

Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations.

It is used for creating an outline or a rough draft of a program.

Pseudocode summarizes a program's flow, but excludes underlying details.

Pseudocode is not an actual programming language. So it cannot be compiled into an executable program.

It uses short terms or simple English language syntaxes to write code for programs before it is actually converted into a specific programming language.

This is done to identify top level flow errors, and understand the programming data flows that the final program is going to use.

This definitely helps save time during actual programming as conceptual errors have been already corrected.

Firstly, program description and functionality is gathered and then pseudocode is used to create statements to achieve the required results for a program.

Detailed pseudocode is inspected and verified by the designer's team or programmers to match design specifications.

Catching errors or wrong program flow at the pseudocode stage is beneficial for development as it is less costly than catching them later.

Once the pseudocode is accepted by the team, it is rewritten using the vocabulary and syntax of a programming language.

The purpose of using pseudocode is an efficient key principle of an algorithm.

It is used in planning an algorithm with sketching out the structure of the program before the actual coding takes place.

Most of our algorithms using pseudocode resembles C and Pascal.

1. **Comments** begins with // and continue until the end of line

2. **Blocks** are indicated with matching braces { and }

   Collection of simple statements can be represented as block.

3. **Identifiers** are names for entities and they begin with a letter. The data type of the identifiers is not explicitly declared in algorithm. Simple data type such as integer, float, char, Boolean and so on.

4. **Assignment Statement** are used to assign values to variables.

5. **Boolean values-** There are two Boolean values true or false. In order to produce these values, the logical operators **and**, **or** and **not** and the relational operators **$<, \leq, >, \geq, =$** and **$\neq$** are provided.

6. **Multidimensional arrays-**Elements of multidimensional arrays are accessed using [and]. For example, *A* is a two dimensional array, the [i,j]th element of the array is denoted as *A*[i,j]. Array indices start at 0.

7. **Loop Statements-**Looping statement such as **for**, **while** and **repeat until** are employed in algorithm to do looping executions.

```
While <condition> do
    {
        <statement-1>
        <statement-2>
              .
              .
              .
        <statement-n>
    }
```

The statements are executed as long as <condition> is true. When <condition> becomes false, the loop is exited. The value of <condition> is evaluated at the top of the loop.

The general form of **for** loop is

```
for variable := value1 to value2 step step do
    {
        <statement-1>
        <statement-2>
              .
```

```
                  .
                  .
              <statement-n>
          }
```

Here *value1*, *value2* and *step* are arithmetic expressions. The clause "**step** *step"* is optional and is taken as +1 if it does not occur.
The statements are executed as *value1* reaches the value of *value2.*

The general form of **repeat-until** is

```
repeat
      <statement-1>
      <statement-2>
          .
          .
          .
      <statement-n>
until <condition>
```

The statements are executed as long as <condition> is false.

The value of <condition> is validated after executing the statements.

The instruction **break;** can be used within any of the looping instructions to force exit.

A **return** statement within any of the above also will result in existing the loops.

8. **Conditional statements-**A conditional statements has the following form:

   if *<condition>* **then** *<statement>*

if *<condition>* **then** *<statement 1>* **else** *<statement 2>*
Here *<condition>* is a Boolean expression

The general form of **case** statement is

```
case
    {
        :<condition-1>:<statement-1>
        :<condition-2>:<statement-2>
                .
                .
                .
        :<condition-n>:<statement-n>
    }
```

9. **Read and Write :** Input and Output are done using the instructions **read** and **write.** No format is used to specify the size of input or output quantities.

10. **Algorithm:** There is only one type of procedure called Algorithm, consists of heading and a body. The general form of algorithm

Algorithm Name (<parameter list>)

where "Name" is the name of the procedure and (<parameter list>) is a listing of procedure parameters. The body is one or more simple statements enclosed within braces { and }. An algorithm may or may not return any values. Simple variables to procedures are passed by value. Arrays and records are passed by reference.

Recursive algorithm

In computer science, all algorithms are implemented with programming language functions.

We can view function as something that is called by another function. It executes its code and then returns control to the calling function.

Recursive functions can call themselves or it may call another function which again calls same function inside it.

The function which calls by itself is called a direct recursive function.

The function which calls a function and that function calls its called function is called indirect recursive function.

<u>**Performance analysis**</u>

Performance of an algorithm is a process of evaluating the algorithm.

In other words, performance of algorithm means predicting the resources which are required to an algorithm to perform its task.

Generally the performance of an algorithm depends on

1. Whether your algorithm provides exact solution to your problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space is required to solve the problem?
5. How much time it takes to solve the problem?

Majorly performance analysis of an algorithm is performed by using the following measures.

- Space complexity
- Time complexity

Space complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.

For any algorithm, memory is required for the following purposes...

1. Memory required to store program instructions
2. Memory required to store constant values
3. Memory required to store variable values
4. And for few other things

Generally, when a program is under execution, it used the computer memory for three reasons.

- Instruction space :
    - It is the amount of space used to store compiled version of program or instructions.

- Environmental stack :

    - It is the amount of space used to store information of partially executed functions at the time of function call.

- Data space
    - It is the amount of memory used to store all the variables and constants.

When we want to perform analysis of an algorithm based on its space complexity, we consider Data space only. That means we calculation only memory required to store variables, constant, structures, etc.

To calculate the space complexity, we must know the memory required to store different data type values. For example, the C programming language compiler requires,

1. 2 bytes to store integer value
2. 4 bytes to store floating point value
3. 1 byte to character value
4. 6 or 8 bytes to store double value

Example 1:

```
int sqare(int a)
{
    Return a * a ;
}
```

In the above piece of code, 2 bytes of memory required for integer variable ‘a’, and another 2 bytes of memory required to return result.

Totally 4 bytes of memory required to execute the above example.

And this 4 bytes of memory requirement is constant for any value of integer variable ‘a’.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be **constant space complexity**.

Example 2:

```
int sum(int a[], int n)
{
    int sum=0,i;
    for (i=0;i=n;i++)
        sum =sum+a[i];
    return sum;
}
```

In the above piece of code, it requires,

n*2 bytes of memory for integer array variable ‘a[]’
2 bytes for variable ‘n’
4 bytes for local variables “sum” and “i” (2 bytes for each)
2 bytes for return value

Totally 2n+8bytes required. Here the amount for memory depends on the input value of ‘n’.

If the amount of memory required by an algorithm is increased with the increase of input value then that space complexity is said to be **linear space complexity**.

Time Complexity

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, running time of an algorithm depends upon,

1. Whether it is running on Single processor machine or Multi processor machine.
2. Whether it is a 32 bit machine or 64 bit machine
3. Read and Write speed of the machine.
4. The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
5. Input data

When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.

We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc., To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It performs sequential execution.
2. It requires 1 unit of time for Arithmetic and Logical operations.
3. It requires 1 unit of time for Assignment and Return value.
4. It requires 1 unit of time for Read and Write operations.

Example 1:

```
    int sum(int a, int b)
    {
        return a+b;
    }
```

In above sample code, it requires 1 unit of time to calculate a+b and 1 unit of time to return the value.

That means, totally it takes 2 units of time to complete its execution.

And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be **Constant Time Complexity**.

Example 2 :

```
    int  sum(int a[], int n)
    {
        int sum = 0,i;
      for (i = 0, i < n, i++)
        sum =sum + a[i];
        ruturn sum
    }
```

Time/Operation	Repeatation	Total
1	1	1
1+1+1	1+(n+1)+n	2n+2
1+1	(1+1)2	2n
1	1	1
		4n+4

In above calculation Time/Operation

Cost is the amount of computer time required for a single operation in each line.

Repeatation is the amount of computer time required by each operation for all its repeatations.

Total is the amount of computer time required by each operation to execute.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be **Linear Time Complexity.**

PERFORMANCE ANALYSIS

Asymptotic Notations

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Generally, there are 3 types of asymptotic notations, they are

1. Big Oh (O)
2. Big Omega (Ω)
3. Big Theta (θ)

1. Big Oh(O)

Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
Big Oh notation can be defined as follows,
Consider function f(n) the time complexity of an algorithm and g(n) is the most signification term. If $f(n) <= C\, g(n)$ for all $n >= n_o$ and $C > 0$ and $n_o > 0$, then we represent,

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

Example:
Consider the following f(n) and g(n)...
$f(n) = 3n + 2$; $g(n) = n$

If we want to represent f(n) as O(g(n)) then it must satisfy f(n) <= C x g(n) for all values of C > 0 and n_0>= 1

f(n) <= C g(n)

3n + 2 <= C n

Above condition is always TRUE for all values of C = 4 and n >= 2.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

2. Big Omega(Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows.

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C x g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Example:

Consider the following f(n) and g(n)...

f(n) = 3n + 2 ; g(n) = n

If we want to represent f(n) as O(g(n)) then it must satisfy f(n) <= C x g(n) for all values of C > 0 and n_0>= 1

f(n) <= C g(n)

3n + 2 <= C n

Above condition is always TRUE for all values of C = 1 and n >= 1.

By using Big - Omega notation we can represent the time complexity as follows...

3n + 2 = Ω(n)

3. Big Theta(θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

That means Big - Theta notation always indicates the average time required by an algorithm for all input values.

That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If C1*g(n) <= f(n) >= C2*g(n) for all n >= n_0, C1, C2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Example:
Consider the following f(n) and g(n)...
f(n) = 3n + 2 ; g(n) = n
if we want to represent f(n) as Θ(g(n)) then it must satisfy C1 g(n) <= f(n) >= C2 g(n) for all values ofC1, C2 > 0 and n0>= 1
C1 g(n) <= f(n) >= C2 g(n)
C1 n <= 3n + 2 >= C2
Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 1.
By using Big - Theta notation we can represent the time complexity as follows...

3n + 2 = Θ(n)

Divide and conquer method:

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the sub problems into even smaller sub-problems, we may eventually reach a stage where no more division is possible.
The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.
Divide and conquer approach is a three step approach,

1. **Divide/Break**

   - This step involves breaking the problem into smaller sub-problems.
   - Sub-problems should represent a part of the original problem.
   - This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible.
   - At this stage, sub-problems become atomic(tiny) in nature but still represent some part of the actual problem.

2. **Conquer/Solve**

   - This step receives a lot of smaller sub-problems to be solved.
   - Generally, at this level, the problems are considered 'solved' on their own.

3. **Merge/Combine**

- When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem.
- This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.
  Some of the Divide and Conquer approach algorithms are,

    1. Binary Search
    2. Finding maximum and minimum
    3. Merge Sort
    4. Quick Sort

What is search? Search is an operation or technique that helps to find a place of given element or value in the list.

Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not.

Some of the standard searching techniques are:

- Linear or Sequential Search
- Binary Search

Linear Search:

This is the simplest method for searching.

In this technique of searching, the element to be found in searching the elements to be found is searched sequentially in the list.

This method can be performed on a sorted or an unsorted list (usually arrays).

In case of a sorted list searching starts from 0th element and continues until the element is found from the list or the element whose value is greater than (assuming the list is sorted in ascending order), the value being searched is reached.

As against this, searching in case of unsorted list also begins from the 0thelement and continues until the element or the end of the list is reached.

1. **Binary Search/Half Interval Search/ Algorithm**

   Binary search is a very fast and efficient searching technique. It requires the list to be in sorted order

   Binary search is a search algorithm that finds the position of a target value within a sorted array

Binary Search Algorithm compares the input element x with the value of the middle element in array. If the values match, return the index of middle.

Otherwise, if x is less than the middle element, then the algorithm recurs (happen against) for left side of middle element, else recurs for right side of middle element.

For example:

Input:

List

0	1	2	3	4	5	6	7	8	9
10	12	20	32	50	55	65	80	99	110

Case 1: Searching Element: 12

Step 1 :

Search Element (12) is compared with middle element (50)

0	1	2	3
10	12	20	32

Both are not matching. And 12 is smaller than 50. So we search only in the left side sub list (i.e 10,12,20,32)

Step 2 :

Search Element (12) is compared with middle element (12)
Both are matching. So the resulting index is 1

Case 2 : Searching Element: 99

Step 1 :

Search Element (80) is compared with middle element (50)
Both are not matching. And 80 is greater than 50. So we search only in the right side sub list (i.e 55,65,80,99,110)

5	6	7	8	9
55	65	80	99	110

Step 2 :

Search Element (99) is compared with middle element (80)
Both are not matching. And 99 is greater than 80. So we search again the right side sub list (i.e 99,110)

8	9
99	110

Step 3 :

Search Element (99) is compared with middle element (99)
Both are matching. So the resulting index is 8.

Binary search algorithm is implemented using following steps...

Step :1	Read the search element from the user
Step :2	Find the middle element in the sorted list
Step :3	Compare, the search element with the middle element in the sorted list.
Step :4	If both are matching, then display "Given element found!!!" and terminate the function
Step :5	If both are not matching, then check whether the search element is smaller or larger than middle element.
Step :6	If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
Step :7	if the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
Step :8	Repeat the same process until we find the search element in the list or until sublist contains only one element.
Step :9	If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

```
Algorithm  BinSrc(a,i , l, x)
//Given an array a[i, l] of elements  in non decreasing order and 1 <= i <= l
//x is the searching element
{
   if i = l
      {
         If a[i] = x then return i;
            {
            else return 0; (Searching element not in a given array)
            }
    Else
         {  // reduce problem in to sub problem.
             mid :=[ (i+l)/2];
               If a[mid] = x then return mid;
               else
                     if x < a[mid] then BinSrc(a,I,mid-1,x);
                     else BinSrc(a,mid+1,l,x);
          }
```

Finding Maximum and Minimum :

Another problem that can be solved simply by divide and conquer approach is finding maximum and minimum items in a set of n elements.

Naïve Method:

```
Algorithm MaxMinElement(number[])
Max := number[1];
Min := number[1];
for (I = 2 to n) do
  if number[i] > max then max := number[i];
  if number[i] < min the min    := number[i];
return Max,Min
```

The number of comparison in this naïve(inexperienced) method is 2n-2

Divide and Conquer Method:

In this approach, the array is divided into two halves.

