Chapter 1: Introduction and Elementary Data Structures

1.1. Introduction to Algorithm analysis

Definition

A step-by-step procedure for solving a problem or accomplishing some end especially by a computer

Characteristics of an algorithm

Any algorithm that we write must satisfy the following characteristics

1. **Input**: An algorithm must have zero or more inputs.
2. **Output**: An algorithm must produce one or more outputs.
3. **Definiteness**: Each instruction in an algorithm should be clear and unambiguous.
4. **Finiteness**: An algorithm must terminate after a finite no of instructions.
5. **Effectiveness**: Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

The **study of an algorithm** includes many important and active areas of research. There are four areas of study one can identify.

1. How to devise an algorithm :

There are five design strategies with the help of that we can design easy and good algorithms,

- Divide and conquer
- Greedy method
- Dynamic programming
- Backtracking
- Branch and Bound

By mastering these entire techniques one can easily design new and useful algorithms. They also minimize the time complexity of an algorithm.

2. How to analyze an algorithm

Estimating the efficiency of an algorithm in terms of time and space an Algorithm requires is called analysis of an algorithm. It is also called as performance analysis. This is an important result that allows us to make quantitative judgments about the value of one algorithm over another. It also allows us to predict whether the software will meet any efficiency constraints that exist.

3. How to validate an algorithm

The process of producing correct answer to given input is called algorithm validation. Once an algorithm is designed, it is necessary to show that it computes Correct answer for all possible legal inputs. Once the validity of the method has Been shown a program can be written and a second phase begins. This is called as program verification.

4. How to test an algorithm

Testing a program consists of two phases:

- Debugging
- Profiling or performance measurement

Debugging is the process of running the programs on sample data sets to determine error occur and, if so, to correct them.

Profiling is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

Pseudo code for expressing algorithms

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.A compound statement can be represented as a block. The body of a procedure also forms a block. Statements are delimited by a semicolon.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

```
Node = Record
{
   data type – 1   data-1;
         .
         .
  data type – n  data – n;
  node * link;
}
```

   Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

5. Assignment of values to variables is done using the assignment statement.

```
<Variable>:= <expression>;
```

6. There are two Boolean values TRUE and FALSE. In order to produce these values, the logical operators and relational operators are provided.

→ Logical Operators AND, OR, NOT
→Relational Operators <, <=,>,>=, =, ! =

7. The following looping statements are employed.

For, while and repeat-until

While Loop:

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

As long as condition is true, the statements get executed. When the condition becomes false, the loop is exited.

For Loop:

```
For variable: = value-1 to value-2 Step step do

{
  <Statement-1>
        .
        .
        .
<Statement-n>
}
```

Here value1, value2, and step are arithmetic expressions. The clause 'Step step' is optional and taken as +1 if it does not occur.

repeat-until:

```
repeat
        <Statement-1>
                .
                .
                .
        <Statement-n>
until<condition>
```

Repeat the set of statements as long as condition is false.

8. A conditional statement has the following forms.

   → If <condition> then <statement>

   → If <condition> then <statement-1> else <statement-1>

   Case statement:

```
Case
{
        : <condition-1> : <statement-1>
                .
                .
                .
        : <condition-n> : <statement-n>
        : else : <statement-n+1>
}
```

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:

    Algorithm, the heading takes the form,

    Algorithm Name (Parameter lists)

    Where name is the name of the procedure and <parameter list> is a list of procedural parameters. The body has one or more statements enclosed with braces {and}.

Example 1: Algorithm that finds and returns the maximum of n given numbers.

```
1.  Algorithm Max(A,n)
2.  // A is an array of size n
3.  {
4.  Result := A[1];
5.  for i:= 2 to n do
6.     if A[i] > Result then
7.           Result :=A[i];
8.    return Result;
}
```

Where A and n are procedural parameters. Result and i are local variables.

Example 2: Selection Sort:

- Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
- A Simple solution given by the following.
- "From those elements that are currently unsorted ,find the smallest & place it next in the sorted list."

Sample Algorithm:

```
1. for i: = 1 to n do
2. {
3.              Examine a[i] to a[n] and suppose the smallest element is at a[j];
4.              Interchange a[i] and a[j];
5.  }
```

→ There are two subtasks here. The first subtask is to find the smallest element in the list and the second subtask is to interchange that smallest element with the first element of the list.

Algorithm:

```
1. Algorithm Selection sort (a, n)
2. // Sort the array a [1: n] into non-decreasing order.
3. {
4.       for i: =1 to n do
5.       {
6.             j:=i;
7.             for k: =i+1 to n do
8.                   if (a[k]<a[j]) then j:=k;
9.                   t:=a[i];
10.                  a[i]:=a[j];
11.                  a[j]:=t;
12        }
13  }
```

1.1.1. Asymptotic Notations

Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

<u>Space Complexity:</u> The space complexity of an algorithm is the amount of memory it needs to run to completion.

→ The Space needed by an algorithm is seen to be the sum of the following component.

I. A fixed part that is independent of the characteristics (eg: number, size) of the inputs and outputs.

The part typically includes the instruction space (i.e. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

II. A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.

The space requirement S(p) of any algorithm p may therefore be written as, S (P) = c+ Sp (Instance characteristics), where 'c' is a constant.

Time Complexity:

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

- The time T(p) taken by a program P is the sum of the compile time and the run time(execution time).

- The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation. This rum time is denoted by t_p (instance characteristics).

- The number of steps any problem statement is assigned depends on the kind of statement.

For example, comments → 0 steps.

Assignment statements → 1 steps.

[This does not involve any calls to other algorithms]

We can find the number of program steps needed by a program to solve a particular problem instance in one of the two ways.

1. We introduce a new variable, count into the program. This is a global variable with initial value 0. Statement to increment count by the appropriate amount are introduced into the program

Algorithm:

```
Algorithm sum(a,n)
{
    s= 0.0;
    count = count+1; // count is global ; it is initially zero.
for i:=1 to n do
{
 count: =count+1; // For ‘ for loop ‘
s=s + a[i];
count:=count+1; // For assignment
}
count=count+1; // For last time of for loop
count=count+1; // for the return
return s;
}
```

- If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum executes a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributed by each statement.

- First determine the number of steps per execution (s/e) of the statement and the total number of times (i.e., frequency) each statement is executed.
- By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	S/e	Frequency	Total
1. Algorithm Sum(a,n)	0	-	0
2.{	0	-	0
3. S=0.0;	1	1	1
4. for I=1 to n do	1	n+1	n+1
5. s=s+a[I];	1	n	n
6. return s;	1	1	1
7. }	0	-	0
Total			2n+3

Let f and g be two non-negative functions.

1. **Big 'O' notation:**

The function f(n) is said to be **Big oh** of g(n) i.e. f(n)=O(g(n)) iff there exists positive constants c and no such that f(n)≤c*g(n) for all n, n ≥ no.

Example:

If any program takes less time, then it is best writing program. It depends on algorithm efficiency. The efficiency of algorithm is measured using the O-notation, called **big-O** stands for '**Order of'**

If suppose consider, 1 + 2 + - - - - (n-1) = n(n-1) / 2

Then ½ n^2-n/2 ,delete constants

N^2-n

Don't consider smaller amounts. Then **O(n^2)**. We categorize the O-nation based on these dominant toms to create various classes of function to measure algorithms efficiency.

N	**O(1)**	**O(n)**	**O(log n)**	**O(n log n)**	**O(n^2)**	**O(n^3)**
1	1	1	1	1	1	1
2	1	2	1	2	4	8

4	1	4	2	8	16	64
8	1	8	3	24	64	512
16	1	16	4	64	256	4096

O(1) -- > An algorithm will run for a constant amount of time irrespective of the size of input it receives. Ex. Is push to stack, pop from stack.

O(n) - -> If the size of the input is large, the time taken to complete the process will increase.

Ex, find the sum of numbers in an array, of the array size increases, the time taken to complete the sum is also increases.

O(n^2) - -> The amount of time increases by n2 if you increase the size if the input by n.

Ex. Bubble, insertion sort.

O(2^n) - -> The amount of time the algorithm will take to complete doubles each time you increase the input.

An algorithm with O(logn) has the best efficiency.

2. Big-Omega Notation

Just as O-notation provides an asymptotic upper bound on a function, □ notation provides an asymptotic lower bound.
Formal Definition: A function f(n) is □(g (n)) if there exist constants c and k ∈ ℛ+
such that f(n) >=c. g(n) for all n>=k.

f(n)= □(g (n)) means that f(n) is greater than or equal to some constant multiple of g(n) for all values of n greater than or equal to some k.

Example*:* If f(n) =$n^{2,}$ then f(n)= □(n)

In simple terms, f(n)= □(g (n)) means that the growth rate of f(n) is greater that or equal to g(n).

3. Theta Notation

A function f (n) belongs to the set of □ (g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1.g(n) and c2.g(n), for sufficiently large values of n.

Formal Definition: A function f (n) is □ (g(n)) if it is both *O(g(n))* and □ *(g(n))*. In other words, there exist constants c1, c2, and k >0 such that c1.g (n)<=f(n)<=c2. g(n) for all n >= k

If f(n)= □ (g(n)), then g(n) is an asymptotically tight bound for f(n).
In simple terms, f(n)= □ (g(n)) means that f(n) and g(n) have the same rate of growth.

4. Little-o Notation

Big-Oh notation may or may not be asymptotically tight,
for example: 2n2 = O(n2)
=O(n3)

f(n)=o(g(n)) means for all c>0 there exists some k>0 such that f(n)<c.g(n) for all n>=k. Informally, f(n)=o(g(n)) means f(n) becomes insignificant relative to g(n) as n approaches infinity.
Example: f(n)=3n+4 is o(n2)

In simple terms, f(n) has less growth rate compared to g(n).
g(n)= 2n2 g(n) =o(n3), O(n2), g(n) is not o(n2).

5. Little-Omega (□ notation)

Little-omega (□) notation is to big-omega (□) notation as little-o notation is to Big-Oh notation.

We use □ notation to denote a lower bound that is not asymptotically tight.

Formal Definition: f(n)= □ (g(n)) if there exists a constant no>0 such that 0<= c. g(n)<f(n) for all n>=k.

Example: $2n^2$=□(n) but it's not □ (n^2).

Relational Properties of the Asymptotic Notations

Transitivity

- if f(n)=Θ(g(n)) and g(n)= Θ(h(n)) then f(n)=Θ(h(n)),
- if f(n)=O(g(n)) and g(n)= O(h(n)) then f(n)=O(h(n)),
- if f(n)=Ω(g(n)) and g(n)= Ω(h(n)) then f(n)=Ω (h(n)),
- if f(n)=o(g(n)) and g(n)= o(h(n)) then f(n)=o(h(n)), and
- if f(n)=ω (g(n)) and g(n)= ω(h(n)) then f(n)=ω (h(n)).

1.1.2. Analysis of Algorithm

1.2. Review of elementary Data Structures

1.2.1. Heaps

1.2.2. Hashing

1.2.3. Set Representation

1.2.3.1. UNION, FIND Operation

Chapter 2: Divide and Conquer (6hr)

2.1. The General Method of Divide and Conquer

- ✓ Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1<k<=n, yielding 'k' sub problems.
- ✓ These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.
- ✓ If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.
- ✓ Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.

- ✓ For those cases the re application of the divide-and-conquer principle is naturally expressed by a recursive algorithm.

Example:

1) Consider the case in which a=2 and b=2. Let T(1)=2 & f(n)=n.

   We have,

$$\begin{aligned} T(n) &= 2T(n/2)+n \\ &= 2[2T(n/2/2)+n/2]+n \\ &= [4T(n/4)+n]+n \\ &= 4T(n/4)+2n \\ &= 4[2T(n/4/2)+n/4]+2n \\ &= 4[2T(n/8)+n/4]+2n \\ &= 8T(n/8)+n+2n \\ &= 8T(n/8)+3n \end{aligned}$$

- In general, we see that $T(n)=2^iT(n/2^i)+in.$, for any log n >=i>=1.