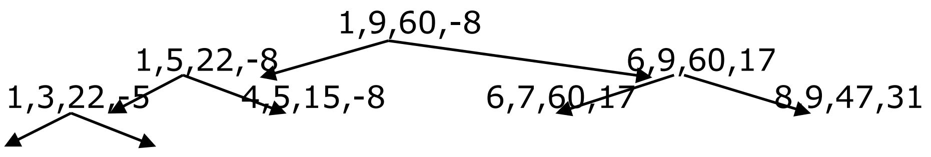
Then using recursive approach maximum and minimum numbers in each halves are found.

Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

1	2	3	4	5	6	7	8	9
22	13	-5	-8	15	60	17	31	47

Recursively finding maximum and minimum

1	Algorithm MaxMin(i,j,max,min) // I, j are integers, 1 <=i <= j <= n a[i,j]
2	{
3	If i =j then max := min := a[i] ; // small problem
4	else If i = j -1 then
5	{
6	If a[i] < a[j] then max := a[j]; min =a[i];
7	else max := a[i]; min = a[j];
8	}
9	Else
10	{ //problem P is not small divide problem P into subproblems
11	Mid := [(i+j)/2];
12	MaxMin(i,mid,max,min);
13	MaxMin(mid+1,j,max1,min1);
14	If max < max1 then max := max1
15	If min > min1 then min := min1
16	}
17	}



1,2,22,13 3,3,-5,-5

Tree of recursive calls of MaxMin

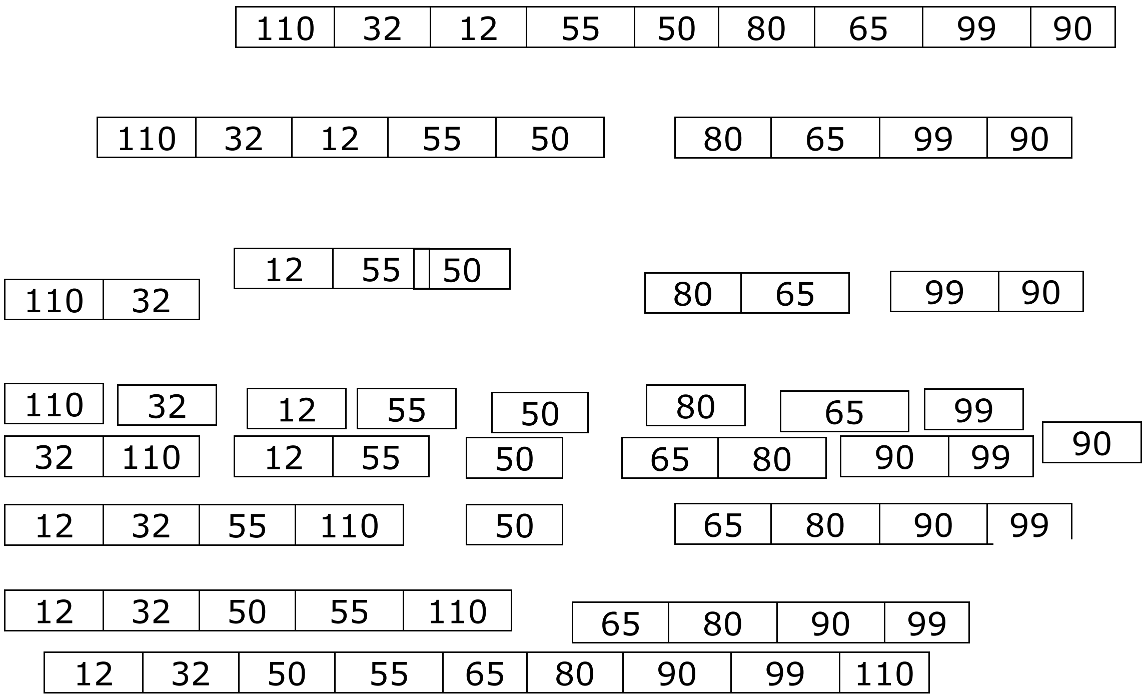
Merge Sort Algorithm

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

The algorithm divides the array in two halves; recursively sorts them and finally merges the two sorted halves.

Merge sort keeps on dividing the array into equal halves until it can no more be divided. By definition, if it is only one element in the array, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

For example :



Step :1	Check if it is only one element in the array it is already sorte
Step :2	Divide the array or list recursively into two halves until it ca
Step :3	Merge the smaller lists into new list in sorted order

```
Algorithm  MergeSort(low, high)
{
   If  low < high
     {
        mid := [(low+high)/2]
        MergeSort(low,mid);
        MergeSort(mid+1,high);
        Merge(low,mid,high);
     }
}
```

```
Algorithm  Merge(low,mid,high)
{
h:=low; I :=low; j := mid+1;
while a[h] < a[j] and j <= high
{
    If a[h] <= a[j]
       {
           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 + 1;
     }
For k := low to high do  a[k] := b[k];
}
```

Quick Sort :

Quick sort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order.

Quick sort is a sorting algorithm. The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the sub arrays on left and right of pivot element.

There are many different versions of quick Sort that pick pivot in different ways.

- Always pick first element as pivot.
- Always pick last element as pivot (implemented below)
- Pick a random element as pivot.
- Pick median as pivot.

1	Algorithm QuickSort(a[],low,high)
2	//Sorts the elements a[low]….a[high] which in the global array a[1:n]
3	{
4	If low < high
	//divide P into two subproblems
4	{
5	//Pivot is the partitioning index, arr[pi] is now at the right place//
5	pi := Partition(a[],low,high)
6	QuickSort(a[],p,pi-1); //before pivot
7	QuickSort(a[],pi+1,q); // after pivot
8	}
9	}

1	Algorithm Partition (a[],low,high)
2	{
3	Pivot = a[high];
4	i = low-1 //index on smaller element
5	For (j=low; j <= high-1; j++)
6	{
7	If a[j] <= pivot then
8	{
9	i++ // increment index of smaller element
10	Swap a[i] and a[j]
11	}
12	}
13	Swap a[i+1] and a[high]
14	Return (i+1)

a[]	0	1	2	3	4	5	6
value	10	80	30	90	40	50	70

Low = 0; high = 6; pivot = arr[high]=70

Initialize index of smaller element, i = low-1 = 0 – 1 = -1

I = -1, j = 0, pivot = 70

Loop	J	Condition	Action	I	Resulted array
1	0	a[j] < pivot	Do I++ Swap(a[i],a[j]	0	arr[]= {10,80,30,90,40,50,70}
2	1	a[j] > pivot	Do nothing	0	arr[]= {10,80,30,90,40,50,70}
3	2	a[j] < pivot	Do i++ Swap(a[i],a[j]	1	arr[]= {10,30,80,90,40,50,70}
4	3	a[j]>pivot	Do nothing	1	arr[]= {10,30,80,90,40,50,70}
5	4	a[J]<pivot	Do I++	2	arr[]= {10,30,40,90,80,50,70}

			Swap(a[i],a[j]		
6	5	a[j]<pivot	Do I++ Swap(a[i],a[j]	3	arr[]= {10,30,40,50,80,90,70}
7	Swap(a[i+1),a[high])				arr[]= {10,30,40,50,70,90,80}
Now the j value is less than high-1 that j = 5 and high-1 is also 5,loop ends					

Now pivot is positioned in its place

Now take the last element a[6] = 80 as our new pivot do the partition again.

Unit - II

Greedy Methods: Knapsack Problem, Minimum Cost Spanning Trees, Optimal Storage on Tapes and Single Source Shortest Path Problem.

Greedy Methods

In an algorithm design there is no single algorithm helps to solve all computation problems.

Different problems require the use of different kinds of techniques.

A good programmer uses all these techniques based on the type of problem.

Some commonly-used techniques are:

- Divide and conquer
- Randomized algorithms
- Greedy algorithms
- Dynamic programming

What is a 'Greedy algorithm'?

A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**.

This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

A greedy algorithm is a simple, intuitive (sensitive) algorithm that is used in optimization problems. Greedy algorithms are quite successful in some problems,

The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.

Knapsack Problem

The knapsack problem or rucksack(backpack) problem is a problem in combinatorial optimization: Given a set of items, each with a weight and 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 and the total value is as large as possible.

The objective of Knapsack problem is to fill the knapsack with items to get maximum benefit (value or profit) without crossing the weight capacity of the knapsack.

Example :

Assume that we have a knapsack with max weight capacity, W = 16.

Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Consider the following items and their associated weight and value:

Item	Weight	Value
I-1	6	6
I-2	10	2
I-3	3	1
I-4	5	8
I-5	1	3
I-6	3	5

Steps :

1. Calculate value per weight for each item (we can call this value density)
2. Sort the items as per the value density in descending order
3. Take as much item as possible not already taken in the knapsack

   1. Density = value/weight

Item	Weight	Value	Density Value/weight
I-1	6	6	1.000
I-2	10	2	0.200
I-3	3	1	0.333
I-4	5	8	1.600
I-5	1	3	3.000
I-6	3	5	1.667

2. Sort the items as per density in descending order

Item	Weight	Value	Density Value/weight
I-5	1	3	3.000
I-6	3	5	1.667
I-4	5	8	1.600
I-1	6	6	1.000
I-3	3	1	0.333
I-2	10	2	0.200

Now we will pick items such that our benefit is maximum and total weight of the selected items is at most W.

Objective is to fill the knapsack with items to get maximum benefit without crossing the weight limit W = 16.

Maximum benefit without crossing the weight limit

How to fill the knapsack table?

```
Is Weight(i)+ Total Weight <= w, if its yes, then take whole item
else
   (w-total weight)/$w_i$= 16-15=1/3=0.333
endif
```

Item	Weight	Value	Total Weight	Benefit
I5	1	3	1	3
I6	3	5	4	8
I4	5	8	9	16
I1	6	6	15	22
I3	1	0.333	16	22.333

So, total weight in the knapsack = 16 and total value inside it = 22.333336

Greedy algorithms don’t always yield optimal solutions but, when they do, they’re usually the simplest and most efficient algorithms available.

Algorithm :

Algorithm Knapsock	
Step : 1	For each item I_j,compute the ratio v_j/w_j (i.e., value per unit weight), $1 \leq j \leq n$.
Step : 2	Sort the items in decreasing order of their ratios.
Step : 3	Set TotalWeight = 0, TotalBenefit = 0 , W=WeightLimit(16)
	For (i=0 to n)
	If item[i].weight+TotalWeight <= W
	TotalWeight=TotalWeight+item[i].weight
	TotalBenefit = TotalBenefit +item[i].benefit
	Else
	Wt = W-TotalWeight;
	Value = wt * (item[i].benefit/item[i].weight)
	TotalWeight = TotalWeight + Wt
	TotalBenefit = TotalBenefit + value
	Break; (exit loop)
	Endif
Step : 4	Output the contents of Knapsack

Weight Limit = 16 ; TotalWeight = 0, TotalBenefit = 0

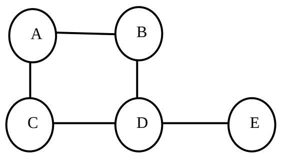
i	Checking Condition	TotalWeight	Total Benefit
0	If (0+3 <= 16	1	3
1	If (1+3 <= 16	1+3 = 4	3+5 =8
2	If(4+5 <= 16	4+5=9	8+8 = 16
3	If(9+6 <=16	9+6=15	16+6 = 22
4	If(15+1 > 16)	Wt = 16-15 = 1 15+1 = 16	Value = 1/3=0.333 22+0.333=22.333

Minimum Cost Spanning Trees- Network Design

What is graph?

A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



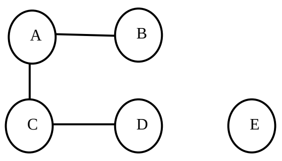


Fig.(A) Connected Graph   Fig.(B) Disconnected Graph

In the above graph,

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

E = {Ab, Ac, Bd, Cd, De}

Connected Graph: A graph is connected when there is a path between every pair of vertices. In a connected graph, there are no unreachable vertices.

Disconnected Graph: A graph G is said to be disconnected if there exist two nodes in G such that no path in G has those nodes as endpoints.

What is Tree?

A tree is an undirected graph in which any two vertices are connected by exactly one path.

What is a Spanning Tree?

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

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 sub graph of G (every edge in the tree belongs to G).

A Spanning Tree T of an undirected graph G is a sub graph that is a tree which includes all of the vertices of G, with minimum possible number of edges.

Definition:

Let G = (V,E) be an undirected connected graph. A sub graph t = (V,E') of G is spanning tree iff t is a tree.

For example : Fig.1 is a undirected connect graph, and Fig.1(a),1(b) and 1(c) are three spanning trees of graph Fig.1(a)

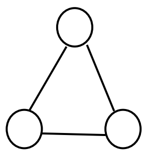


Fig.1

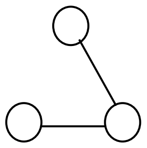


Fig.1(a)

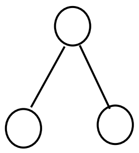


Fig.1(b)

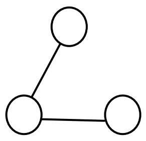


Fig.1(c)

We found three spanning trees off one complete graph. A complete undirected graph can have maximum n^{n-2} number of spanning trees, where n is the number of nodes.

In the above addressed example, $3^{3-2} = 3$ spanning trees are possible.

Properties of Spanning Tree

- A connected graph G can have more than one spanning tree.
- All possible spanning trees of graph G, have the same number of edges and vertices.

- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
- Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic (acyclic means without cycles).
- Spanning tree has n-1 edges, where n is the number of nodes (vertices).
- From a complete graph, by removing maximum e - n + 1 edges, we can construct a spanning tree.
- A complete graph can have maximum n^{n-2} number of spanning trees.

What is a Minimum Cost Spanning Tree?

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

The standard application is to a problem like phone network design.

You have a business with several offices;

you want to lease phone lines to connect them up with each other; and

The phone company charges different amounts of money to connect different pairs of cities.

You want a set of lines that connects all your offices with a minimum total cost.

It should be a spanning (across) tree, since if a network isn’t a tree you can always remove some edges and save money.

General applications of MST are :

- Telephone
- Electrical
- Hydraulic

- TV cable
- Computer
- Road

Kruskal's Algorithm:

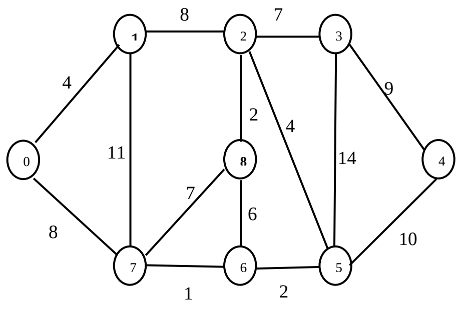
Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

The steps for implementing Kruskal's algorithm are as follows:

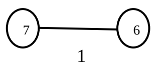
Algorithm Kruskalsmst	
Step:1	Sort all the edges in non-decreasing order of their weight.
Step :2	Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
Step :3	Keep adding edges until we reach all vertices

.

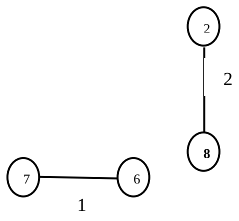


Weight	Edges
1	6-7
2	2-8
2	5-6
4	0-1
4	2-5
6	6-8
7	2-3
7	7-8
8	0-7
8	1-2
9	3-4
10	4-5
11	1-7
14	3-5

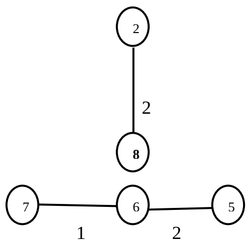
1. Pick edge 6-7



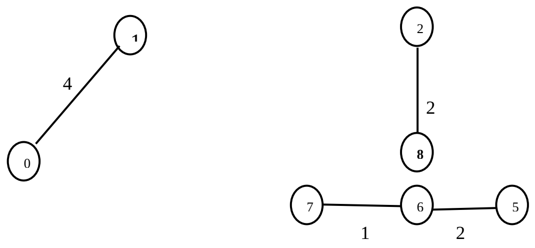
2. Pick edge 2-8



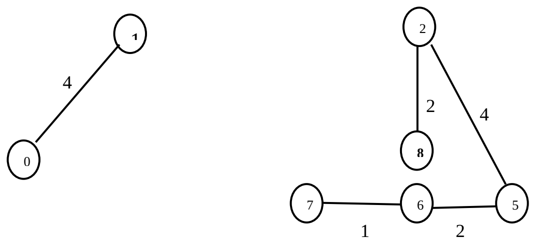
3. Pick edge 6-5: No cycle is formed, include it.



4. Pick edge 0-1: No cycle is formed, include it.

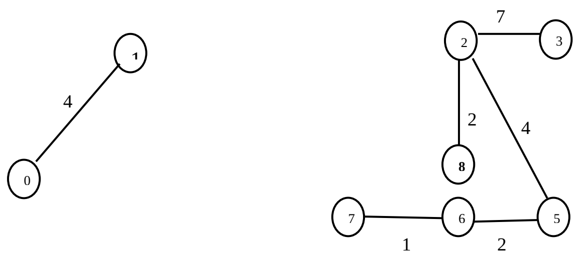


5. Pick edge 2-5: No cycle is formed, include it.

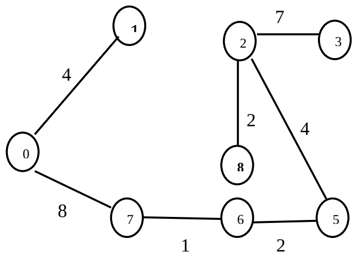


6. Inclusion of edge 6-8 results cycle, So reject it

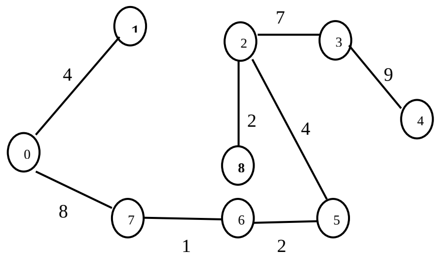
7. Pick edge 2-3: No cycle is formed, include it.



8. Inclusion of edge 7-8 results cycle, So reject it
9. Pick edge 2-3: No cycle is formed, include it.



10. Inclusion of edge 1-2 results cycle, So reject it
11. Pick edge 3-4: No cycle is formed, include it.



Total Weight is :
4+8+1+2+4+2+7+9=**37**

Since the number of edges included equals (V – 1), the algorithm stops here.

1	t= θ;
2.	While (t <less than n-1 edges) and E ≠ θ do
3.	{
4.	Choose an Edge (v,w) from E of lowest cost;

5.	If (v,w) does not create cycle in t then
	Add (v,w) to t;
	else
	discard (v,w);

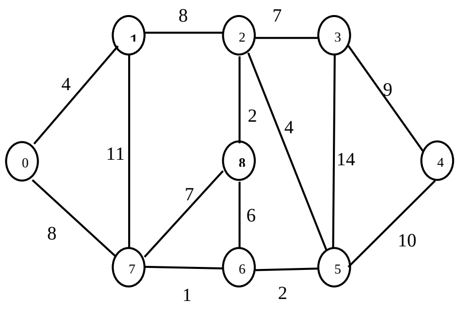
6.	}

Prim’s Algorithm

- Prim’s algorithm is also a Greedy algorithm.
- It starts with an empty spanning tree.
- The idea is to maintain two sets of vertices.
- The first set contains the vertices already included in the MST, the other set contains the vertices not yet included.
- At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges.
- After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Algorithm:

Algorithm Primsmst	
Step 1:	Create a set *mstSet* that keeps track of vertices already included in MST.
Step 2:	Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
Step 3:	While mstSet doesn’t include all vertices
	Pick a vertex *u* which is not there in *mstSet* and has minimum key value. Include *u* to mstSet. Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u*-*v* is less than the previous key value of *v*, update the key value as weight of *u*-*v*



1. The set *mstSet* is initially empty and keys assigned to vertices are

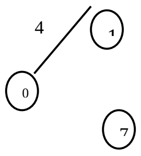
   mstSet = {0, INF, INF, INF, INF, INF, INF, INF}

2. *Now pick the vertex with minimum key value. The vertex 0 is picked, include it in mstSet.*

   mstSet = {0}

3. *Adjacent vertices of 0 are 1 and 7 and its corresponding key values are 4 and 8. So the lowest key value 4 of vertex 1 is included in mstSet.*

   mstSet = {0,1}

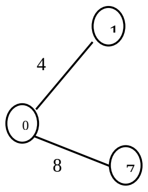


4. Now the open vertices are 0 and 1.
   - Adjacent veritex of 0 is 7 with key value 8

   - Adjacent verticies of 1 are 2 and 7 with key values 8, 11 respectively.

   The lowest key value here is 8. So randomly we select vertex 7 here and vertex 7 is included in mstSet

   mstSet = {0,1,7}

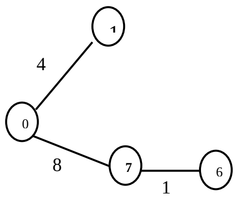


5. Now the open vertices are 1 and 7.

   - Adjacent verticies of 1 is 2 with key values 8. The vertex 7 is considered here because it is already in the mstSet.
   - Adjance vertices of 7 are 6 and 8 with key values 1 and 7 respectivly.

   So The lowest key value vertex 6 is included in mstSet.

   mstSet = {0,1,7,6}

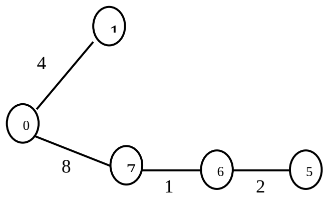


6. Now the open vertices are 1 and 6.

   - Adjacent verticies of 1 is 2 with key values 8.
   - Adjance vertices of 6 are 5 and 8 with key values 2 and 6 respectivly.

   So The lowest key value(2) vertex 5 is included in mstSet.

   mstSet = {0,1,7,6,5}

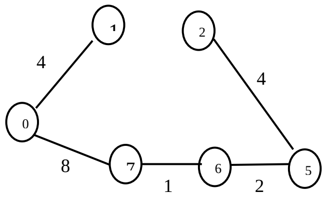


7. Now the open vertices are 1 and 5.

   - Adjacent verticies of 1 is 2 with key values 8.
   - Adjacent verticies of 5 are 2,3,4 with key values 4,14,10 respectively.

   The lowest key value (4) of vertex 2 in included in mstSet.

mstSet = {0,1,7,6,5,2}

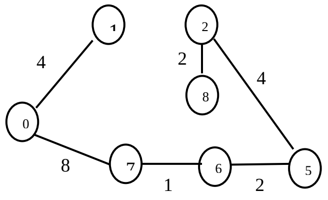


8. Now the open vertices are 2 and 5.

- Adjacent verticies of 2 are 3 and 8 with key value 7 and 2 respectively.
- Adjacent verticies of 5 are 3,4 with key values 14,10 respectively.

The lowest key value (2) of vertex 8 is included in mstSet.

mstSet = {0,1,7,6,5,2,8}

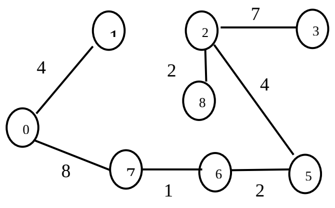


9. Now the open vertices are 2 and 5.

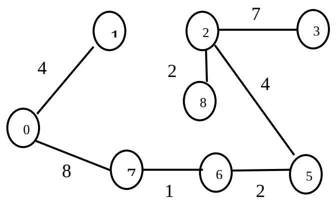
- Available Adjacent vertex of 2 is 3 with key value 7.
- Available Adjacent verticies of 5 are 3,4 with key values 14,10 respectively.

The lowest key value (7) of vertex 3 is included in mstSet.

mstSet = {0,1,7,6,5,2,8,3}



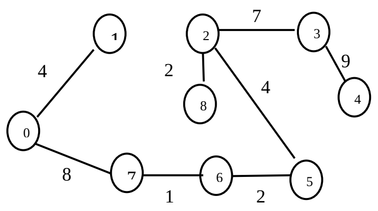
10. Now the open vertices are 3 and 5



- Available Adjacent vertex of 3 is 4 with key value 9.
- Available Adjacent vertex of 5 is also 4 with key value 10.

So the lowest key value (9) of vertex 4 is included in mstSet.

mstSet = {0,1,7,6,5,2,8,3,4}



The Total weight is : 4+8+1+2+4+2+7+9 = 37

Optimal Storage on Tapes

Given ‘n’ programs to be stored on tape, the lengths of these programs are i1, i2….in respectively. Suppose the programs are stored in the order of i1, i2…in. We have a tape of length L i.e. the storage capacity of the tape is L. We are also given ‘n’ programs where length of each program is i is Li.

Let ‘Tj’ be the time to retrieve program ‘ij’.

It is now required to store these programs on the tape in such a way so that the mean retrieval time is minimum. MRT is the average tome required to retrieve any program stored on this tape.

Assume that the tape is initially positioned at the beginning.

‘Tj’ is proportional to the sum of all lengths of programs stored in front of the program ‘ij’.

The goal is to minimize MRT (Mean Retrieval Time).

MRT = $1/n \sum_{i=0}^{n} T_i$

Example 1:

1. Number of programs(n) = 3, Length of programs (l1, l2, l3) = (5, 10, 3).
2. We can store these 3 programs on the tape in any order but we want that order which will minimize the MRT.
3. Suppose we store the programs in order (L1, L2, L3).
4. Then MRT is given as (5+(5+10)+(5+10+3))/3=38/3.

Consider the following Table

Ordering	Mean Retrieval Time (MRT)
L1,L2,L3	5+(5+10)+(5+10+3)=38/3
L1,L3,L2	5+(5+3)+(5+3+10) = 31/3
L2,L1,L3	10+(10+5)+(10+5+3) = 43/3
L2,L3,L1	10+(10+3)+(10+3+5) = 41/3
L3,L1,L2	3+(3+5)+(3+5+10) = 29/3
L3,L2,L1	3+(3+10)+(3+10+5) = 34/3

It should be seen that the minimum MRT of 29/3 is obtained in case of (L3,L1,L2). Hence the optimal solution is achieved if the programs are stored in increasing order of their length.

Algorithm MRT	
Step 1:	Sum = 0;
Step 2:	For (i=1 ;i <= n,i++)
Step 3:	{
Step 4:	For (j=1;j<=I,j++)
Step 5:	{
Step 6:	Sum = sum + Lj;
Step 7:	}
Step 8:	MRT = sum/n;
Step 9:	}

The time complexity of this algorithm including the time to do sorting is O(n^2).

After getting the minimum MRT, the programs are stored in tape.