→ $T(n) = 2^{\log n} T(n/2^{\log n}) + n \log n$

→Corresponding to the choice of i=logn

Thus, $T(n) = 2^{\log n} T(n/2^{\log n}) + n \log n$

$= n.\ T(n/n) + n \log n$

$= n.\ T(1) + n \log n$ [since, log 1=0, $2^0=1$]

$= 2n + n \log n$

2.2. Binary Search

BINARY SEARCH:

```
1.  Algorithm Bin search(a,n,x)
2.  // Given an array a[1:n] of elements in non-decreasing
3.  //order, n>=0,determine whether 'x' is present and
4.  // if so, return 'j' such that x=a[j]; else return 0.
5.  {
6.  low:=1; high:=n;
7.  while (low<=high) do
8.  {
9.        mid:=[(low+high)/2];
10.       if (x<a[mid]) then high;
11.       else if(x>a[mid]) then
                 low=mid+1;
12.     else return mid;
13.   }
14.   return 0;
15. }
```

- Algorithm, describes this binary search method, where Binsrch has 4I/ps a[], I , l & x.
- It is initially invoked as Binsrch (a,1,n,x)
- A non-recursive version of Binsrch is given below.
- This Binsearch has 3 i/ps a,n, & x.
- The while loop continues processing as long as there are more elements left to check.
- At the conclusion of the procedure 0 is returned if x is not present, or 'j' is returned, such that a[j]=x.
- We observe that low & high are integer Variables such that each time through the loop either x is found or low is increased by at least one or high is decreased at least one.
- Thus we have 2 sequences of integers approaching each other and eventually low becomes > than high & causes termination in a finite no. of steps if 'x' is not present.

Example:

1) Let us select the 14 entries.

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

→ Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of 'x'.

→ Only the variables, low, high & mid need to be traced as we simulate the algorithm.

→ We try the following values for x: 151, -14 and 9.

For this search there is 2 successful searches & 1 unsuccessful search.

Theorem: Algorithm Binsearch(a,n,x) works correctly.

Proof:

We assume that all statements work as expected and that comparisons such as x>a[mid] are appropriately carried out.

- Initially low =1, high= n,n>=0, and a[1]<=a[2]<=……..<=a[n].
- If n=0, the while loop is not entered and is returned.
- Otherwise we observe that each time thro' the loop the possible elements to be checked of or equality with x and a[low], a[low+1],……..,a[mid],……a[high].
- If x=a[mid], then the algorithm terminates successfully.
- Otherwise, the range is narrowed by either increasing low to (mid+1) or decreasing high to (mid-1).
- Clearly, this narrowing of the range does not affect the outcome of the search.
- If low becomes > than high, then 'x' is not present & hence the loop is exited.

2.3**. Finding Maximum and Minimum**

2.4. Merge Sort

- As another example divide-and-conquer, we investigate a sorting algorithm that has the nice property that is the worst case its complexity is O(n log n)
- This algorithm is called merge sort
- We assume throughout that the elements are to be sorted in non-decreasing order.
- Given a sequence of 'n' elements a[1],…,a[n] the general idea is to imagine then split into 2 sets a[1],…..,a[n/2] and a[[n/2]+1],….a[n].
- Each set is individually sorted, and the resulting sorted sequences are merged to produce a single sorted sequence of 'n' elements.
- Thus, we have another ideal example of the divide-and-conquer strategy in which the splitting is into 2 equal-sized sets & the combining operation is the merging of 2 sorted sets into one.

Algorithm for Merge Sort:

1. Algorithm MergeSort(low, high)
2. //a[low:high] is a global array to be sorted
3. //Small(P) is true if there is only one element

```
4.  //to sort. In this case the list is already sorted.
5.  {
6.  if (low<high) then //if there are more than one element
7.  {
8.  //Divide P into sub problems
9.  //find where to split the set
10. mid = [(low+high)/2];
11. //solve the sub problems.
12. mergesort (low,mid);
13. mergesort(mid+1,high);
14. //combine the solutions.
15. merge(low,mid,high);
16. }
17. }
```

- Consider the array of 10 elements a[1:10] =(310, 285, 179, 652, 351, 423, 861, 254, 450, 520)
- Algorithm Mergesort begins by splitting a[ ] into 2 sub arrays each of size five (a[1:5] and a[6:10]).
- The elements in a[1:5] are then split into 2 sub arrays of size 3 (a[1:3]) and 2 (a[4:5])
- Then the items in a a[1:3] are split into sub arrays of size 2 a[1:2] & one(a[3:3])
- The 2 values in a[1:2} are split to find time into one-element sub arrays, and now the merging begins.

Example:

(310| 285| 179| 652, 351| 423, 861, 254, 450, 520)

→ At this point there are 2 sorted sub arrays & the final merge produces the fully sorted result.

(179, 254, 285, 310, 351, 423, 450, 520, 652, 861)

- *If the time for the merging operations is proportional to 'n', then the computing time for merge sort is described by the recurrence relation.*

$$T(n) = \begin{cases} a & n=1, \text{'a' a constant} \\ 2T(n/2)+cn & n>1, \text{'c' a constant.} \end{cases}$$

→ When 'n' is a power of 2, n= 2^k, we can solve this equation by successive substitution.

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

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

$= 4(2T(n/8)+cn/4)+2cn$

*

*

$= 2^k\, T(1)+kCn.$

$= an + cn \log n.$

→ It is easy to see that if $s^k<n<=2^k+1$, then $T(n)<=T(2^k+1)$. Therefore,

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

2.5. **Quick Sort**

- The divide-and-conquer approach can be used to arrive at an efficient sorting method different from merge sort.
- In merge sort, the file a[1:n] was divided at its midpoint into sub arrays which were independently sorted & later merged.
- In Quick sort, the division into 2 sub arrays is made so that the sorted sub arrays do not need to be merged later.
- This is accomplished by rearranging the elements in a[1:n] such that a[I]<=a[j] for all I between 1 & n and all j between (m+1) & n for some m, 1<=m<=n.
- Thus the elements in a[1:m] & a[m+1:n] can be independently sorted.
- No merge is needed. This rearranging is referred to as partitioning.
- Function partition of Algorithm accomplishes an in-place partitioning of the elements of a[m:p-1]
- It is assumed that a[p]>=a[m] and that a[m] is the partitioning element. If m=1 & p-1=n, then a[n+1] must be defined and must be greater than or equal to all elements in a[1:n]

- The assumption that a[m] is the partition element is merely for convenience, other choices for the partitioning element than the first item in the set are better in practice.
- The function interchange (a,I,j) exchanges a[I] with a[j].

```
Algorithm Quicksort(p,q)
//Sort the elements a[p],….a[q] which resides
//is the global array a[1:n] into ascending
//order; a[n+1] is considered to be defined
// and must be >= all the elements in a[1:n]
{
if(p<q) then // If there are more than one element
{
// divide p into 2 subproblems
j=partition(a,p,q+1);
//'j' is the position of the partitioning element.
//solve the subproblems.
quicksort(p,j-1);
quicksort(j+1,q);
//There is no need for combining solution.
}    }
```

2.6. Selection Sort

Chapter 3: Greedy Algorithms

3.1. General Characteristic of Greedy Algorithms

- Greedy method is the most straightforward design technique for finding the solution to a given problem.
- As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future.

DEFINITION:

- A problem with N inputs will have some constraints and any subsets that satisfy these constraints are called a feasible solution.
- A feasible solution that either maximizes or minimizes a given objective function is called an optimal solution.
- For an algorithm that uses greedy method works in stages, considering one input at a time. Based on this input, it is decided whether the particular input gives the optimal solution or not.

Control abstraction for Greedy Method:

```
1. Algorithm Greedy (a, n)
2.//a[1:n] contain the 'n' inputs
3. {
4. Solution =0; //Initialize the solution.
5. For i=1 to n do
6. {
7. x=Select (a);
8. if (Feasible (Solution, x))then
9.        Solution =union(Solution, x);
10.}
11. return solution;
12.}
```

- The function Select selects an input from a[] and removes it. The selected input's value is assigned to X.
- Feasible is a Boolean value function that determines whether X can be included into the solution vector.
- The function Union combines X with The solution and updates the objective function.
- The function Greedy describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the function subset, feasible & union are properly implemented.

3.2. Graph Minimum Spanning Tree (MST) - Kruskal's and Prims's Algorithms

- Let G(V,E) be an undirected connected graph with vertices 'v' and edge 'E'.

- A sub-graph t=(V,E’) of the G is a Spanning tree of G iff ‘t’ is a tree.
- The problem is to generate a graph G’= (V, E) where ‘E’ is the subset of E, G’ is a Minimum cost spanning tree.
- Each and every edge will contain the given non-negative length .Connect all the nodes with edge present in set E’ and weight has to be minimum.

NOTE:

- We have to visit all the nodes.
- The subset tree (i.e.) any connected graph with ‘N’ vertices must have at least N-1 edges and also it does not form a cycle.

Definition:

- A spanning tree of a graph is an undirected tree consisting of only those edges that are necessary to connect all the vertices in the original graph.
- A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Application of the spanning tree:

1. Analysis of electrical circuit.
2. Shortest route problems.

Minimum cost spanning tree:

- The cost of a spanning tree is the sum of cost of the edges in that tree.
- There are 2 method to determine a minimum cost spanning tree are

1. Kruskal’s Algorithm
2. Prom’s Algorithm.

KRUSKAL’S ALGORITHM:

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

- In this algorithm, a minimum cost-spanning tree ‘T’ is built edge by edge.
- Edges are considered for inclusion in ‘T’ in increasing order of their cost.

- An edge is included in 'T' if it doesn't form a cycle with edge already in T.
- To find the minimum cost spanning tree the edges are inserted to tree in increasing
  Order of their cost

Algorithm:

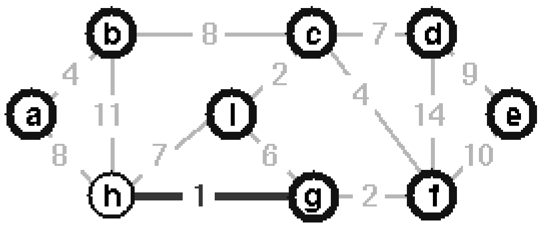
```
Algorithm kruskal(E,cost,n,t)
//E→set of edges in G that has 'n' vertices.
//cost[u,v]→cost of edge (u,v).t→set of edge in minimum cost spanning tree
// the final cost is returned.
{
for i=1 to n do parent[i]= -1;
i=0; mincost=0.0;
While((i<n-1)and (heap not empty)) do
{
        j=find(n);
        k=find(v);
        if(j not equal k) than
        {
                i=i+1
                t[i,1]=u;
                t[i,2]=v;
                mincost= mincost + cost[u,v];
                union(j,k);
        }
 }
if(i not equal to n-1) then write("No spanning tree")
else return minimum cost;
}
```

Analysis

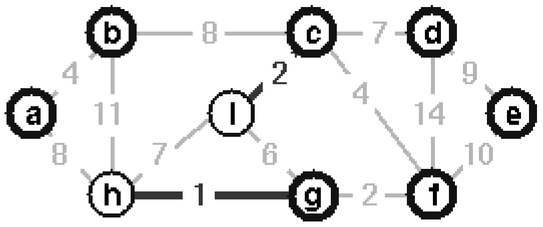
- The time complexity of minimum cost spanning tree algorithm in worst case is O(|E|log|E|), where E is the edge set of G.

Example: Step by Step operation of Kurskal's algorithm.

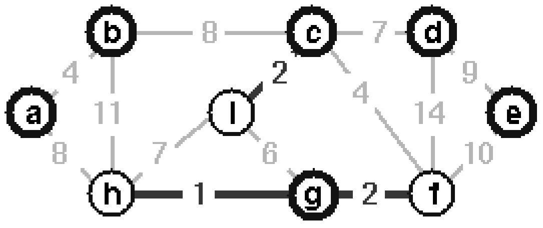
Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be representative. Let's choose vertex g arbitrarily.



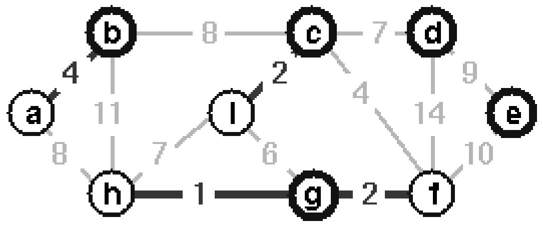
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



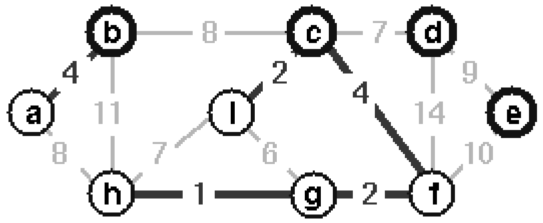
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



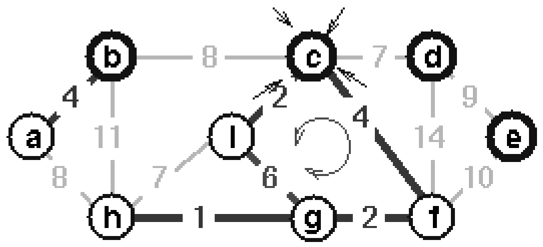
Step 4. Edge (a, b) creates a third tree.



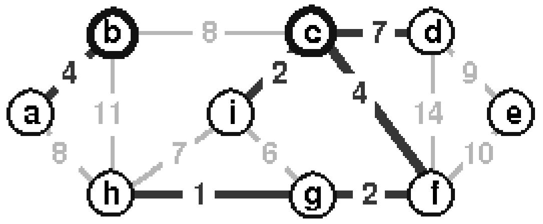
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



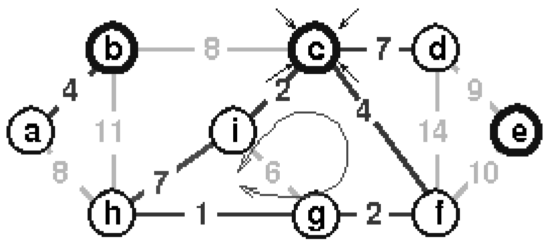
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



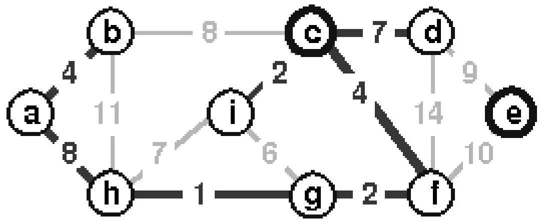
Step 7. Instead, add edge (c, d).



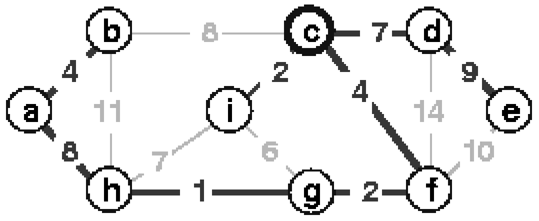
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).

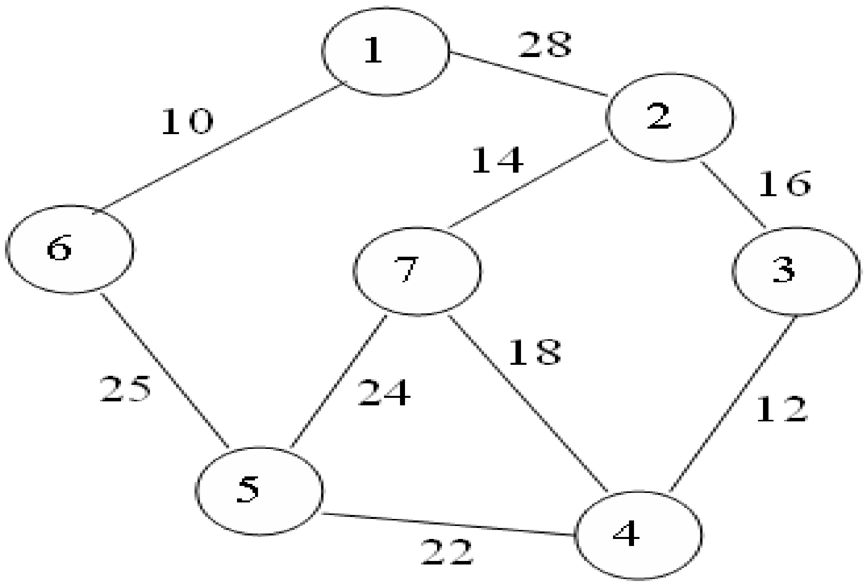


Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



PRIM'S ALGORITHM

Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have been reached.



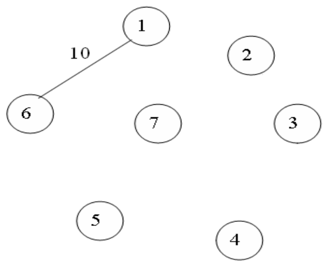
```
Algorithm Prim(E, cost, n, t)

{
  Let (k,l) be an edge of minimum cost in E;
  Mincost: =cost[k, l];
  t[1,1]:=k; t[1,2]:=l;
  for i:=1 to n do
          If (cost[i, l]<cost[i, k]) then near[i]:=l;
          else near[i]:=k;
 near[k]:=near[l]:=0;
 for i:=2 to n-1 do
  {
          Let j be an index such that near[j]≠0 and
          Cost[j, near[j]] is minimum;
          t[i,1]:=j; t[i,2]:=near[j];
          Mincost:=mincost+ Cost[j, near[j]];
          near[j]:=0;
          for k:=1 to n do
                  If ((near[k]≠0) and (Cost[k, near[k]]>cost[k ,j])) then
```

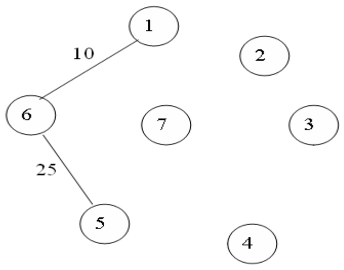
```
                              near[k]:=j;
 }
return mincost;
}
```

- The prims algorithm will start with a tree that includes only a minimum cost edge of G.
- Then, edges are added to the tree one by one, the next edge (i,j) to be added in such that i is a vertex included in the tree, j is a vertex not yet included, and cost of (i,j), cost[i, j] is minimum among all the edges.
- The working of prims will be explained by following diagram

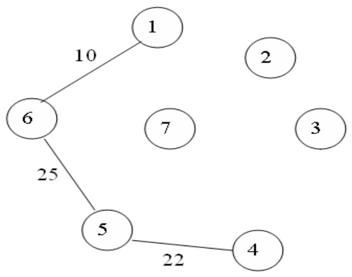
Step 1:



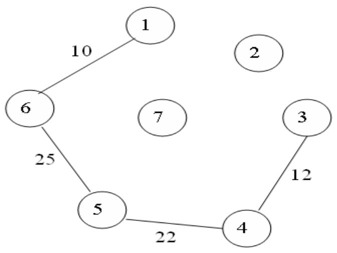
Step 2:



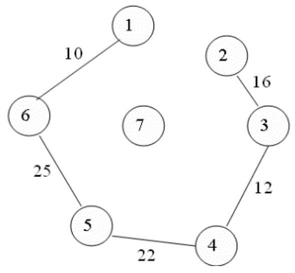
Step 3:

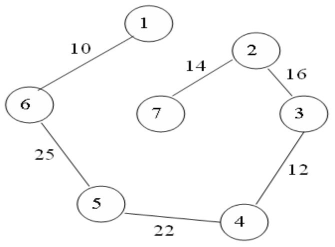


Step 4:



Step 5: **Step 6:**





3.3. Shortest Paths

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:

1. Is there a path from A to B?
2. If there is more than one path from A to B? Which is the shortest path?

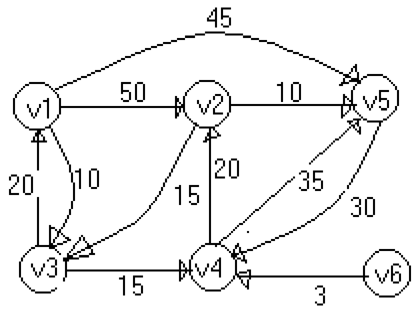


Fig 7.1

The problems defined by these questions are special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination. The graphs are digraphs representing streets. Consider a digraph G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:

Consider the digraph of fig 7-1. Let the numbers on the edges be the costs of travelling along that route. If a person is interested to travel from v1 to v2, then he encounters many paths. Some of them are

1. v1 v2 = 50 units
2. v1 v3 v4 v2 = 10+15+20=45 units
3. v1 v5 v4 v2 = 45+30+20= 95 units
4. v1 v3 v4 v5 v4 v2 = 10+15+35+30+20=110 units

The cheapest path among these is the path along v1 v3 v4 v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1 v2 that costs 50 units. One can also notice that, it is not possible to travel to v6 from any other node.

To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.

A much simpler method would be to solve it using matrix representation. The steps that should be followed is as follows,

Step 1: find the adjacency matrix for the given graph. The adjacency matrix for fig 7.1 is given below

	V1	V2	V3	V4	V5	V6
V1	-	50	10	Inf	45	Inf
V2	Inf	-	15	Inf	10	Inf

V3	20	Inf	-	15	inf	Inf
V4	Inf	20	Inf	-	35	Inf
V5	Inf	Inf	Inf	30	-	Inf
V6	Inf	Inf	Inf	3	Inf	-

Step 2: consider v1 to be the source and choose the minimum entry in the row v1. In the above table the minimum in row v1 is 10.

Step 3: find out the column in which the minimum is present, for the above example it is column v3. Hence, this is the node that has to be next visited.

Step 4: compute a matrix by eliminating v1 and v3 columns. Initially retain only row v1. The second row is computed by adding 10 to all values of row v3.

The resulting matrix is

	V2	V4	V5	V6
V1□ Vw	50	Inf	45	Inf
V1□ V3□ Vw	10+inf	10+15	10+inf	10+inf
Minimum	50	25	45	inf

Step 5: find the minimum in each column. Now select the minimum from the resulting row. In the above example the minimum is 25. Repeat step 3 followed by step 4 till all vertices are covered or single column is left.

The solution for the fig 7.1 can be continued as follows

	V2	V5	V6
V1□ Vw	50	45	Inf

V1□ V3□ V4□ Vw	25+20	25+35	25+inf
Minimum	45	45	Inf

	V5	V6
V1□ Vw	45	Inf
V1□ V3□ V4□ V2□ Vw	45+10	45+inf
Minimum	45	inf

	V6
V1□ Vw	Inf
V1□ V3□ V4□ V2□ V5□ Vw	45+inf
Minimum	inf

Finally the cheapest path from v1 to all other vertices is given by V1□ V3□ V4□ V2□ V5.

```
Algorithm Shortest Paths(v, cost, dist, n)
//dist[j],1<=j<=n is set to the length of the shortest path from vertex v to vertex j
// in a digraph G with n vertices
//dist[v] is set to zero. Cost adjacency matrix is cost[1:n,1:n]
{
  for i:=1 to n do
  { // initialize S
        S[i]:=false;
        dist[i]:=cost[v,i];
  }
  S[v]:=true;
12.  dist[v]:=0.0; // put v in S
13.  for num:=2 to n-1 do
14.  {
15.           //Determine n-1 paths from v
16.          Choose u from among those vertices not in S such that dist[u] is minimum;
17.           S[u]:=true;  //put u in S
18.           for(each w adjacent to u with s[w]=false) do
19.           {//Update distances
20.                   If(dist[w]>dist[u]+cost[u, w]) then
21.                           dist[w]=dist[u]+cost[u, w];
22.           }
23.  } }
```

The analysis of the algorithm is as follows

- The for loop (line 6) executes n+1 times therefore O(n).
- The for loop in line 13 is executed n-2 times. Each execution of this loop requires O(n) time to select the vertex at line 16 & 17.
- The inner most for loop at line 18 takes O(n).
- So the for loop of line 13 takes O(n^2).
    - ❖ The time complexity of the algorithm is **O(n^2)**.