Solution: Sorted order is 3,5,10 (L3,L1,L2)

3	5	10

Algorithm for more than one tape :

Algorithm Store(n,m)	
// n is the number of programs, m is the number of tapes	
Step 1:	{
Step 2 :	Sort the programs in ascending order using any of the sorting algorithm
Step 3 :	J = 1; //Next tape to store on
Step 4 :	For I = 1 to n do;
Step 5 :	{
Step 6 :	Write ("append program", i ,"to permutation of tape",j);
Step 7 :	j = (j+1) mod m
Step 8 :	}
Step 9 :	}

Example:

Let us assume, we want to store files of lengths (in MB)
{12, 34, 56, 24, 11, 34, 34, 45}

Sorted files are

{11, 12, 24, 34, 34, 34, 45, 56}

Now distribute the files:

Inserting 11

Tape 1	11		
Tape 2			
Tape 3			

Inserting 12

Tape 1	11		
Tape 2	12		
Tape 3			

Inserting 24

Tape 1	11		
Tape 2	12		
Tape 3	24		

Inserting 34

Tape 1	11	34	
Tape 2	12		
Tape 3	24		

Inserting 34

Tape 1	11	34	
Tape 2	12	34	
Tape 3	24		

Inserting 34

Tape 1	11	34	
Tape 2	12	34	
Tape 3	24	34	

Inserting 45

Tape 1	11	34	45
Tape 2	12	34	
Tape 3	24	34	

Inserting 56

Tape 1	11	34	45
Tape 2	12	34	56
Tape 3	24	34	

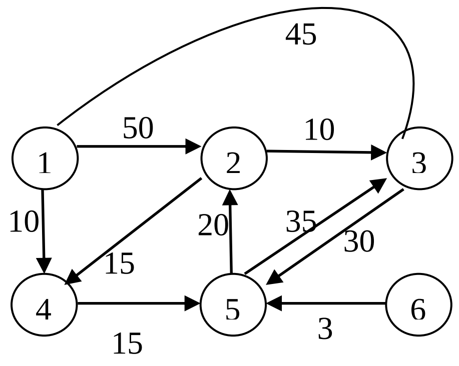
The time complexity of this algorithm is o(n log n).

Single Source Shortest Path Problem

The **shortest path problem** is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.

The single-source shortest path problem, in which we have to find shortest paths from a source vertex v to all other vertices in the graph.

Algorithm ShortestePath(v,cost,dist,n)	
v- source vertex, dist[j] –shortest path between v to j, cost – overall crossed paths, n – number of vertices	
Step 1:	{
Step 2 :	For I = I to n do
Step 3 :	S[i] = false; dist[i] := cost[v,i]
Step 4 :	}
Step 5 :	{
Step 6 :	S[v] = True ; S[v] = 0;
Step 7 :	{
Step 8 :	for j := 2 to n-1 do
Step 9 :	{
Step 10:	S[j] = True;
Step 11:	for (each k adjacent to j and s[k] = false) do
Step 12:	{
Step 13:	If dist[k]> dist[j] + cost[j,k] then
Step 14:	dist[k] := dist[j] + cost[j,k]
Step 15:	}
Step 16:	}



	Path	Dist
1	1,4	10
2	1,4,5	25
3	1,4,5,2	45
4	1,3	45

Unit - III

Dynamic Programming: Multistage Graphs, 0/1 knapsack and Traveling Salesman Problem. Basic Traversal and Search Techniques: Techniques for Binary Tree, Techniques for Graphs: Depth First Search and Breadth First Search - Connected Components and Spanning Tree – Bi-connected Components and DFS.

Dynamic Programming

Dynamic Programming (also known as Dynamic Optimization) is a method for solving a complex problem by breaking it down into a collection of simpler sub problems, solving each of those sub problems just once, and storing their solutions.

The next time the same sub problem occurs, instead of re-computing its solution, one simply looks up the previously computed solution, thereby saving computation time as well as storage space.

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

Dynamic programming algorithms are often used for optimization.

A dynamic programming algorithm will examine the previously solved sub-problems and will combine their solutions to give the best solution for the given problem.

Multistage Graphs

A multistage graph G=(V,E) which is a directed graph. In this graph all the vertices are partitioned into the k stages where k>=2. Usually multistage graphs are weighted graphs.

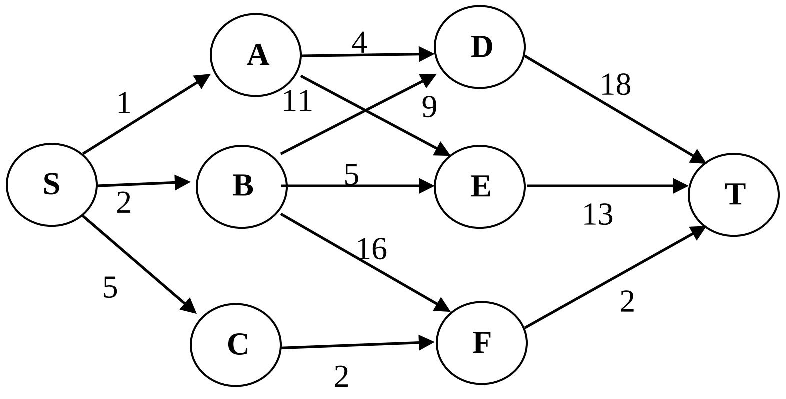
In multistage graphs, starting vertex is called as source vertex and the ending vertex is called as sink(go under).

In multistage graph problem we have to find the shortest path from source to sink.

For Example :

The Shortest path in Multistage Graphs

Dynamic Programming approach – Forward Approach



d(S,T)= min{1+d(A,T), 2+d(B,T), 5+d(C,T)}

d(A,T) = min{4+d(D,T), 11+d(E,T))}
= min{4+18, 11+13}
= min{22,24}
= 22

d(B,T) = min{ 9+d(D,T), 5+d(E,T), 16+d(F,T)
= min{ 9+18, 5+13, 16+2}
= min{ 27, 18, 18}
= 18

d(C,T) = min{2+d(F,T)}
= min{2+2}
= 4

d(S,T) = min{1+23, 2+18, 5+4}
= min{24, 20, 9}
= 9

$d(S,T) = S \rightarrow C \rightarrow F \rightarrow T$

1. Identify source and destination nodes.

2. Find all possible paths to reach destination from source and sum of weights of adjacent nodes.

3. The path giving the least weight will be the minimum spanning path.

```
Algorithm FGraph($G, k, n, p$)
// The input is a $k$-stage graph $G = (V, E)$ with $n$ vertices
// indexed in order of stages. $E$ is a set of edges and $c[i, j]$
// is the cost of $\langle i, j \rangle$. $p[1 : k]$ is a minimum-cost path.
{
    $cost[n]$ := 0.0;
    for $j := n - 1$ to 1 step $-1$ do
    { // Compute $cost[j]$.
        Let $r$ be a vertex such that $\langle j, r \rangle$ is an edge
        of $G$ and $c[j, r] + cost[r]$ is minimum;
        $cost[j] := c[j, r] + cost[r]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := 2$ to $k - 1$ do $p[j] := d[p[j - 1]]$;
}
```

Dynamic Programming approach – Backward Approach

Backward Approach is just the reverse of forward approach, here Source node and the next node is considered at every stage.

d(S,A) = 1
d(S,B) = 2
d(S,C) = 5

d(S,D) = min{dist(S,A) + dist(A,D), dist(S,B) + dist(B,D)}
= min{1+4= 5, 2+9=11} = 5

d(S,E) = min{dist(S,A) + dist(A,E), dist(S,B) + dist(B,E)
= min{1+11=12, 2+5=7} = 7

d(S,F) = min{dist(S,B) + dist(B,F), dist(S,C) + dist(C,F)}
= min{2+16 = 18, 5+2 =7} = 7

Dist(S,T) = min{d(S,D)+d(D,T), d(S,E)+d(E,T), d(S,F)+d(F,T)}
= min{5+18 = 23, 7+13 = 20, 7+2= 9}
= 9 d(S,T) = $S \rightarrow C \rightarrow F \rightarrow T$

```
Algorithm BGraph($G, k, n, p$)
// Same function as FGraph
{
    $bcost[1] := 0.0$;
    for $j := 2$ to $n$ do
    { // Compute $bcost[j]$.
        Let $r$ be such that $\langle r, j\rangle$ is an edge of
        $G$ and $bcost[r] + c[r, j]$ is minimum;
        $bcost[j] := bcost[r] + c[r, j]$;
        $d[j] := r$;
    }
    // Find a minimum-cost path.
    $p[1] := 1$; $p[k] := n$;
    for $j := k - 1$ to 2 do $p[j] := d[p[j + 1]]$;
}
```

0/1 knapsack

Earlier we have discussed Fractional Knapsack problem using Greedy approach. We have shown that Greedy approach gives an optimal solution for Fractional Knapsack.

In this dynamic programming problem we have n items each with an associated weight and value (benefit or profit).

The objective is to fill the knapsack with items such that we have a maximum profit without crossing the weight limit of the knapsack.

Since this is a 0-1 knapsack problem hence we can either take an entire item or reject it completely. We can not break an item and fill the knapsack.

Point to remember

- In this problem we have a Knapsack that has a weight limit W.
- There are items i1, i2, ..., in each having weight w1, w2, … wn and some benefit (value or profit) associated with it v1, v2, ... vn
- Our objective is to maximize the benefit such that the total weight inside the knapsack is at most W.
- Since this is a 0-1 Knapsack problem so we can either take an entire item or reject it completely. We can not break an item and fill the knapsack.

Example:

Assume that we have a knapsack with max weight capacity W= 5. Our objective is to fill the knapsack with items such that the benefit (value or profit) is maximum.

Following table contains the items along with their value and weight.

Item i	1	2	3	4
Value	100	20	60	40
Weight	3	2	4	1

Total Items : 4

Total capacity of the knapsack is 5

Now we create a value table v[i,w], where I denotes number of items. W denotes weights of the items. Rows denotes the items and column denotes the weight.

v[i,w]	Wt=0	1	2	3	4	5
Item =0	0	0	0	0	0	0
1(3)	0	0	0	100	100	100
2(2)	0	0	20	100	100	120
3(4)	0	0	20	100	100	120
4(1)	0	40	40	100	140	140

```
If w[i] > w then
        V[i,w]= v[i-1,w]
Else
        V[i,w] = max{ v[i-1,w], val[i] + v[i-1,w-wt[i] ]}
Endif
```

Item-1 (3,100) (i=1)	Weight
V[1,0] =V[1-1,0] = 0	0
V[1,1] =V[1-1,1] = 0	1
V[1,2] =V[I-1,2] = 0	2
V[1,3] = Max{0,100+v[0,3-3] =100	3
V[1,4] = Max{0, 100+v[0,4-3]=100	4
V[1,4] = Max{0,100+v[0,5-3]=100	5

Item-2 (2,20) (i=2)	Weight
V[2,0]= V[2-1,0]=v[1,0] = 0	0
V[2,1]= V[2-1,1]=V[1,1]=0	1
V[2,2]= Max{v[1,2]=0, 20+v[1,2-2]=20} = 20	2
V[2,3]= Max{v[1,3]=100, 20+v[1,3-2] =20}=100	3
V[2,4]=max {v[1,4]=100,20+v[1,4-2]=20}=100	4
V[2,5]=max{v[1,5]=100,20+v[1,5-2]=100=120}=120	5

Item-3 (4,60) (i=3)	Weight
V[3,0]= v[3-1,0]=v[2,0]= 0	0
V[3,1]= V[3-1,1]=V[2,1]=0	1
V[3,2]= v[3-1,2] =v[2,2]=20	2
V[3,3]= v[3-1,3]=v[2,3] = 100	3
V[3,4]=max{v[3-1,4]=100, 60+v[3-1,4-3]=v[2,1]=0}=100	4
V[3,5]=max{v[3-1,5]=120,60+v[3-1,5-3]=v[2,2]= 20=80}=120	5

Item-4 (1,40) (i=4)	Weight
V[4,0]= v[4-1,0]=v[3,0]= 0	0
V[4,1]= Max{ v[4-1,1]=0, 40+v[4-1,1-1]=v[3,0]=0=40}=40	1
V[4,2]= Max{v[4-1,2]=20, 40+v[4-1,2-1]=v[3,1]=0=40}=40	2
V[4,3]= Max{v[4-1,3]=100,40+v[4-1,3-1]=v[3,2]=20=60}=100	3
V[4,4]= Max{v[4-1,4]= 100, 40+v[4-1,4-1]=v[3,3]=100=140}=140	4
V[4,5]= Max{v[4-1,5]=140, 40+v[4-1,5-1]=v[3,4]=100=140}=140	5

Maximum value earned v[4,5] = 140

Algorithm 01knapsackvaltable (v, w, n, W)

```
for w = 0 to W do
   v[0, w] = 0
endfor

for i = 1 to n do
   v[i, 0] = 0
endfor

   for w = 1 to W do
      if w[i] > w then
              v[i, w] = v[i-1, w]
      else
          if (val[i]+v[i-1,w-w[i]]) > v[i-1,w]
              v[i, w] = v[i] + v[i-1, w-w[i]]
          else
              v[i, w] = v[i-1, w]
          endif
       endif
   endfor
```

Items that were put inside the knapsack are found using the following rule

Algorithm ItemsPick

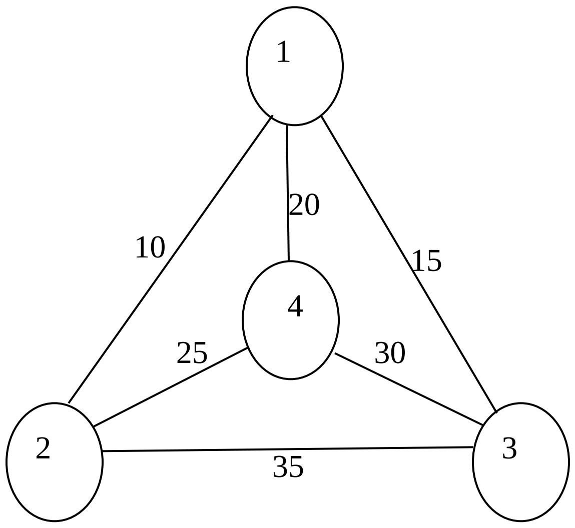
```
Set i =n and w= W
While i > 0 and w > 0 then
   If v[i,w] # v[i-1,w] then
     Mark the ith item and Add to knapsack
     Set i  = i – 1
     Set w = w-wt[i]
  Else
    Set I = I -1
  Endif
Endwhile
```

i	w	V[i,w]	V[I-1,w]	Knapsack
4	5	140	120	4
3	4	100	100	4
2	4	100	100	4
1	4	100	0	4,1=140

So, items we are putting inside the knapsack are 4 and 1.

Travelling Salesman Problem

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?