3.4. Scheduling

We have given a set of n jobs. Each job i is associated with a profit Pi(>0) and a deadline di(>=0). We have to find a sequence of job, which will be completed within its deadlines, and it should yield a maximum profit.

Points To remember:

- To complete a job, one has to process the job for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by this deadline.
- The value of a feasible solution J is the sum of the profits of the jobs in J, or
  $\sum_{iEj}$ Pi.

→Since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.

→So the waiting time and the processing time should be less than or equal to the dead line of the job.

ALGORITHM:

```
Algorithm JS(d,j,n)
//The job are ordered such that p[1]>p[2]…>p[n]
//j[i]  is  the ith job in the optimal solution
// Also at terminal d [J[ i]]<=d[J[i+1]],1<i<k
 {
  d[0]= J[0]=0;
  J[1]=1;
  k=1;
  for i =1 to n do
 {
             // consider jobs in non increasing order of P[i]; find the position for i and check
            //  the feasibility of insertion
            r=k;
            while((d[J[r]]>d[i] )and (d[J[r]] !=  r) do r =r-1;
            if (d[J[r]]<d[i])and (d[i]>r))then
            {
                        for q=k to (r+1) step –1 do     J [q+1]=j[q]
                        J[r+1]=i;
                        k=k+1;
            }
}
return k;
}
```

Example :

1. n=5 (P1,P2,…P5)=(20,15,10,5,1)
   (d1,d2….d3)=(2,2,1,3,3)

Feasible solution	*Processing Sequence*	*Value*
(1)	(1)	20
(2)	(2)	15
(3)	(3)	10
(4)	(4)	5
(5)	(5)	1
(1,2)	(2,1)	35
(1,3)	(3,1)	30
(1,4)	(1,4)	25
(1,5)	(1,5)	21
(2,3)	(3,2)	25
(2,4)	(2,4)	20
(2,5)	(2,5)	16
(1,2,3)	(3,2,1)	45
(1,2,4)	(1,2,4)	40

The Solution 13 is optimal

2. n=4 (P1,P2,…P4)=(100,10,15,27)
   (d1,d2….d4)=(2,1,2,1)

Feasible solution	*Processing Sequence*	*Value*
(1,2)	(2,1)	110
(1,3)	(1,3)	115
(1,4)	(4,1)	127

(2,3)	(9,3)	25
(2,4)	(4,2)	37
(3,4)	(4,3)	42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

The solution 3 is optimal.

Chapter 4: Dynamic Programming

4.1. Introduction to Dynamic Programming

Dynamic programming is a method in which the solution to a given problem can be obtained by making a sequence of decisions. The idea of dynamic programming is quite simple: avoid calculating the same thing twice, usually by keeping a table of known result that fills up a sub instances are solved.

Divide and conquer is a top-down method. When a problem is solved by divide and conquer, we immediately attack the complete instance, which we then divide into smaller and smaller sub-instances as the algorithm progresses.

Dynamic programming on the other hand is a bottom-up technique. We usually start with the smallest and hence the simplest sub- instances. By combining their solutions, we obtain the answers to sub-instances of increasing size, until finally we arrive at the solution of the original instances. The essential difference between the greedy method and dynamic programming is that in greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing sub-optimal sub-sequences cannot be optimal and so will not be generated.

4.2. All pairs Shortest Path - Floyd-Warshall Algorithm

- Let G=<N,A> be a directed graph 'N' is a set of nodes and 'A' is the set of edges.
- Each edge has an associated non-negative length.
- We want to calculate the length of the shortest path between each pair of nodes.

- Suppose the nodes of G are numbered from 1 to n, so N={1,2,...N},and suppose G matrix L gives the length of each edge, with L(i,j)=0 for i=1,2...n,L(i,j)>=for all i & j, and L(i,j)=infinity, if the edge (i,j) does not exist.
- The principle of optimality applies: if k is the node on the shortest path from i to j then the part of the path from i to k and the part from k to j must also be optimal, that is shorter.
- First, create a cost adjacency matrix for the given graph.
- Copy the above matrix-to-matrix D, which will give the direct distance between nodes.
- We have to perform N iteration after iteration k.the matrix D will give you the distance between nodes with only (1,2...,k)as intermediate nodes.
- At the iteration k, we have to check for each pair of nodes (i,j) whether or not there exists a path from i to j passing through node k.

4.3. Shortest Path - Dijkstra Algorithm

4.4. 0/1 Knapsack

- This problem is similar to ordinary knapsack problem but we may not take a fraction of an object.
- We are given ‘ N ‘ object with weight W_i and profits P_i where I varies from l to N and also a knapsack with capacity ‘ M ‘.
- The problem is, we have to fill the bag with the help of ‘ N ‘ objects and the resulting profit has to be maximum
- Formally, the problem can be started as, maximize $\sum_{i=1}^{n} X_i P_i$

subject to $\sum_{i=l}^{n} X_i W_i$ L M

- Where X_i are constraints on the solution $X_i \in \{0,1\}$. (u) X_i is required to be 0 or 1. if the object is selected then the unit in 1. if the object is rejected than the unit is 0. That is why it is called as 0/1, knapsack problem.
- To solve the problem by dynamic programming we up a table T[1…N, 0…M] (ic) the size is N. where ‘N’ is the no. of objects and column starts with ‘O’ to capacity (ic) ‘M’.

In the table T[i,j] will be the maximum valve of the objects i varies from 1 to n and j varies from O to M.

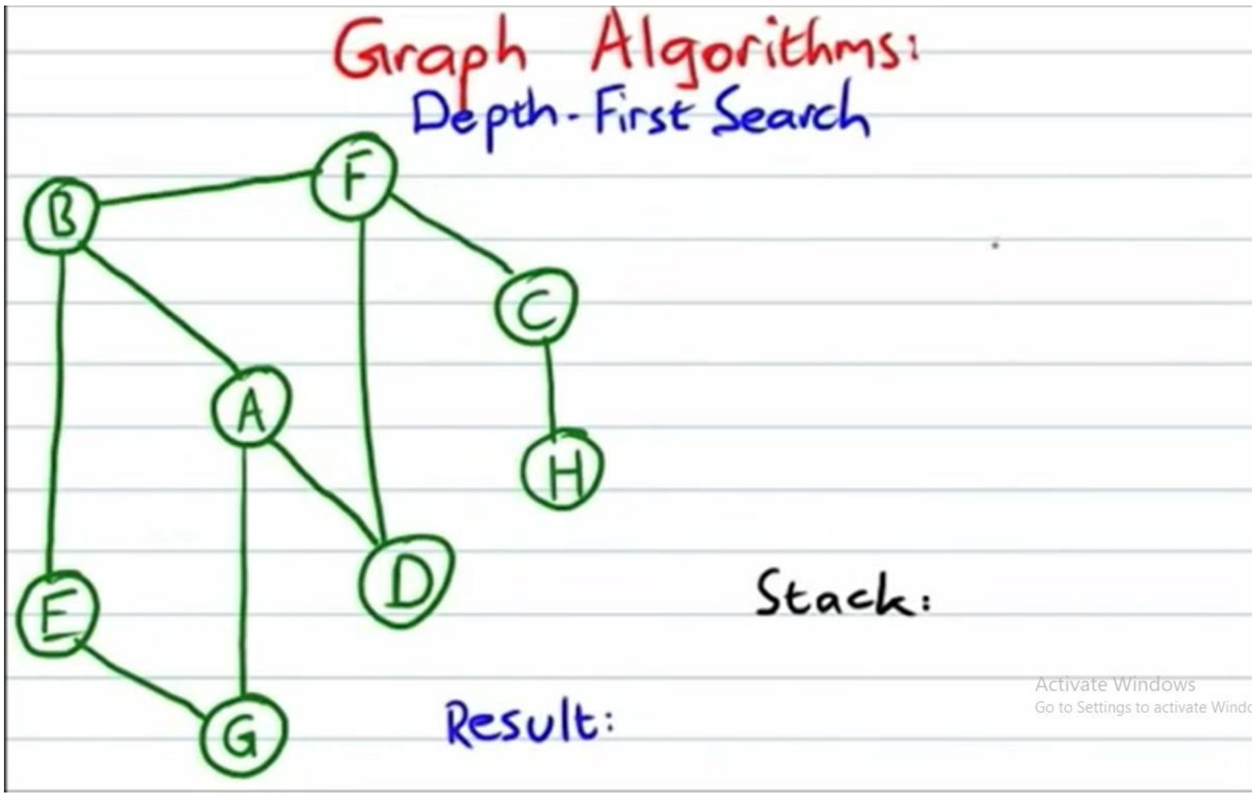
4.4. Depth First Search

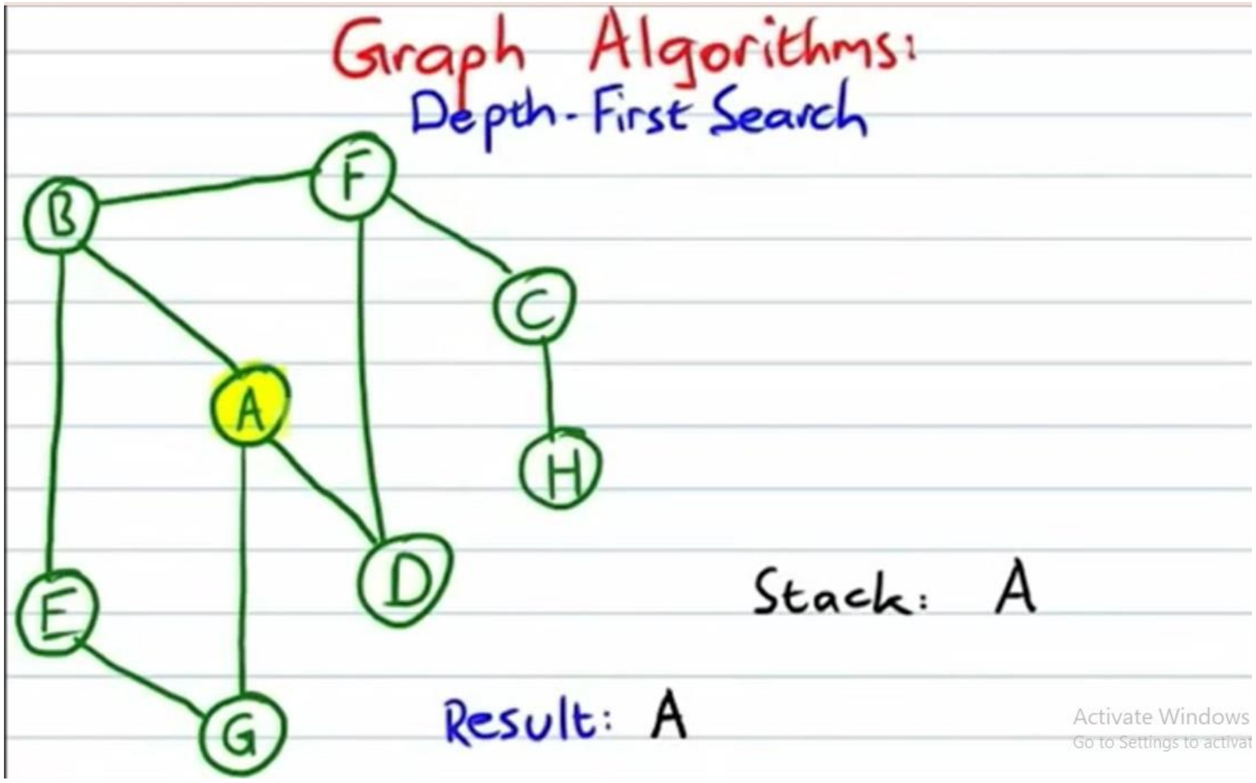
A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of u continues. The search terminates when all reached vertices have been fully explored. This search process is best-described recursively.

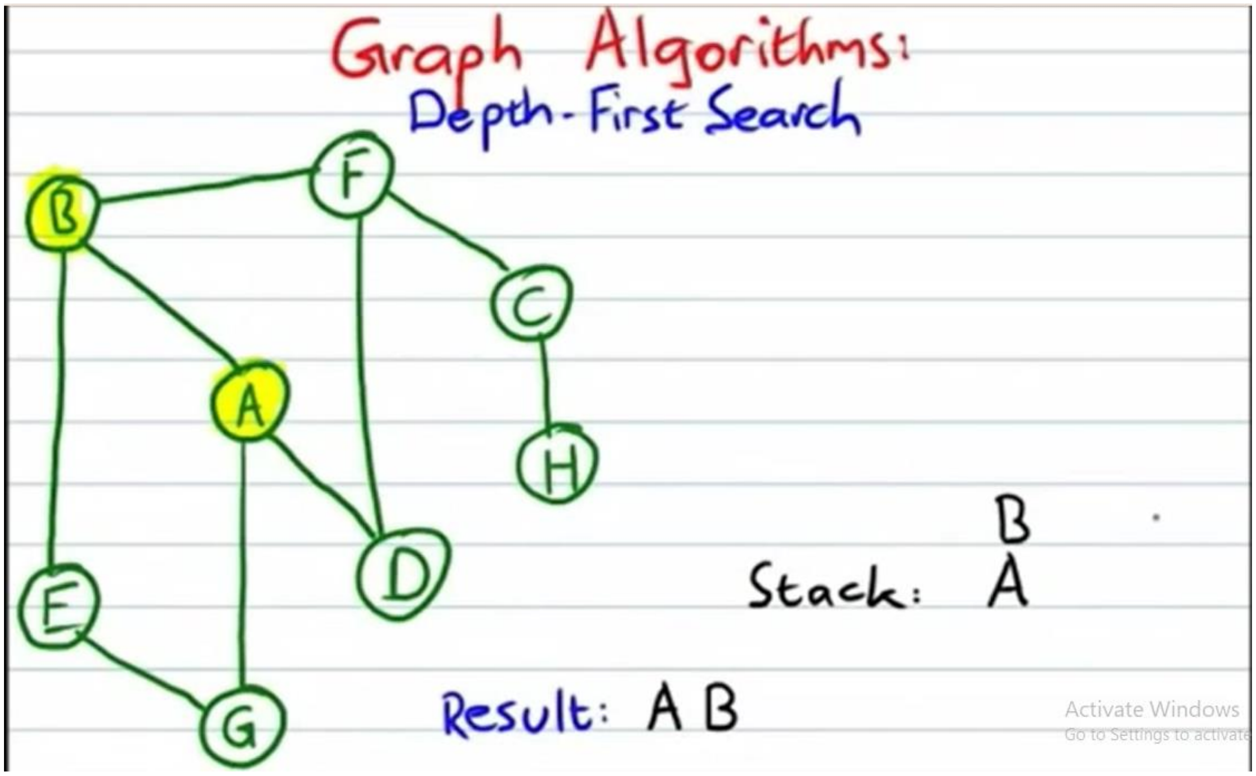
ALGORITHM: Depth First Search

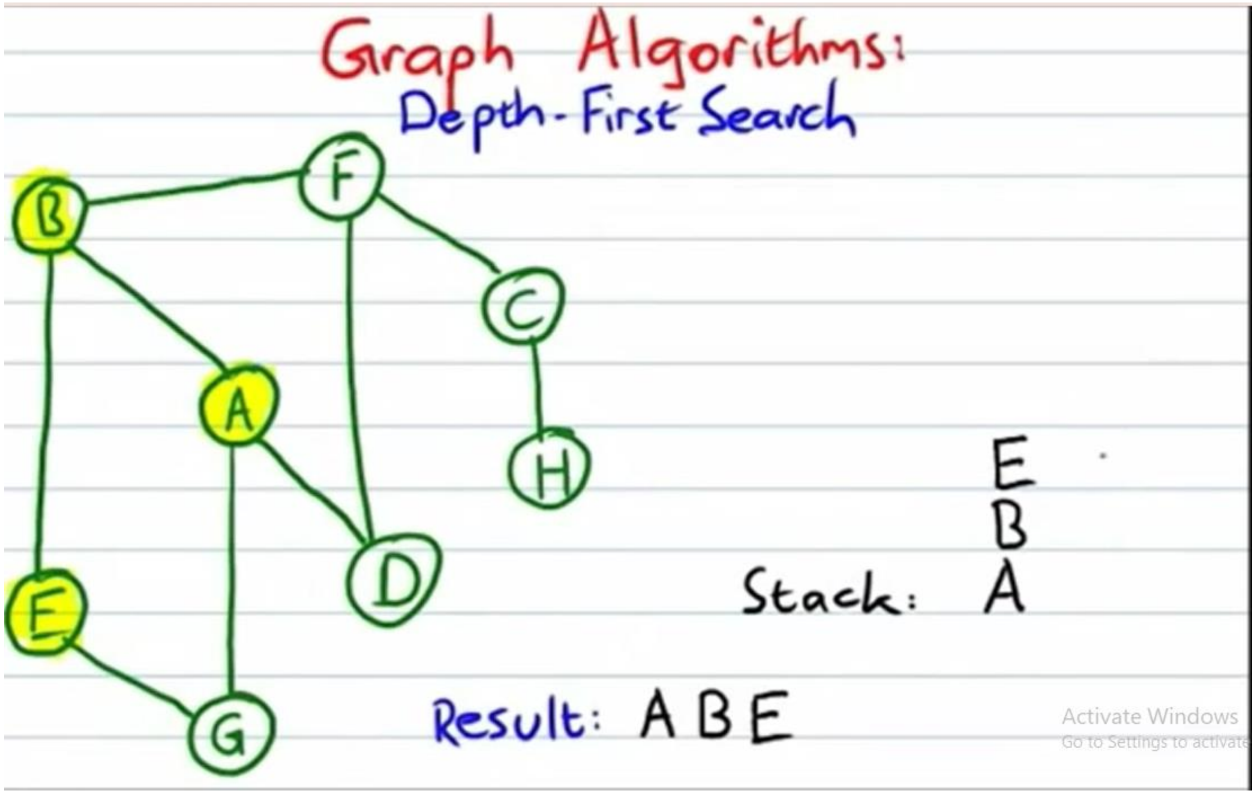
```
Algorithm DFS(v)
{
visited[v]=1
for each vertex w adjacent from v do
{
If (visited[w]=0)then
DFS(w);}}
```

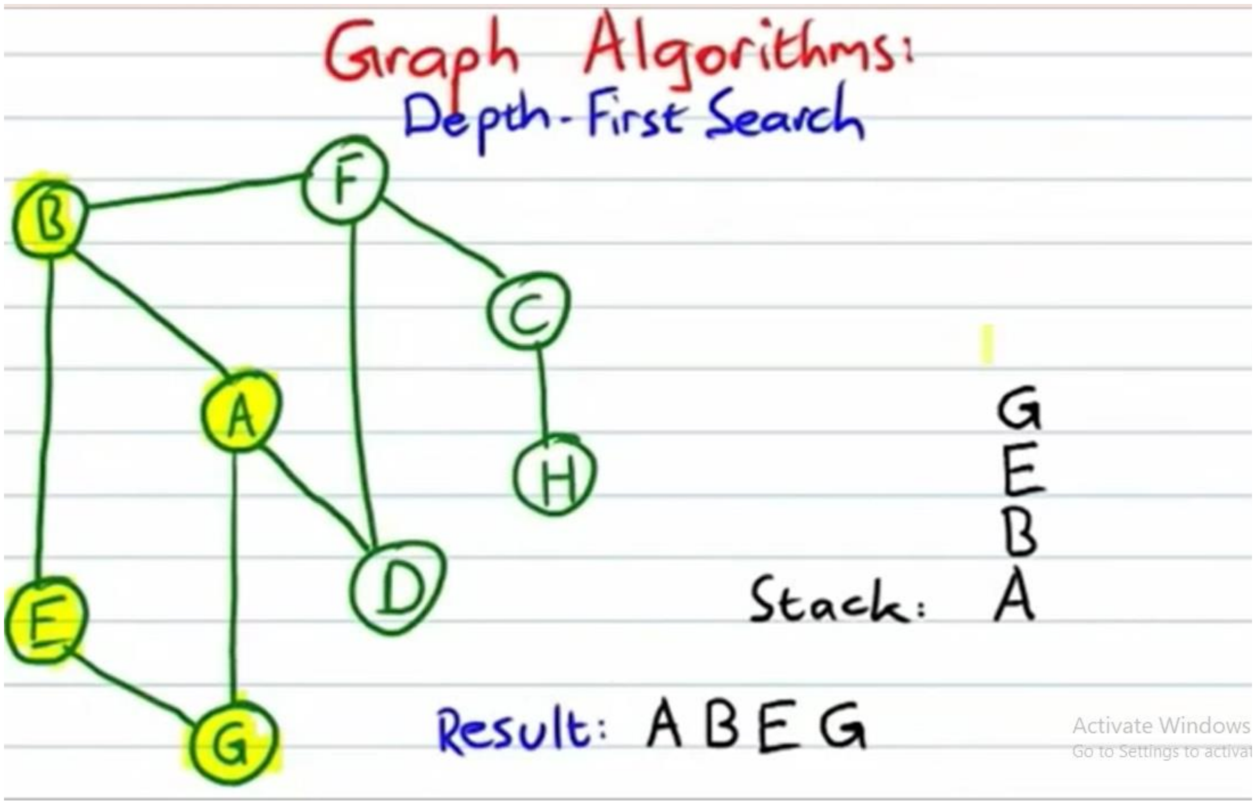
Example


Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
Stack:
Result:


Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
Stack: A
Result: A

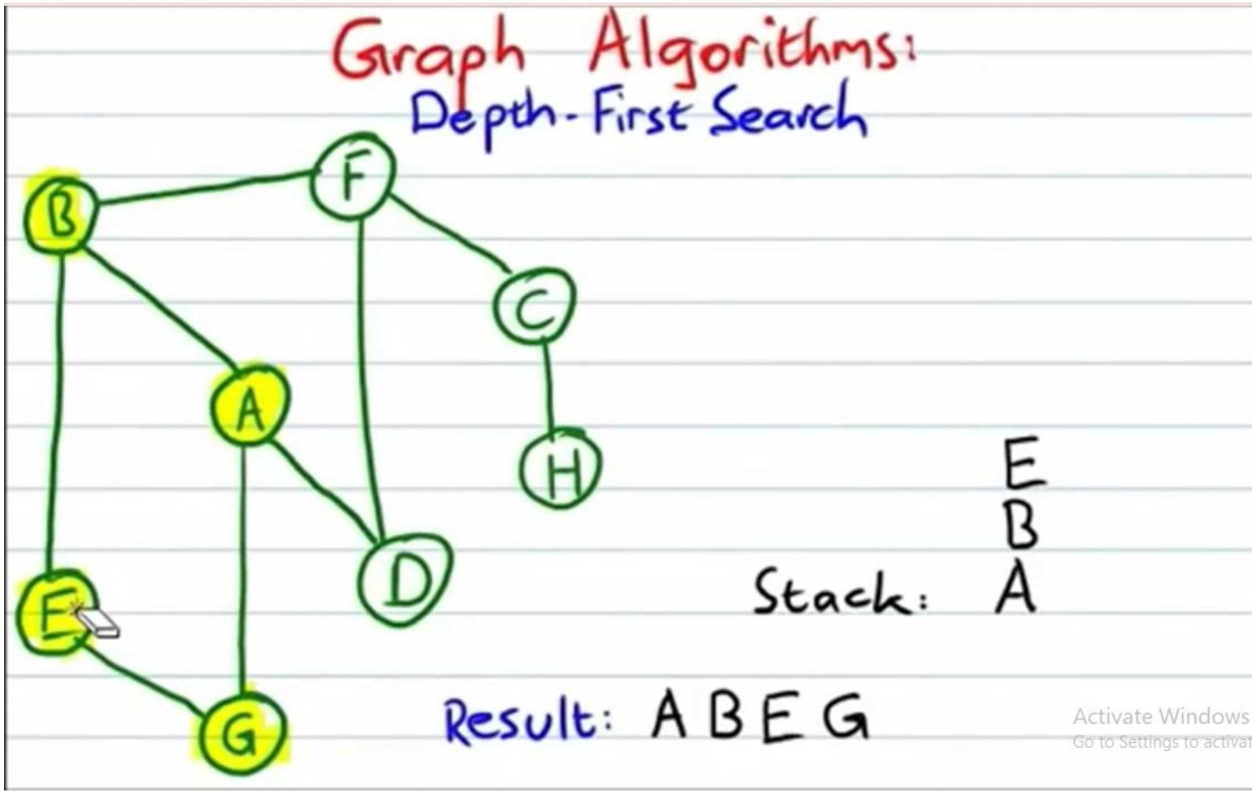

Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
Stack: B A
Result: A B
Activate Windows
Go to Settings to activate


Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
Stack: E B A
Result: A B E
Activate Windows
Go to Settings to activate



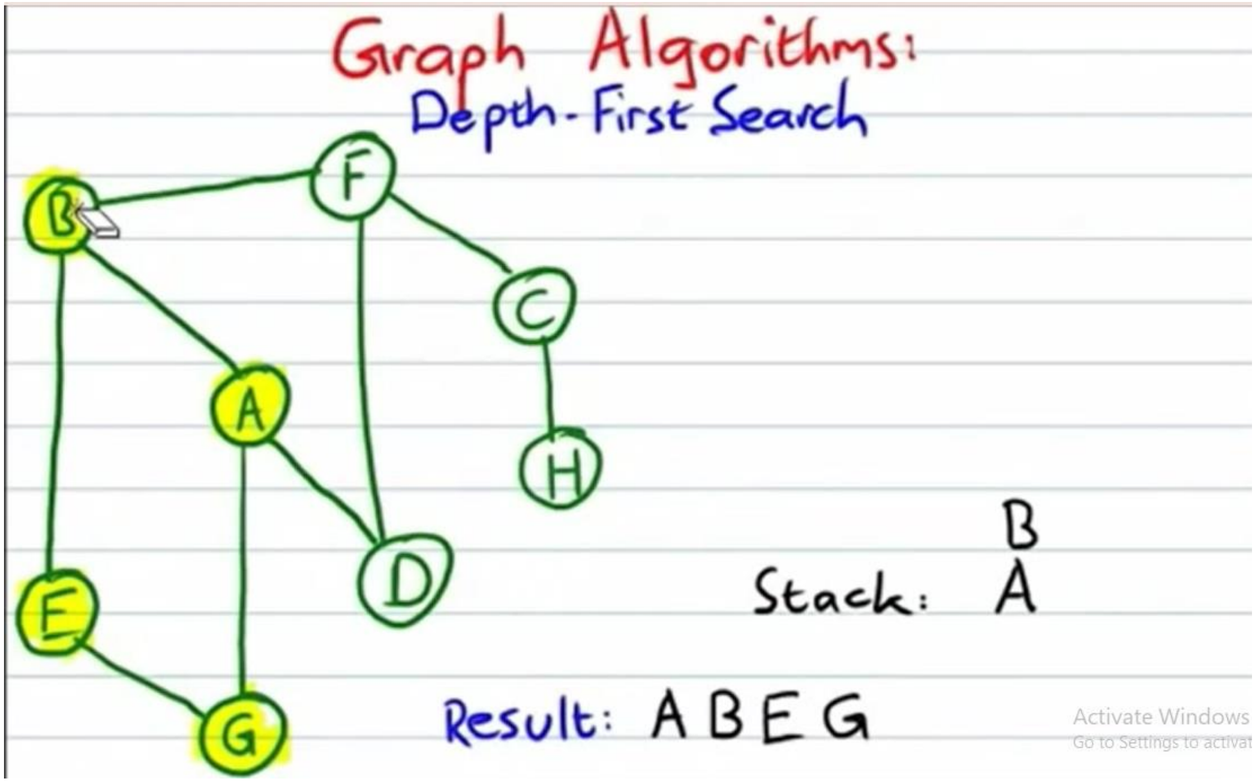
Stack: A B E G

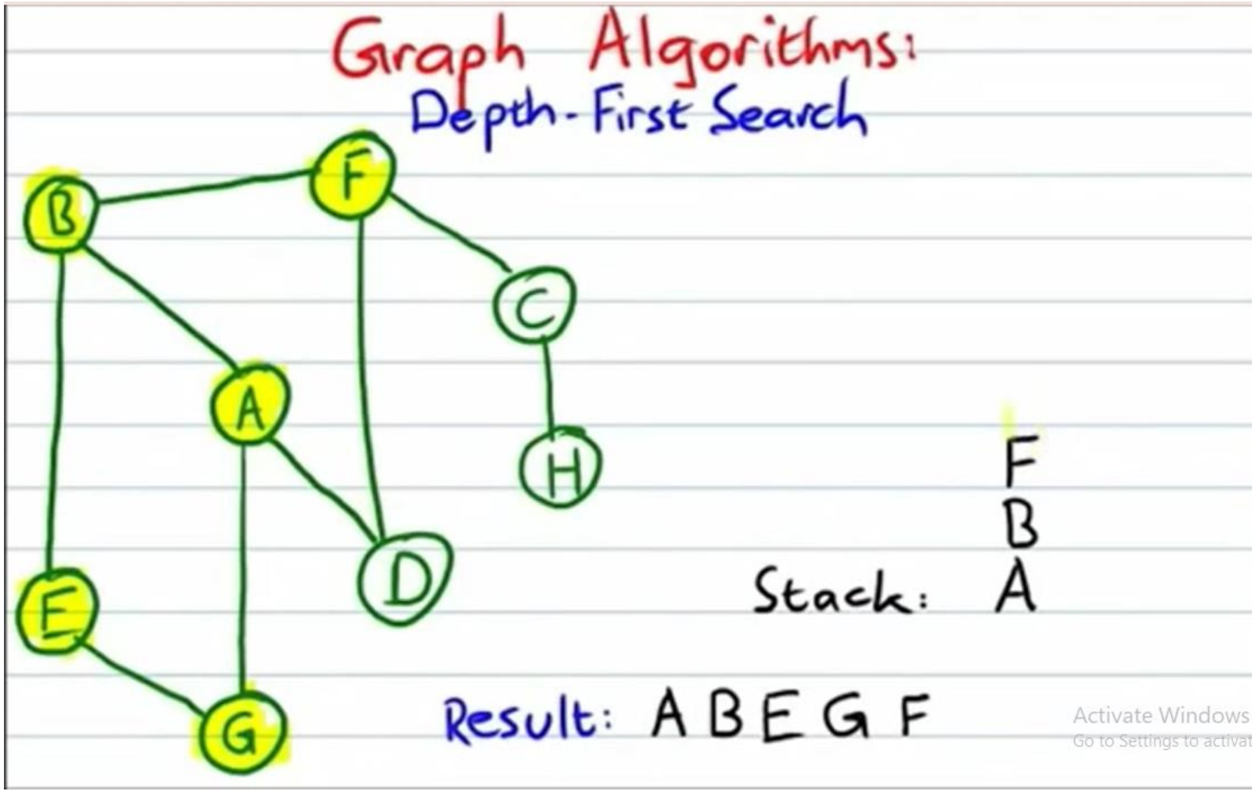
Result: A B E G

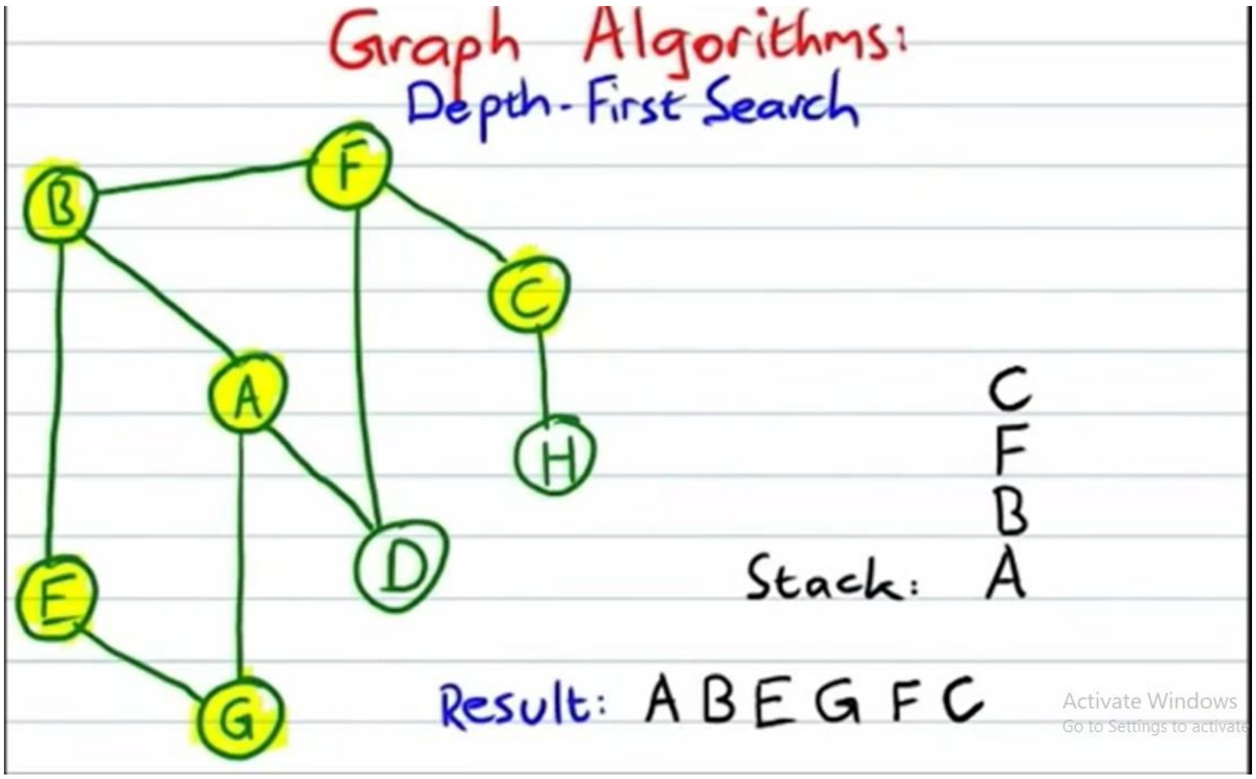


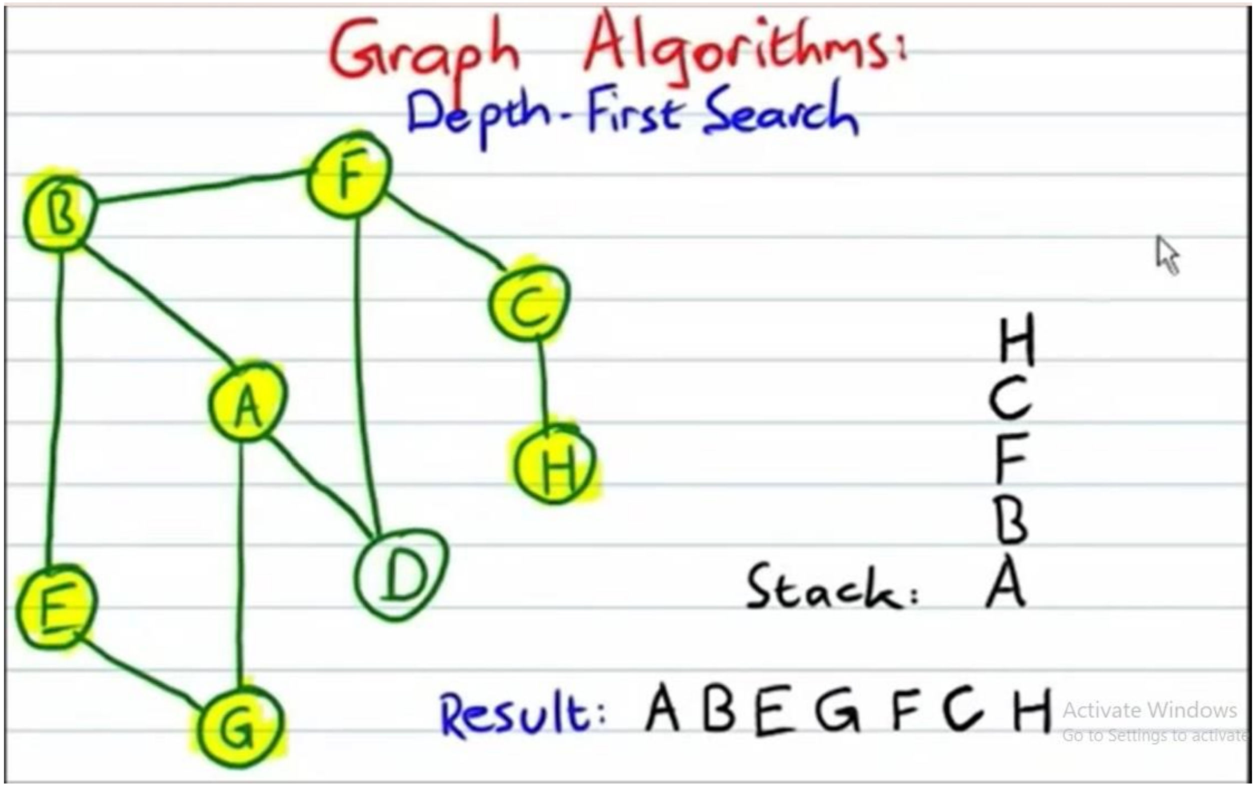
Stack: A B E

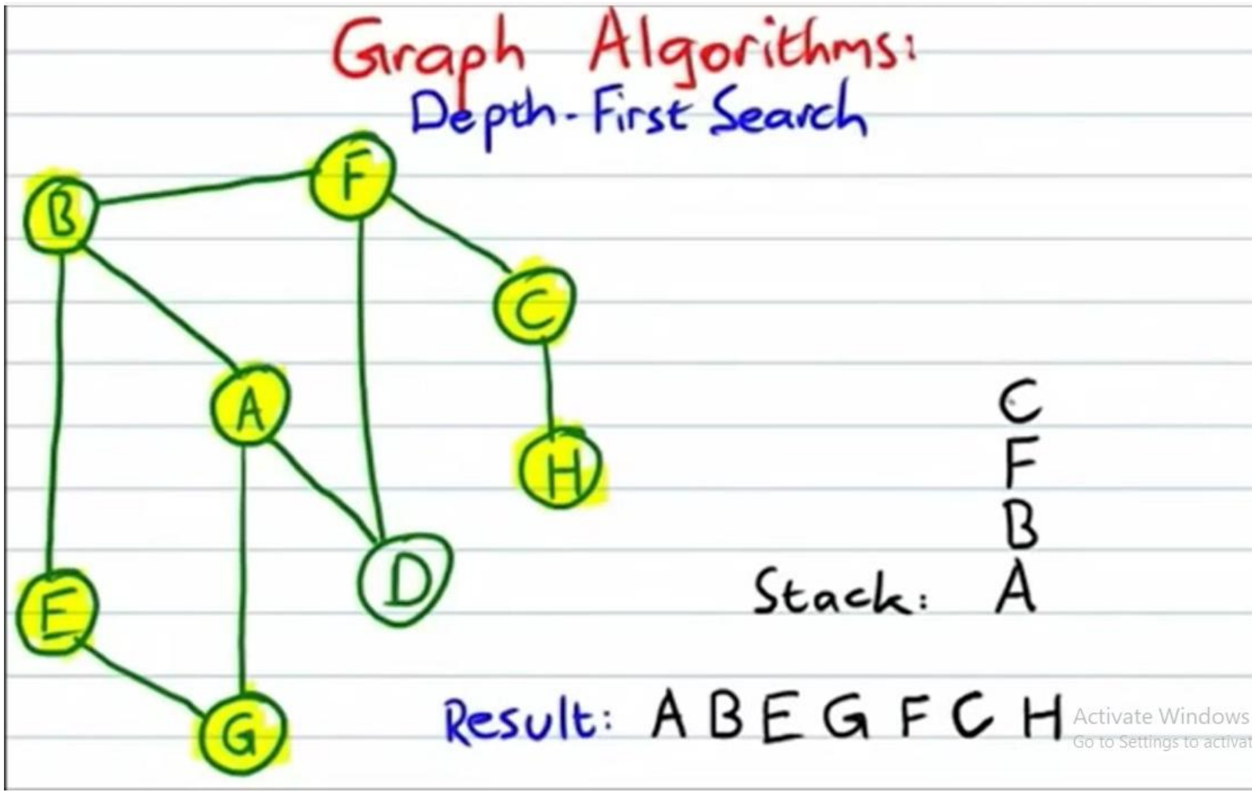
Result: A B E G


Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
B
Stack: A
Result: A B E G