Naive Solution:

1) Consider city 1 as the starting and ending point.
2) Generate all (n-1)! Permutations of cities.
3) Calculate cost of every permutation and keep track of minimum cost permutation.
4) Return the permutation with minimum cost.

n-1! = (4-1)! = 3 X 2 X 1 =6

1,2,3,4,1 = 10+35+30+20 = 95
1,2,4,3,1 = 10+25+30+15 = 80
1,3,2,4,1 = 15+35+25+20 = 95
1,3,4,2,1 = 15+30+25+10 = 80
1,4,2,3,1 = 20+25+35+15 = 95
1,4,3,2,1 = 20+30+35+10 = 95

Dynamic Programming Approach

	1	2	3	4
1	0	10	15	20
2	10	0	35	25
3	15	35	0	30
4	20	25	30	0

Let’s starts from node 1

C(4,phi) = 20 ; C(3,phi) =15 ; C(2,phi) =10

C(2,{3,4}) = min(d(2,3)+C(3,{4}), d(2,4)+C(4,{3})
C(3,{2,4}) = min(d(3,2)+C(2,{4}), d(3,4)+C(4,{2})
C(4,{2,3}) = min(d(4,2)+C(2,{3}), d(4,3)+C(3,{2})

C(2,{3}) = d(2,3)+C(3,phi) = 35 + 15 = 50
C(2,{4}) = d(2,4)+C(4,phi) = 25 + 20 = 45
C(3,{2}) = d(3,2)+C(2,phi) = 35 + 10 = 45
C(3,{4}) = d(3,4)+C(4,phi) = 30 + 20 = 50
C(4,{2}) = d(4,2)+C(2,phi) = 25 + 10 = 35
C(4,{3}) = d(4,3)+C(3,phi) = 30 + 15 = 45

C(2,{3,4}) = min(35+50), 25+45) = min(85,70)=70
C(3,{2,4}) = min(35+45), 30+35) = min(80,65)=65
C(4,{2,3}) = min(25+50), 30+45) = min(75,75)=75

C(1,{2,3,4}) = min(d(1,2)+ C(2,{3,4)), d(1,3) + C(3,{2,4},
d(1,4) + C(4,{2,3})
= min(10+70, 15+65, 20+75)
= min(80,80,95) = 80

Algorithm Dynamic_Programming_TSP

```
C ({1}, 1) = 0
for s = 2 to n do
   for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
      C (S, 1) = ∞
   endfor
   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}
   Endfor
endfor
Return min (C({1, 2, 3, …, n}, j) + d(j, i))
```

Basic Traversal and Search Techniques

The techniques discussed under this topic is divided into two categories,

- Techniques for Binary Trees (Traversal methods)
- Techniques for Graphs (Search methods)

Trees

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.
Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition
Terminology used in trees

Root	The top node in a tree. The first node is called as Root Node. Every tree must have root node. Root node is the origin of tree data structure.
Parent	In a tree data structure the node which is predecessor of any node is called as **PARENT NODE.**
Child Node	In a tree data structure, the node which is descendant(successor) of any node is called

	as **CHILD Node**.
Leaf	In a tree, elements with no children are called leaves
Edge	In a tree data structure, the connecting link between any two nodes is called as EDGE. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.

Tree is a hierarchical data structure. Main uses of trees include maintaining hierarchical data, providing moderate access and insert/delete operations.

Main applications of trees include:

1. Manipulate hierarchical data.

2. Make information easy to search (see tree traversal).

3. Manipulate sorted lists of data.

4. As a workflow for compositing digital images for visual effects.

5. Router algorithms

6. Form of a multi-stage decision-making (see business chess).

Binary Trees

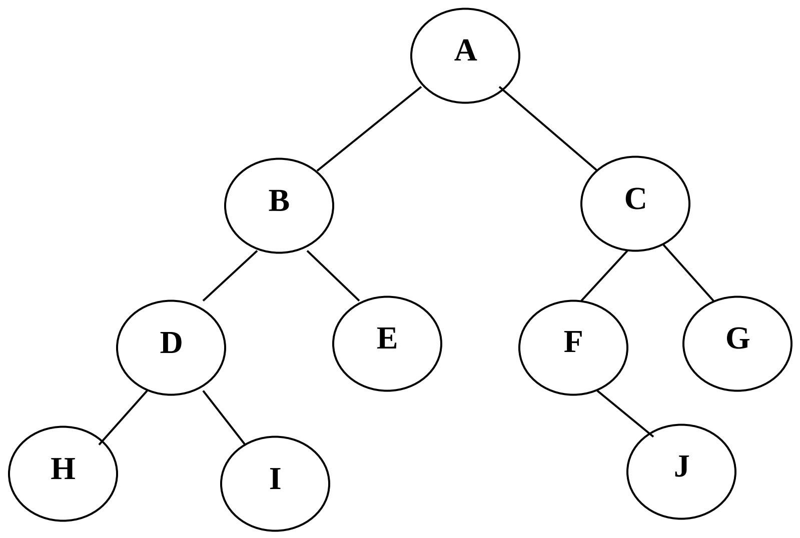
In a normal tree, every node can have any number of children.

Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**.

One is known as left child and the other is known as right child.

Definition: A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children



Binary Tree Traversal

In computer science, tree traversal (also known as tree search) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once.
Such traversals are classified by the order in which the nodes are visited.
Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

Techniques for Binary Tree Traversal

There are three types of binary tree traversals.

1. In - Order Traversal
2. Pre - Order Traversal
3. Post - Order Traversal

1. In-Order Traversal (Left, Root, Right)

   Algorithm Inorder(tree)

   1. Traverse the left subtree, (i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, (i.e., call Inorder(right-subtree))

      H-D-I-B-E-A-F-J-C-G

2. Pre-Order Traversal (Root, Left, Right)

   Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, (i.e.,call Preorder(left-subtree)
   3. Traverse the right subtree, (i.e.,call Preorder(right-subtree))

   A-B-D-H-I-E-C-F-J-G

3. Post-Order Traversal (Left, Right, Root)

   Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree,(i.e.,call Postorder(right-subtree)
   3. Visit the root.

   H-I-D-E-B-J-F-G-A

Graph Traversal
Graph traversal is technique used for searching a vertex in a graph.
The graph traversal is also used to decide the order of vertices to be visit in the search process.
A graph traversal finds the edges to be used in the search process without creating loops that means using graph traversal we visit all vertices of graph without getting into looping path.
Techniques for Graph Tree Traversal
There are two graph traversal techniques and they are as,
1. Depth First Search (DFS)
2. Breadth First Search (BFS)

1. Depth First Search (DFS)

DFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

Steps to be followed in DFS traversal

Step 1 :	Define a Stack of size total number of vertices in the graph.
Step 2 :	Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
Step 3 :	Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
Step 4 :	Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
Step 5 :	When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.
Step 6 :	Repeat steps 3, 4 and 5 until stack becomes Empty.
Step 7 :	When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.

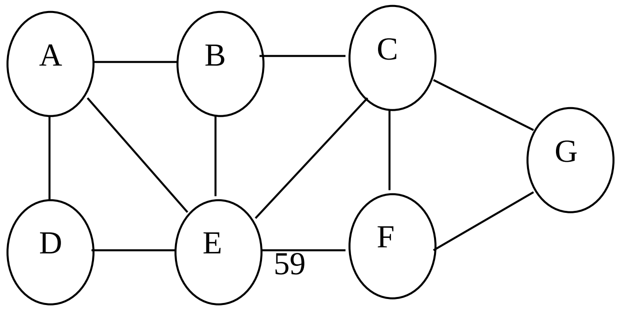
Rules to be followed

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.

Rule 2 – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

Rule 3 – Repeat Rule 1 and Rule 2 until the stack is empty.

Example :

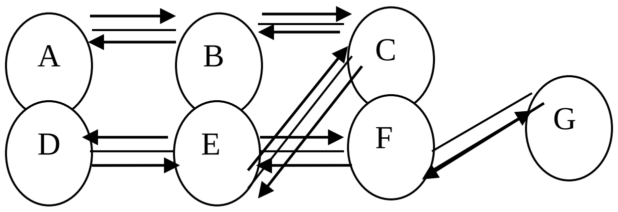


DFS Traversal

Step 1 :	Select a A as starting vertex, Push ‘A’ into stack
Step 2 :	Visit any adjacent vertex of A Which is not visited (B), Push newly visited vertex B on to the stack.
Step 3 :	Visit any adjacent vertex of B Which is not visited (C), Push newly visited vertex C on to the stack.
Step 4 :	Visit any adjacent vertex of C Which is not visited (E), Push newly visited vertex E on to the stack.
Step 5 :	Visit any adjacent vertex of E Which is not visited (D), Push newly visited vertex D on to the stack.
Step 6:	There is no new vertex to be visited from D, So POP D from stack.
Step 7 :	Visit any adjacent vertex of E Which is not visited (F), Push newly visited vertex F on to the stack.
Step 8 :	Visit any adjacent vertex of F Which is not visited (G), Push newly visited vertex G on to the stack.
Step 9 :	There is no new vertex to be visited from G, So POP G from stack.
Step 10 :	There is no new vertex to be visited from F, So POP F from stack.
Step 11 :	There is no new vertex to be visited from E, So POP E from stack.
Step 12 :	There is no new vertex to be visited from C, So POP C from stack.
Step 13 :	There is no new vertex to be visited from B, So POP B from stack
Step 14 :	There is no new vertex to be visited from A, So POP A from stack

Stack became empty so stop DFS Traversal

Final result of DFS spanning tree is as follows



D	Step -5
E	Step -4
C	Step -3
B	Step -2
A	Step -1

G	Step -8
F	Step -7
E	Step -6

G
F
E
C
B
A

Step 9	Step 10	Step 11	Step 12	Step 13	Step 14
F					
E	E				
C	C	C			
B	B	B	B		
A	A	A	A	A	

2. Breadth First Search (BFS)

BFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops.

We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

Steps to be followed in BFS traversal

Step 1 :	Define a Queue of size total number of vertices in the graph.
Step 2 :	Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
Step 3 :	Visit all the **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4 :	When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
Step 5 :	Repeat step 3 and 4 until queue becomes empty.
Step 6 :	When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

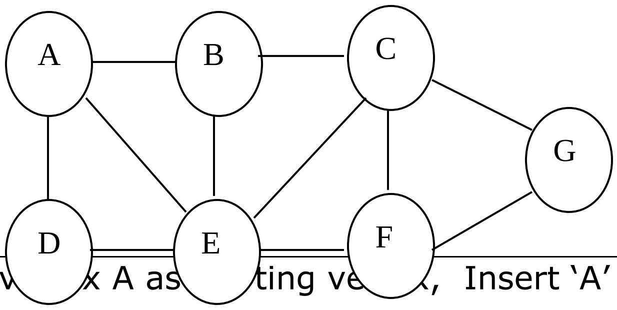
<u>Rules to be followed</u>

Rule 1 – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 – If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 – Repeat Rule 1 and Rule 2 until the queue is empty.

Example :



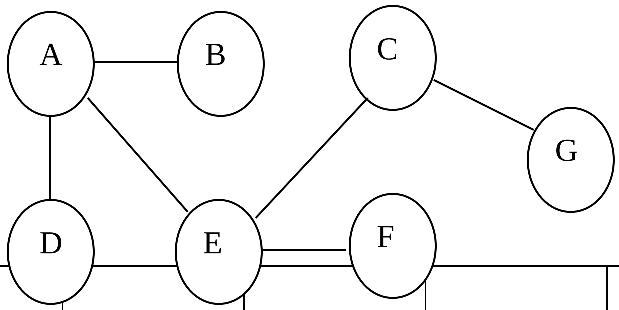
BFS Traversal

Step 1 :	Select the vertex A as starting vertex, Insert 'A' into queue
Step 2 :	Visit all adjacent vertices of A which are not visited (D,E,B). Insert newly visited vertices D,E,B into queue and delete A from the queue.
Step 3 :	Visit all adjacent vertices of D Which are not visited (no vertices). Delete D from queue.
Step 4 :	Visit all adjacent vertices of E Which are not visited (C,F). Insert newly visited vertices C & F into queue and delete E from the queue.
Step 5 :	Visit all adjacent vertices of B Which are not visited (no vertices). Delete C from the queue.
Step 6:	Visit all adjacent vertices of C Which are not visited (G). Insert newly

	visited vertex G into queue and delete C from the queue.
Step 7 :	Visit all adjacent vertices of F Which are not visited (no vertices). Delete F from the queue.
Step 8 :	Visit all adjacent vertices of G Which are not visited (no vertices). Delete G from the queue.

Stack became empty so stop BFS Traversal

Final result of BFS spanning tree is as follows



Step :1

A						

Step :2

	D	E	B			

Step :3

		E	B			

Step :4

			B	C	F	

Step :5

				C	F	

Step :6

					F	G

Step :7

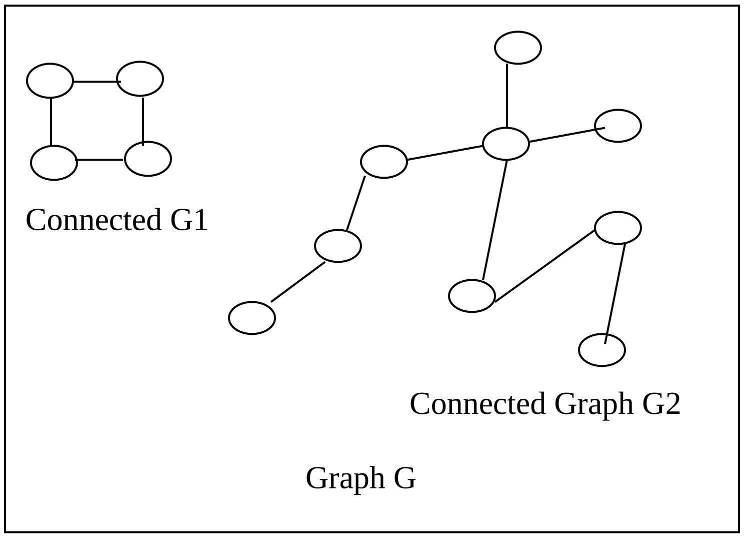
						G

Step :7

Connected Components

In graph theory, a connected component (or just component) of an undirected graph is a sub graph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super graph.

In graph theory, if A is a sub graph of B, then B is said to be a super graph of A.



Spanning Tree

In Graph Theory, a spanning tree T of an undirected graph G is a sub graph, that is, a tree which includes all of the vertices of G, with minimum possible number of edges.

In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree.

A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

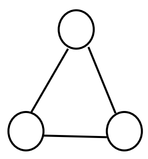


Fig.1

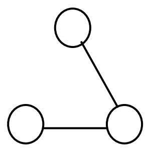


Fig.1(a)

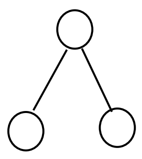


Fig.1(b)

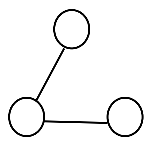


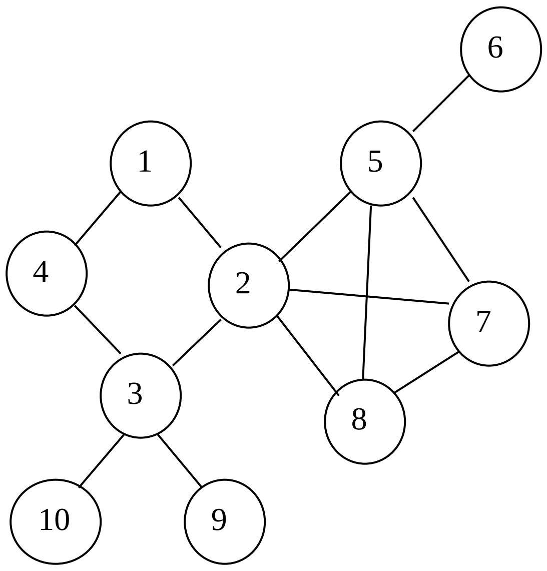
Fig.1(c)

Bi-connected Components and DFS

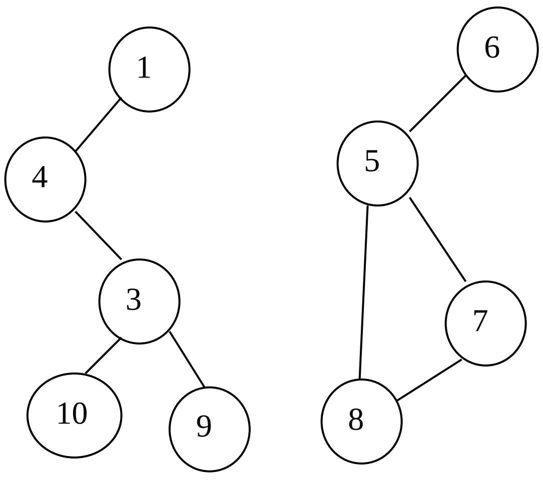
Articulation Point

A Vertex *v* in a connected graph G is an articulation point iff (if and only if) the deletion of vertex *v* together with all edges incident t *v* disconnects the graph into two or more non empty components.

In the following connected graph G, vertex 2 is an articulation point, as the deletion of vertex 2 and edges (1,2), (2,3), (2,5) , (2,7) and (2,8) leaves behind two disconnected non empty components



Graph G



Result of deleting vertex 2

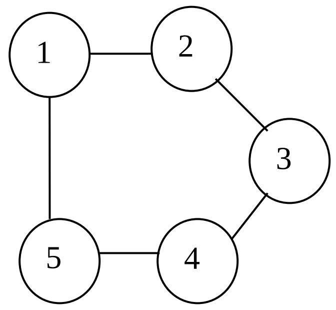
The other articulations points of Graph G are vertex 5 and 3.

Note that if any of the remaining vertices is deleted from graph G then exactly one component is remains.

Bi-connected graph:

A graph G is biconnected iff it contains no articulation point. The above graph G is not biconnected. The following graph G1 is biconnected.

A **bi-connected component** of a graph is a maximal bi-connected subgraph.



Graph G1

The presence of articulation point in a connected graph can be undesirable feature in many cases.

For example, if our graph G represents a communication network with vertices representing communication stations and the edges representing communication lines then the failure of a communication station *i* which is an articulation point would result in loss of communication to points other than station *i.*

On the other hand, if G has no articulation point then if any station *i* fails, we can still communicate between every pair of stations not including station *i*.

Hence the efficient algorithm is required to test if a connected graph is biconnected.

For the case of graphs that are not biconnected, then this algorithm will identify all the articulation points.

Identification of articulation points

Articulation Points: observations

1. The root of the DFS tree is an articulation point if it has two or more children.

2. Any other internal vertex *v* in the DFS tree, if it has a sub tree rooted at a child of *v* that does not have an edge which climb 'higher' than *v, then v* is an articulation point.

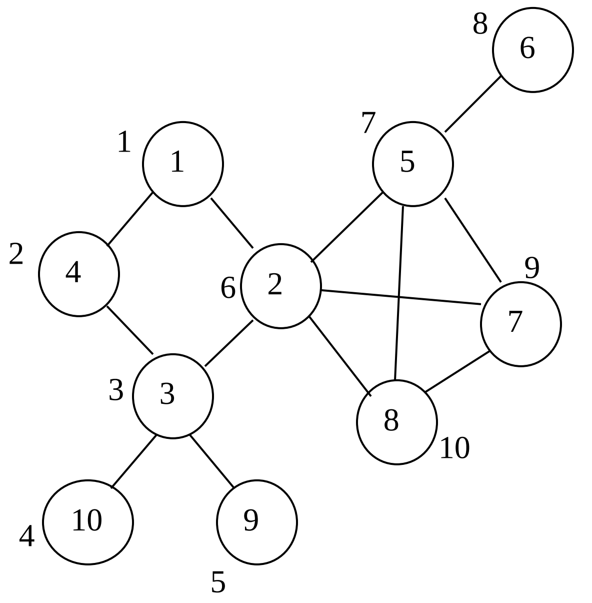
<u>Simple Algorithm</u>

```
Given G(V,E) that is connected
For each u ∈ V
        G' = Remove u from G
        If G' is not connected
                u is an articulation point
        endif
endfor
```

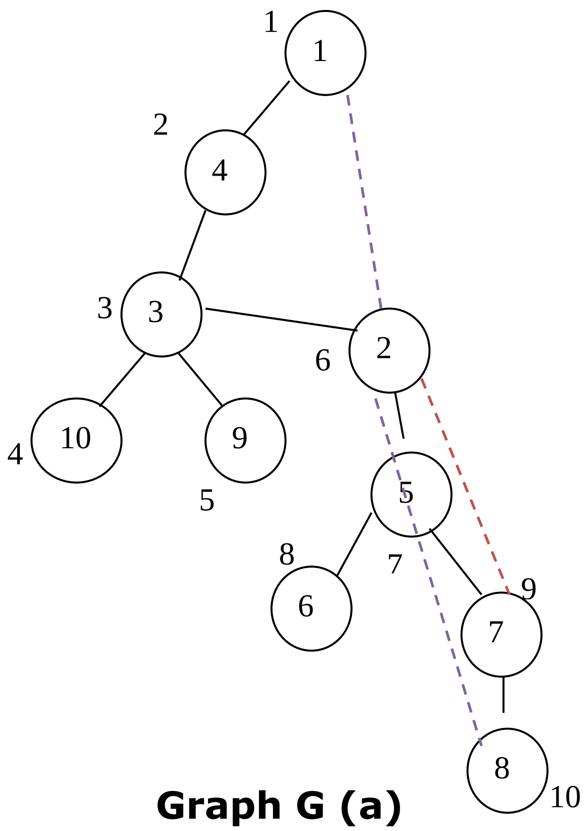
The time complexity of this simple algorithm is O(V * (V+E))

Algorithm Finding_DFS_Articulation

```
For each v in V do
   {
        If (pred[v] == null)
                { //v is a root
                        If (|Adj(v) | > 2)
                                v is an articulation point
                } else
                        for each w in Adj(v) do
                        {
                                If (low(w) >= dfn[v])
                                        v is an articulation point
                        }
        }
```



Graph G



Graph G (a)
DFS Spanning tree of Graph G

Graph G (a) show a depth first spanning tree of the graph G.

In the figures, there is a number outside the each vertex. These numbers corresponds to the order in which a depth first search visits these vertices. This number referred to as the depth first number (DFN) of the vertex.

Thus, DFN(1) = 1, DFN(4) = 2 and DFN(6) = 8.

In the graph G (a), solid edges form the depth first spanning tree. These edges are called tree edges. Broken edges(i.e all remaining edges) are called back edges.

Depth First spanning tree have a property that if (u,v) is any edge in graph G, the relation between u and v is either u is an ancestor of v or v is an ancestor of u.

Low(u) = min {dfn(u), min{L(w) // w is a child of u},
Min {DFN{w} //u,w is a back edge. }

Articulation points : How to climb up :

1. A sub tree can climb to the upper part of the tree by a back edge.
2. A vertex can only climb up to its ancestor

Vertex	1	**2**	**3**	4	**5**	6	**7**	8	9	10
DFN	1	6	3	2	7	8	9	10	5	4
Low	1	1	1	1	6	8	6	6	5	4
Child	4	5	9,10	3	6,7	0	8	0	0	0
low(w) >= dfn[v])	root	6,6 (AP)	5,3 (AP)	1,1 (AP)	8,7 (AP)	0	6,9	0	0	0

Applications of Bi-connected graphs

Biconnected graphs are used in the design of power grid networks.

Consider the nodes as cities and the edges as electrical connections between them, you would like the network to be robust and a failure at one city should not result in a loss of power in other cities. This can be ensured by making the graph biconnected.

It is also used in social network analysis to try and identify the most influential nodes in the network.

Bi-connected components are used in communications Networks.

The vertices represent communication stations and the edges represent communication links. Now suppose that one of the stations that is an articulation point fails. The result is a loss of communication not just to and from that single station, but also between certain other pairs of stations.

Unit – IV

Backtracking: 8 Queens Problems, Sum of Subsets, Graph Colouring, Hamiltonian Cycle and Knapsack Problem.

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems that incrementally builds candidates to the solutions.

Backtracking is a technique used to solve problems with a large search space, by systematically trying and eliminating possibilities.

A standard example of backtracking would be going through a maze. Other examples are crossword, verbal arithmetic, sudoko and many other puzzles.

Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”.

8 Queens Problem

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each other. Find an arrangement of **8** queens on a single chess board such that no two queens are attacking one another.

Thus, a solution requires that no two queens share the same row, column, or diagonal. In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).

Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.

The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.

1) Start in the leftmost column

2) If all queens are placed return true

3) Try all rows in the current column. Do following for every tried row.

    a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.

    b) If placing queen in [row, column] leads to a solution then return true.

    c) If placing queen doesn't lead to a solution then unmark this [row, column] (Backtrack) and go to step (a) to try other rows.

4) If all rows have been tried and nothing worked, return false to trigger backtracking.

Procedure PLACE(k)

```
Global X(1:k)
Integer i, k
For I = 1 to k do
    If X(i) = X(k)   // two queens in the same column
        Or Abs(X(i)-X(k)) = Abs(i-k)  // in the same diagonal
        Return false
    Endif
Endfor
Return true
```

Algorithm NQueens(k,n)

```
Integer k, n, X(1:n)
X(1) = 0; k = 1  // k is the current row, X(k) is current column
While k > 0  // for all rows
    X(k) = X(k) + 1  //move the next column
    While X(k) <= n and not PLACE(k) do
        X(k) = X(K) + 1
    Repeat
    If X(k) <= n   // is position is found
        Then if k =n  // is solution is complete
            Then print X
        Else
            K = k + 1;
            X(k) = 0    // go to next row
        Endif
    Else
        k = k -1  //backtrack
    endif
repeat
end Nqueens
```

Sum of Subsets

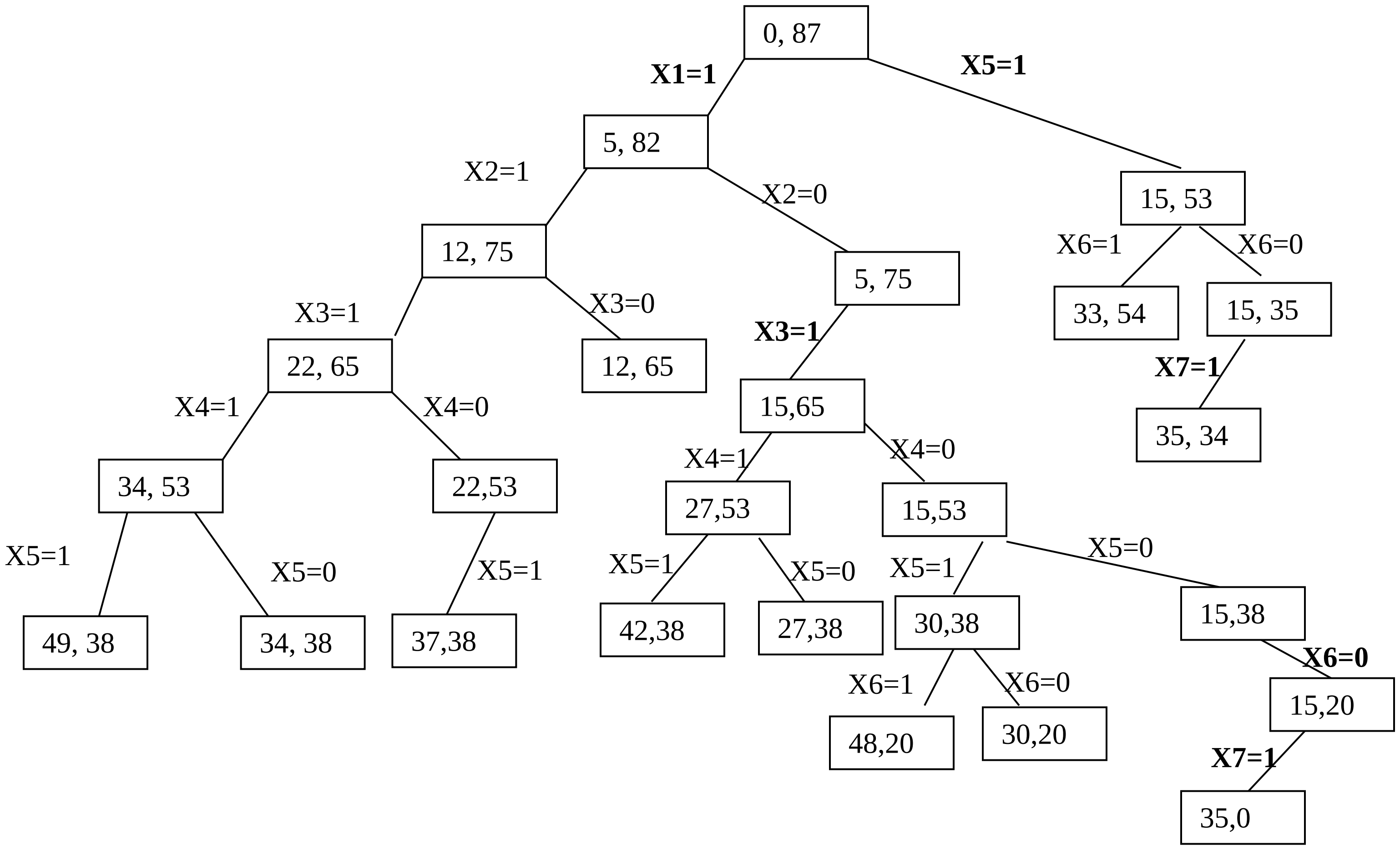
Sum of subsets problem is to find subset of elements that are selected from a given set whose sum adds up to a given number. It is considered that the given set contains,

- non-negative values.
- no duplicates are presented.