Activate Windows
Go to Settings to activat


Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
F
B
Stack: A
Result: A B E G F
Activate Windows
Go to Settings to activat


Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
Stack: A B F C
Result: A B E G F C
Activate Windows
Go to Settings to activate

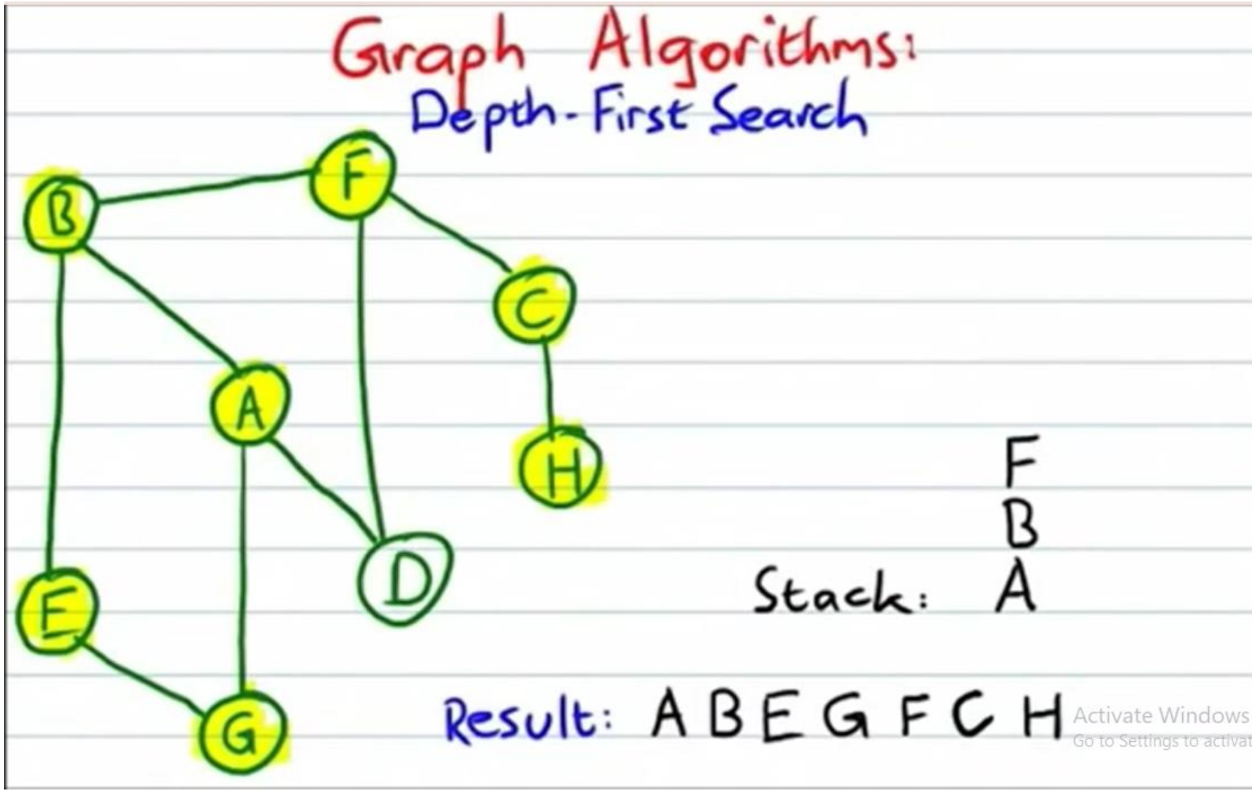

Graph Algorithms:
Depth-First Search
B
F
C
A
H
D
E
G
Stack: A B F C H
Result: A B E G F C H
Activate Windows
Go to Settings to activate