Example

Let S be a set and given sum m = 35

S= {5,7,10,12,15,18,20}



So the subsets that add ups to sum 35 are,
{{15,20},{18,7,10}, {5,10,20}, {18,12,5}}

Example:

Solve following problem and draw portion of state space tree M = 35,

S = (5, 7, 10, 12, 15, 18, 20)

Subset	Sum	Description
Empty {}	0	Initial
{5}	5	Add first element
{5,7}	12 (12 < m)	Add next element
{5,7,10}	22 (22 < m)	Add next element
{5,7,10,12)	34 (34 < m)	Add next element
{5,7,10,12,15)	49 (49 > m)	Backtrack
{5,7,10,12,18}	52 (52 > m)	Backtrack
{5,7,10,12,20}	54 (54 > m)	Bbacktrack
{5,10}	15 (15 < m)	Add next element
{5,10,12)	27 (15 < m)	Add next element
{5,10,12,15}	42 (42 > m)	Backtrack
{5,10,12,18}	45 (45 > m)	Backtrack
{5,10,12,20}	47 (47 > m)	Backtrack
{5,10,15}	30 (30 < m)	Add next element
{5,10,15,18}	48 (48 > m)	Backtrack
{5,10,15,20}	50 (50 > m)	Backtrack
{5,10,18}	33 (33 < m)	Add next element
{5,10,18,20}	53 (53 < m	Backtrack
{5,10,20}	35	**Solution obtained m= 35**
{5,12}	17 (17 < m)	Add next element
{5,12,15}	32 (32 < m)	Add next element
{5,12,15,18}	50 (50 > m)	Backtrack
{5,12,18}	35	**Solution obtained m= 35**
{5,15}	20 (20 < m)	ADD next element
{5,15,18}	38 (38 > m)	Backtracking
{5,15,20}	40 (40 > m)	Backtracking
{5,18}	23 (23 < m)	Add next element
{5,18,20}	43 (43 >m)	Backtracking
{5,20}	25 (25 < m)	Add next Element, but no element present, so Backtracking

S= {5,7,10,12,15,18,20}

Start with 5th element (15)

Subset	Sum	Description
Empty {}	0	Initial
{15}	15	Add 5th element
{15,18}	33 (33 < m)	Add next element
{15,18,20}	53 (53 > m)	Backtrack
{15,20}	35	Solution obtained m= 35

Algorithm:

Let, S is a set of elements and m is the expected sum of subsets. Then:

1. Start with an empty set.
2. Add to the subset, the next element from the list.
3. If the subset is having sum m then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible then repeat step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Recursive Backtracking algorithm for sum of subsets problem

Algorithm SUMOFSUBS(s, k, r)

```
//initiations
Global integer M, n // M-Capacity, n – number of elements in set
Global real W(1:n)  //Given Set
Global boolean X(1:n) // if x(1) is either 1(included in tree) or 0
Real r, s     //s be cumulative sum,
Integer k, j
//generate left child.  Note that s+W(k) <=M

X(k) = 1
If s+X(k) = M //subset found
      return subset
else
  if s+w(k)+w(k+1) <= M
      call SUMOFSUBS(s+w(k), k+1, r-W(k))
  endif
endif

//generate right child.  Note that s+W(k) >M

If s+r-W(k) >= M  and s+W(k+1) <= M
      X(k) = 0
      call SUMOFSUBS(s, k+1, r-W(k))
endif
```

Where

s = $\sum_{j=1}^{k-1}(W(j) x\, X(j))$

r = $\sum_{j=k}^{k-n}(W(j))$

Graph Coloring

Let G be a graph and m be a positive integer. It is to find whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used.

Given a graph of vertices and edges, we want to colour the nodes in such a way that no two adjacent vertices share a same colour, using fewest number of colours.

The problem has two versions,

- m-coloring decision problem :
  we want know it can be coloured with the given colours or not?
- m-coloring optimization Problem :
  we want to know minimum how many colours are required for colouring the vertices.

m is called chromatic number.

The minimum number of colors required for vertex coloring of graph 'G' is called as the chromatic number of G, denoted by X(G).

Problems

1. How many ways the vertices can be coloured and what are the possible colours.

2. Colours are given we want to know whether the graph can be coloured with those vertices or not.

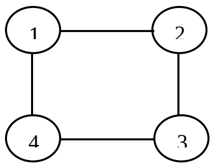
3. Colours are not given we want to how many colour we need colour the vertices

If d is the degree of given graph G, then it can be colored with (d+1) colors.

The degree of a vertex of a graph is the number of edges incident to the vertex, with loops counted twice.

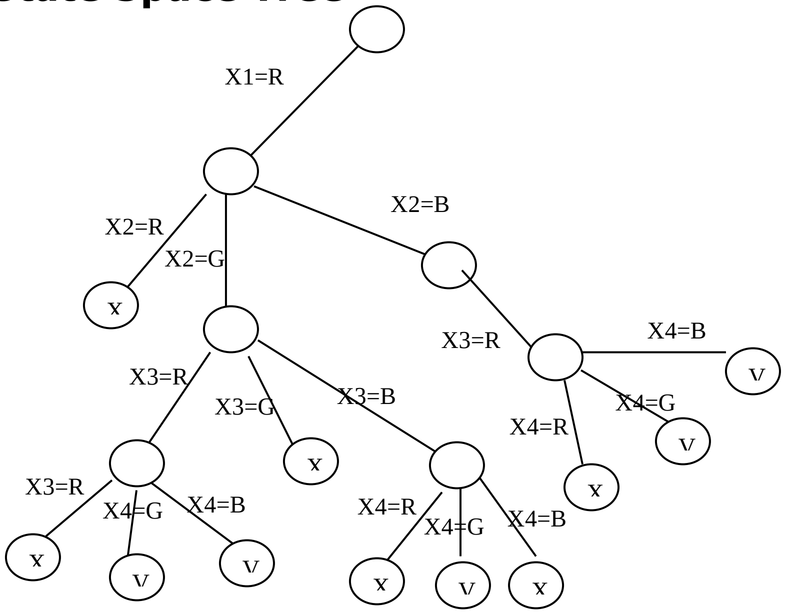
For Example:

Given graph G



m = 3 {R,G,B}

State Space Tree



1	RGRG
2	RGRB
4	RGBG
5	RBRG
6	RBRB

=1+3+(3X3)+(3X3X3)+(3X3X3X3)

$=C^n$

```
Algorithm GraphColour(k)
       For C = 1 to m
                If  issafe(k,c)
                     X[k] = c;
                     If  k+1 < n
                            GraphColour(k+1)
                     Else
                            Print x[]
                     Endif
              Endif
       End for
```

Where,
 k is the node we are going to colour in the level of recursion.

X[k] is the array that holds the current colour at each node.

```
Algorithm issafe(k,c)
       For j=1 to n
         If G[k][j] == 1 and c ==x[j]
              Return false
         Endif
       End for
       Return true
```

Adjacency Matrix(G)

N	1	2	3	4
1	1	1	0	1
2	1	1	1	0
3	0	1	1	1
4	1	0	1	1

Graph Colour k	C	If G[K][J]	if C=X[j]	Return
1	1	G[1][1]=1	1≠0	Return True X[1]=1 (red)
1	1	G[1][2]=1	1≠0	
1	1	G[1][3]=0	1≠0	
1	1	G[1][4]=1	1≠0	
2	1	G[2][1]=1	1=1	Return False
2	2	G[2][1]=1	2≠1	Return True X[2]=2 (green)
2	2	G[2][2]=1	2≠0	
2	2	G[2][3]=1	2≠0	
2	2	G[2][4]=0	2≠0	
3	1	G[3][1]=0	1=1	Return True X[3]=1 (red)
3	1	G[3][2]=1	1≠2	
3	1	G[3][3]=1	1≠0	
3	1	G[3][4]=1	1≠0	
4	1	G[4][1]=1	1=1	Return False
4	2	G[4][1]=1	2≠1	Return True X[3]=2 (green)
4	2	G[4][2]=0	2=2	
4	2	G[4][3]=1	2≠1	
4	2	G[4][4]=1	2≠0	

The time complexity of this backtracking algorithm is O(m^n). Where m is the number of colours and n is the number of vertices.

Graph coloring is one of the most important concepts in graph theory. It is used in many real-time applications of computer science such as –

Clustering, Data mining , Image capturing, Image segmentation ,Networking, Resource allocation, Processes scheduling

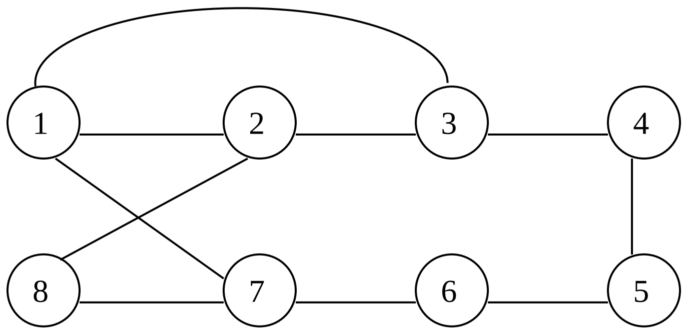
Hamiltonian Cycle (William Rowan Hamilton -1859)

Hamiltonian cycle is a path in a directed or undirected graph that visits each vertex exactly once. The problem is to check whether a given graph contains Hamiltonian cycle or not. A graph that contains a Hamiltonian cycle is called a traceable graph.

In other words, A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path.

In other words, A graph is Hamiltonian-connected if for every pair of vertices there is a Hamiltonian path between the two vertices.

Consider the following graph G1



The graph G1 has Hamiltonian cycles:

1,3,4,5,6,7,8,2,1 and
1,2,8,7,6,5,4,3,1.

The backtracking algorithm helps to find Hamiltonian cycle for any type of graph.

Procedure:

1. Define a solution vector X(Xi……..Xn) where Xi represents the ith visited vertex of the proposed cycle.

2. Create a cost adjacency matrix for the given graph.

3. The solution array initialized to all zeros except X(1)=1,because the cycle should start at vertex '1'.

4. Now we have to find the second vertex to be visited in the cycle.

5. The vertex from 1 to n are included in the cycle one by one by checking 2 conditions,

    i. There should be a path from previous visited vertex to current vertex.

    ii. The current vertex must be distinct and should not have been visited earlier.

6. When these two conditions are satisfied the current vertex is included in the cycle, else the next vertex is tried.

7. When the nth vertex is visited we have to check, is there any path from nth vertex to first vertex. if no path, the go back one step and after the previous visited node.

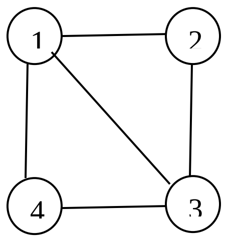
Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm:(Finding all Hamiltonian cycle)

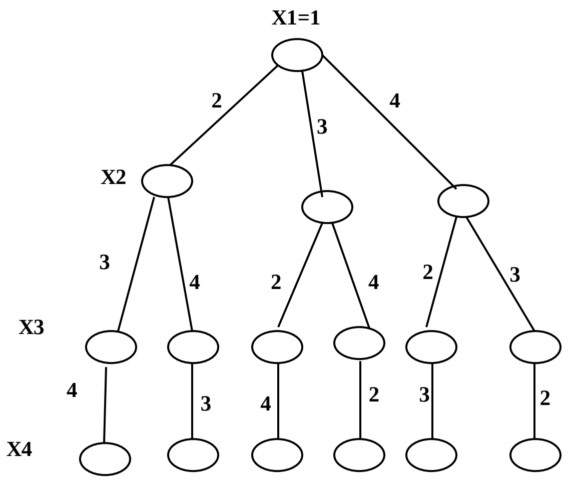
```
Algorithm Hamiltonian (k)
Loop
        Next value (k)
If (x (k)=0) then return;
  If k=n then
      Print (x)
      exit
   Else
    Hamiltonian (k+1);
End if
Repeat
Algorithm Nextvalue (k)
```

```
Repeat
          X [k]=(X [k]+1) mod (n+1); //next vertex
          If (X [k]=0)
                return;
          Endif

         If (G [X [k-1], X [k]] ≠ 0) then
                  For j=1 to k-1 do if (X [j]=X [k]) then break;
                  // Check for distinction.
                  If (j=k) then       //if true then the vertex is distinct.
                            If ((k<n) or ((k=n) and G [X [n], X [1]] ≠ 0))
                                Return
                            Endif
                  Endif
         Endif
Until (false);
```



X= 1,2,3,4,1
X =1,4,3,2,1



Knapsack Problem using Backtracking:

- The problem is similar to the zero-one (0/1) knapsack optimization problem is dynamic programming algorithm.

- We are given 'n' positive weights W_i and 'n' positive profits P_i, and a positive number 'm' that is the knapsack capacity, the is problem calls for choosing a subset of the weights such that,

$$\sum_{1 \le i \le n} WiXi \le \text{m and } \sum_{1 \le i \le n} PiXi \text{ is Maximized.}$$

$X_i \rightarrow$ Constitute Zero-one valued Vector.

- The Solution space is the same as that for the sum of subset's problem.

- Bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node.

- The profits and weights are assigned in descending order depend upon the ratio.

(i.e.) $Pi/Wi \ge P(I+1) / W(I+1)$

Solution :

- After assigning the profit and weights ,we have to take the first object weights and check if the first weight is less than or equal to the capacity, if so then we include that object (i.e.) the unit is 1.(i.e.) $K \rightarrow 1$.

- Then We are going to the next object, if the object weight is exceeded that object does not fit. So unit of that object is '0'.(i.e.) K=0.
- Then We are going to the bounding function ,this function determines an upper bound on the best solution obtainable at level K+1.

- Repeat the process until we reach the optimal solution.

Algorithm:

Algorithm Bknap(k,cp,cw)

// ‘m’ is the size of the knapsack; ‘n’ → no.of weights & profits. W[]&P[] are the //weights & weights. P[I]/W[I] ≥ P[I+1]/W[I+1].
//fw→Final weights of knapsack.
//fp→ final max.profit.
//x[k] = 0 if W[k] is not the knapsack,else X[k]=1.

```
{
      // Generate left child.
       If((W+W[k] ≤m) then
       {
             Y[k] =1;
              If(k<n) then Bnap(k+1,cp+P[k],Cw +W[k])
                 If((Cp + p[w] > fp) and (k=n)) then

                   {
                      fp = cp + P[k];
                      fw = Cw+W[k];
                     for j=1 to k do X[j] = Y[j];
                 }
   }

  if(Bound(cp,cw,k) ≥fp) then
  {
       y[k] = 0;
     if(k<n) then Bnap (K+1,cp,cw);
   if((cp>fp) and (k=n)) then
      {
          fp = cp;
           fw = cw;
             for j=1 to k do X[j] = Y[j];
        }
   }
}
```

Algorithm for Bounding function:

```
Algorithm Bound(cp,cw,k)
// cp→ current profit total.
//cw→ current weight total.
//k→the index of the last removed item.
//m→the knapsack size.

{
    b=cp;
    c=cw;
    for I =- k+1 to n do
  {
        c= c+w[I];
      if (c<m) then b=b+p[I];
          else return b+ (1-(c-m)/W[I]) * P[I];
}
return b;
}
```

Unit - V

Branch and Bound: Least Cost Search. Bounding: FIFO Branch and Bound and LC Branch and Bound. 0/1 Knapsack Problem, Travelling Salesman Problem.

Branch and Bound

- It is generally used for optimization problems.
- As the algorithm progresses, a tree of sub problems is formed.
- The original problem is considered as a root problem.
- A method is used to construct an upper and lower bound for a given problem.
- At each node, apply the bounding methods,
- If the bound matches, it is deemed(judged) a feasible solution to that particular sub problem.
- If bounds do not match, partition the problem represented by the node, and make sub problems into children nodes.
- Continue, using the best known feasible solution to solve sections of the tree, until all nodes have been solved.

Branch and bound is an algorithm design paradigm (pattern) which is generally used for solving combinatorial optimization problems.

(Combinatorial optimization means searching for an optimal solution in a finite or countably infinite set of potential solutions)

These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case.

Branch and Bound solve these problems relatively quickly.

A branch-and-bound algorithm consists of a systematic enumeration (list) of candidate(possible) solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.

The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

Branch & Bound discovers branches within the complete search space by using estimated bounds to limit the number of possible solutions. The different types (FIFO, LIFO, LC) define different 'strategies' to explore the search space and generate branches.

FIFO (first in, first out): always the oldest node in the queue is used to extend the branch. This leads to a breadth-first search, where all nodes at depth d are visited first, before any nodes at depth d+1 are visited.

LIFO (last in, first out): always the youngest node in the queue is used to extend the branch. This leads to a depth-first search, where the branch is extended through every 1st child discovered at a certain depth, until a leaf node is reached.

LC (lowest cost): the branch is extended by the node which adds the lowest additional costs, according to a given cost function. The strategy of traversing the search space is therefore defined by the cost function.

What is the difference between Backtracking and Branch and Bound Method?

Backtracking	Branch and Bound
It is used to find all possible solutions available to the problem.	It is used to solve optimization problem.
It traverse tree by DFS(Depth First Search).	It may traverse the tree in any manner, DFS or BFS.

It realizes that it has made a bad choice & undoes the last choice by backing up.	It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.
It search the state space tree until it found a solution.	It completely searches the state space tree to get optimal solution.
It involves feasibility(possibility) function.	It involves bounding function.

General Branch and Bound Algorithm

1. Starting by considering the root node and applying a lower-bounding and upper-bounding procedure to it.
2. If the bounds match, then an optimal solution has been found and the algorithm is finished.
3. If they do not match, then algorithm runs on the child nodes

Upper Bound:

In a given set, a number which is greater than or equal to all the elements is known as upper bound for that set. More precisely, let us assume a set of real numbers be represented by S. Then, the upper bound for this set would be a number, say k, in such a way that for all x belongs to S, there must be k ≥≥ x.

Lower Bound: The lower bound is said to be the number that is less than or equal to every element in a given set. Let us recall more formal definition. Assume that S be a set of real numbers and k be some number. Then, k is said to be the lower bound for set S, if for every x belongs to, there exists k ≤≤ x.

A set is said to be "bounded above" if it has an upper bound. In the same way, it is called "bounded below" if it has a lower bound.

The algorithm depends on the efficient estimation of the lower and upper bounds of a region/branch of the search space and approaches exhaustive enumeration as the size (n-dimensional volume) of the region tends to zero

FIFO Branch and Bound

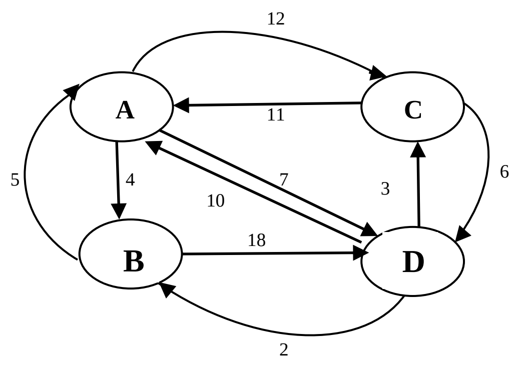
In branch-and-bound terminology, a BFS-like state space search will be called FIFO (First In First Out) search as the list of live nodes is a first-in-first-out list (or queue).

LIFO Branch and Bound

A D-search-like state space search will be called LIFO (Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack).

Travelling Salesman Problem

We are given a set of cities and distances between every pair of cities. The problem is to find the shortest possible route that visit every city exactly one and returns to the starting city.



	A	B	C	D	Reduce by
A	∞	4	12	7	4
B	5	∞	∞	18	5
C	11	∞	∞	6	6
D	10	2	3	∞	2

Step 1: Write the initial cost matrix
Reduce it

After row reduction

	A	B	C	D
A	∞	0	8	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	1	∞
	-	-	1	-

Colum
Reduced

	A	B	C	D
A	∞	0	7	3
B	0	∞	∞	13
C	5	∞	∞	0
D	8	0	0	∞

(Reduced matrix)

So cost of note-1 is

$$\boxed{\text{Cost (1) } = 4 + 5 + 6 + 2 + 1 = 18}$$

<u>Step – 2:</u> Choosing to go to Vertex B: Path A → B

→ From the reduced matrix of Step-1, M [A,B] = 0

→ Set row A and Colum B to ∞

→ Set M[B,A] = ∞

→ Now resulting matrix is

	A	B	C	D	
A	∞	∞	∞	∞	-
B	∞	∞	∞	13	13
C	5	∞	∞	0	-
D	8	∞	0	∞	-

Row reduction

→ Now we reduce this matrix and find the cost

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	5	∞	∞	0
D	8	∞	0	∞
	5	-	-	-

Colu

Reduced

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	0	∞	∞	0
D	3	∞	0	∞

So cost (2) = Cost (1) = Reduction + M[A,B]

= 18+18+0=36

Step 3 : Choosing to go to Vertex C, Path A→ C

→ From the reduced matrix of step-1, M[A,B] = 7

→ Set row A and Colum C to ∞

→ Set M[C,A] = ∞

→ Now reduced matrix is

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	13
C	∞	∞	∞	0
D	8	0	∞	∞

This Matrix Is already Reduced

So Cost (3) = Cost(1)+ Reduction +M[A,C]
= 18+0+7=25

Choosing to go to vertex D : Path A → D

→ from reduced matrix of step 1 – M[A, D] = 3

→ Set Row-a and Colum D to ∞

→ Set M[D,A] to ∞

→ Now the reduced matrix is

	A	B	C	D	
A	∞	∞	∞	∞	-
B	0	∞	∞	∞	-
C	5	∞	∞	∞	5
D	∞	0	0	∞	-

$\xrightarrow[\text{Reduced}]{\text{Row}}$

	A	B	C	A
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	0	∞	∞	∞
D	∞	0	0	0

So cost of (4) = Cost(1) + reduction + M[A,D]
= 18+5+3=26

Thus we have

Cost [2] = 36 Cost [3] = 25 Cost [4) = 26

So we choose the lowest cost path A → C

Step 4: Choosing to go to vertex B path A → C → B

→ from the reduced matrix of step-2 M[C,B] = ∞

→ Set Row C and Colum D to ∞

→ Set M[B,A] = ∞

→ Now the reduce matrix is

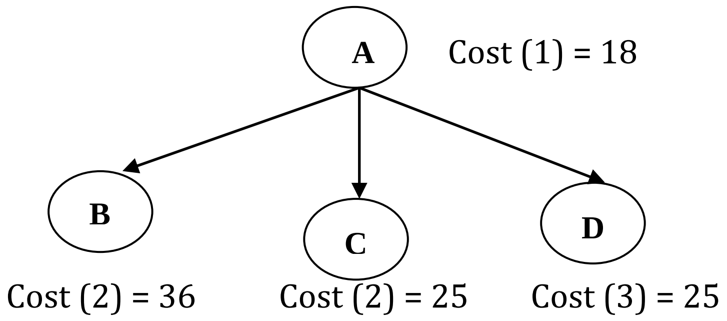
	A	B	C	D	
A	∞	∞	∞	∞	
B	∞	∞	∞	13	-13
C	∞	∞	∞	∞	
D	8	∞	∞	∞	-8

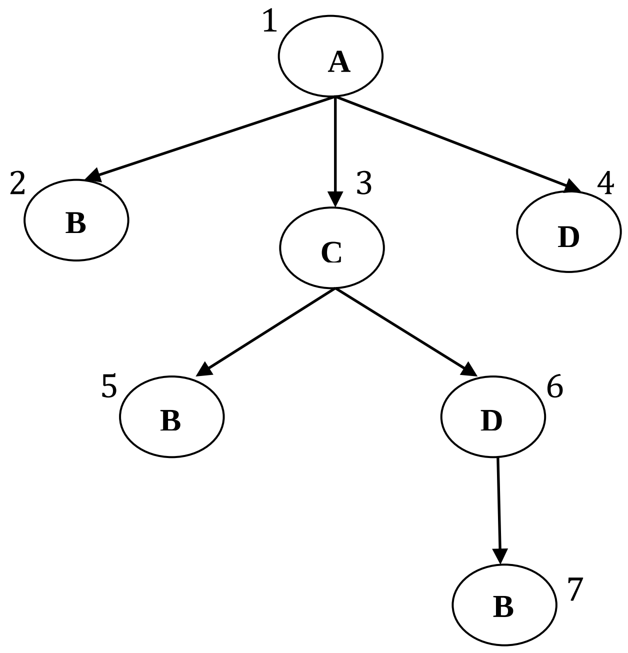
$\xrightarrow[\text{Reduced}]{\text{Row}}$

	A	B	C	A
A	∞	∞	∞	∞
B	∞	∞	∞	0
C	∞	∞	∞	∞
D	0	∞	∞	∞

Cost [5] = Cost [3] + Reducing + M[C,B]

= 25+13+8+∞ = ∞





A → C → D

Choosing to go to vertex D Path A → C → D

→ From the reduce matrix of step-2 M[C,D] = 0

→ Set Row C and Colum D to ∞

→ Set M[D,A] = ∞

→ Now the reduced matrix is

	A	B	C	D
A	∞	∞	∞	∞
B	0	∞	∞	∞
C	∞	∞	∞	∞
D	∞	0	∞	∞

-
-
-
-

Cost (6) = Cost (3)+ Reduction M[C,D]
= 25 + 0 + 0
= 25

Choosing to go to vertex B Mode (7) (Path A → C → D → B)

→ From the reduce matrix of step-3 M[D,B] = 0

→ Set Row D and Colum B to ∞

→ Set M[B,A] = ∞

→ Now the reduced matrix is

	A	B	C	D
A	∞	∞	∞	∞
B	∞	∞	∞	∞
C	∞	∞	∞	∞
D	∞	∞	∞	∞

This matrix is already completely reduced. So the cost of mode 7
Cost [7] = Cost[6]+Reduced[D,B]
= 25 + 0 + 0 = 25

This optional path is
A → C → D → B → A with cost of 25
[12 + 6 + 2 + 5 = 25]