Graph Algorithms:
Depth-First Search

Stack: A B F C

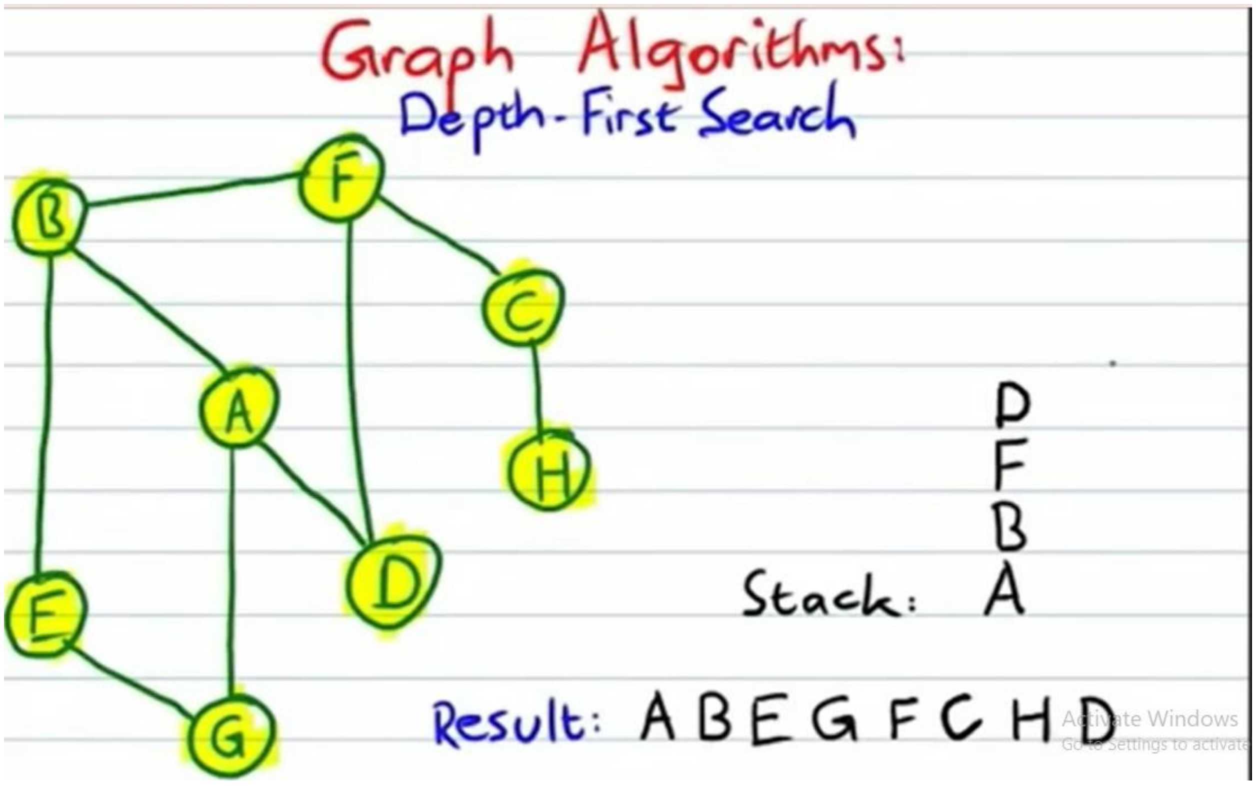
Result: A B E G F C H

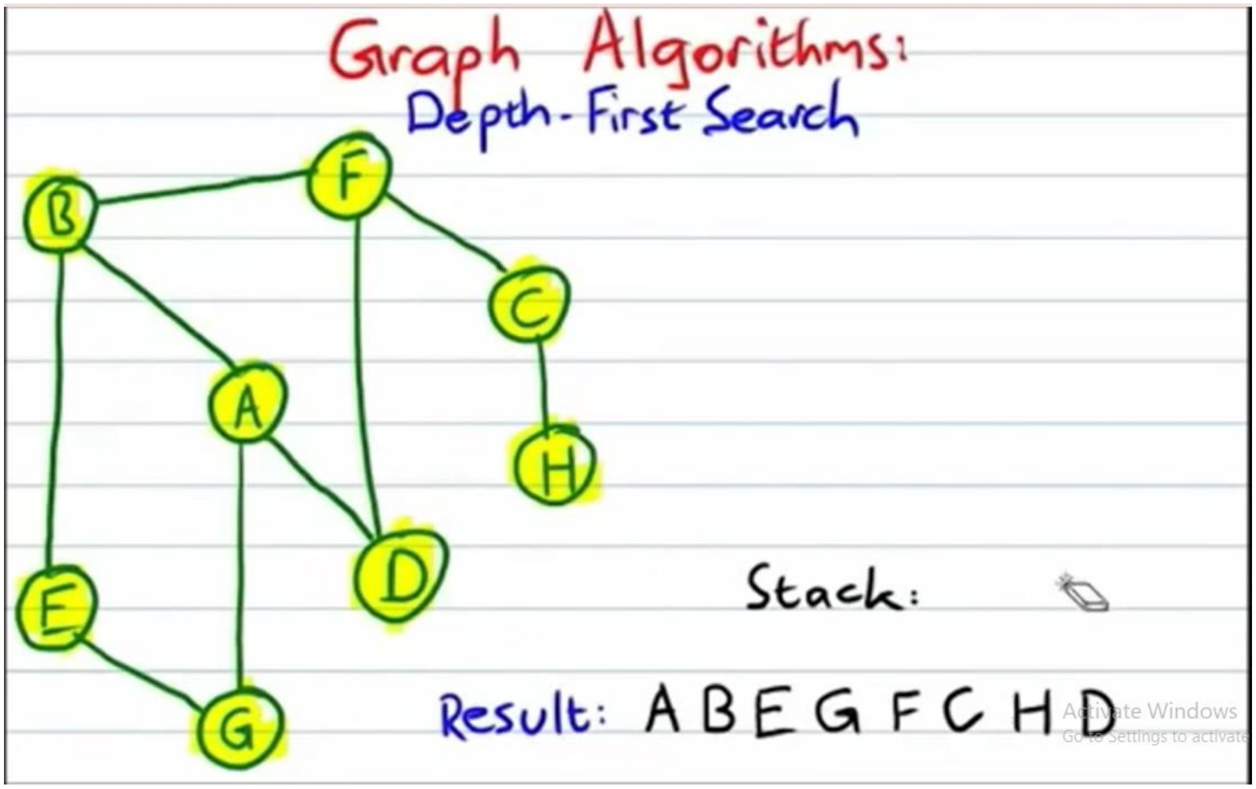


Graph Algorithms:
Depth-First Search

Stack: A B F

Result: A B E G F C H





\

Chapter 5: Back Tracking

- Many problems which deal with searching for a set of solutions or for an optimal solution satisfying some constraints can be solved using the backtracking formulation.
- To apply backtracking method, the desired solution must be expressible as an n-tuple ($X_1 \ldots X_n$) where X_i is chosen from some finite set S_i.
- The problem is to find a vector, which maximizes or minimizes or satisfies a criterion function $P(X_1 \ldots. X_n)$.
- Its basic idea is to build up the solution vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major advantage of this method is, once we know that a partial vector ($X_1 \ldots X_i$) will not lead to an optimal solution, then ($m_{i+1} \ldots. m_n$) possible test vectors may be ignored entirely.
- Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.
- These constraints are classified as:

  i) Explicit constraints.

  ii) Implicit constraints.

1) Explicit constraints:

Explicit constraints are rules that restrict each Xi to take values only from a given set.

Some examples are,

Xi >=0	or	Si = {all non-negative real nos.}
Xi =0 or 1	or	Si={0, 1}.
$Li \le Xi \le Ui$	or	Si= {a: $Li \le a \le Ui$}

- All tuples that satisfy the explicit constraint define a possible solution space for I.

2) Implicit constraints:

The implicit constraints determine which of the tuples in the solution space I can actually satisfy the criterion function.

Some terminologies used in backtracking

- Each node in the tree is called a **problem state.**
- All paths from the root to other nodes define the **state space** of the problem.

- **Solution states**: These are the problem states S for which the path from root to S define a tuple in the solution space.
- **Answer states:** These are the leaf nodes which correspond to an element in the set of solutions, i.e. these are the states which satisfy the implicit constraints.
- The tree organization of the solution space is referred to as the **state space tree**.
- A node which has been generated and all of whose children have not yet been generated is called **live node**.
- The live node whose children are currently being generated is called an **E-node**.
- A **dead node** is a generated node which is not to be expanded further or all of whose children have been generated.

  There are two methods of generating state space tree

  - **Backtracking:** In this method, as soon as a new child C of current E-node N is generated, this child will be the new E-node. The N will become the E-Node again when the sub tree C has been fully explored.
  - **Branch and Bound:** E-node will remain as E-node until it is dead.

5.1. 8 Queens Problem

This 8 queens problem is to place n-queens in an 'N*N' matrix in such a way that no two queens attack each otherwise no two queens should be in the same row, column, diagonal.

Solution:

- The solution vector $X(X_1 \ldots X_n)$ represents a solution in which Xi is the column of the Ith row where Ith queen is placed.
- First, we have to check no two queens are in same row.
- Second, we have to check no two queens are in same column.
- The function, which is used to check these two conditions, is [I, X (j)], which gives position of the Ith queen, where I represents the row and X (j) represents the column position.
- Third, we have to check no two queens are in the same diagonal.
- Consider two dimensional array A[1:n,1:n] in which we observe that every element on the same diagonal that runs from upper left to lower right has the same (row - column) value.
- Also, every element on the same diagonal that runs from upper right to lower left has the same (row + column) value.

- Suppose two queens are in same position (i,j) and (k,l) then two queens lie on the same diagonal , if and only if |j-l|=|i-k|.

STEPS TO GENERATE THE SOLUTION:

- Initialize x array to zero and start by placing the first queen in k=1 in the first row.
- To find the column position start from value 1 to n, where ‘n’ is the no. of columns or no. of queens.
- If k=1 then x(k)=1.so (k,x(k)) will give the position of the kth queen. Here we have to check whether there is any queen in the same column or diagonal.
- For this considers the previous position, which had already, been found out. Check whether X(i)=X(k) for column |X(i)-X(k)|=(i-k) for the same diagonal.
- If any one of the conditions is true then return false indicating that kth queen can’t be placed in position X(k).
- For not possible condition increment X(k) value by one and precede until the position is found.
- If the position X(k) $\leq$ n and k=n then the solution is generated completely.
- If k<n, then increment the ‘k’ value and find position of the next queen.
- If the position X(k)>n then kth queen cannot be placed as the size of the matrix is ‘N*N’.
- So decrement the ‘k’ value by one i.e. we have to back track and after the position of the previous queen.

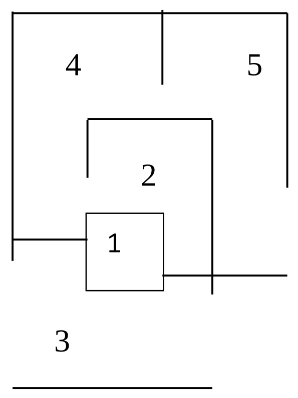
Algorithm:

```
Algorithm place(k,i)
//return true if a queen can be placed in kth row and ith column. Otherwise it
//returns false X[] is a global array whose first k-1 values have been set.
//abs() returns the absolute value of r.
{
  for j=1 to k-1 do
        if ((X[j]=i)            //two in the same column.
                or (abs (X[j]-i)=abs (j-k)))     //or in the same diagonal.
        then return false;
  return true;
}
```

5.2. Graph Coloring

- Let 'G' be a graph and 'm' be a given positive integer. If 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. So it's called M-color ability decision problem.
- The graph G can be colored using the smallest integer 'm'. This integer is referred to as chromatic number of the graph.
- A graph is said to be planar iff it can be drawn on plane in such a way that no two edges cross each other.
- Suppose we are given a map then, we have to convert it into planar. Consider each and every region as a node. If two regions are adjacent then the corresponding nodes are joined by an edge.

Consider a map with five regions and its graph.



1 is adjacent to 2, 3, 4.

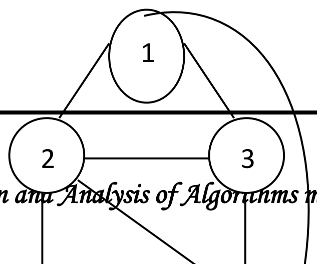
2 is adjacent to 1, 3, 4, 5

3 is adjacent to 1, 2, 4

4 is adjacent to 1, 2, 3, 5

5 is adjacent to 2, 4

Planar graph representation of map



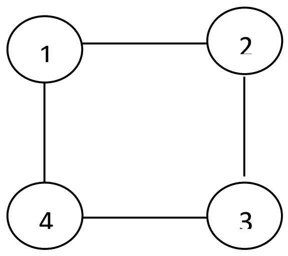
Steps to color the Graph:

- First create the adjacency matrix graph(1:m,1:n) for a graph, if there is an edge between i,j then C(i,j) = 1 otherwise C(i,j) =0.
- The Colors will be represented by the integers 1,2,…..m and the solutions will be stored in the array X(1),X(2),………..,X(n) ,X(index) is the color, index is the node.
- The formula which is used to set the color is,

  X(k) = (X(k)+1) % (m+1)
- First one chromatic number is assigned ,after assigning a number for 'k' node, we have to check whether the adjacent nodes has got the same values if so then we have to assign the next value.
- Repeat the procedure until all possible combinations of colors are found.
- The function which is used to check the adjacent nodes and same color is,

  If((Graph (k,j) == 1) and X(k) = X(j))

Example:



N= 4

M= 3

Adjacency Matrix:

$\left| \begin{matrix} 0 & 1 & 0 & 1 \end{matrix} \right|$

$$\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{array}$$

 Problem is to color the given graph of 4 nodes using 3 colors.

Node-1 can take the given graph of 4 nodes using 3 colors.

 The state space tree will give all possible colors in that, the numbers which are inside the circles are nodes, and the branch with a number is the colors of the nodes.

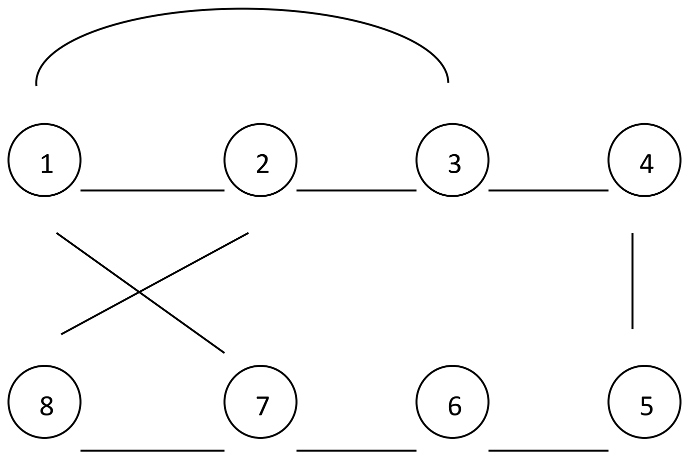
Time complexity for m coloring

 The time spent by Nextvalue to determine the children is O(mn)

Total time is = $O(nm^n)$.

5.3. Hamiltonian Cycle

- Let G=(V,E) be a connected graph with 'n' vertices. A HAMILTONIAN CYCLE is a round trip path along 'n' edges of G that visits every vertex once and returns to its starting position.
- If the Hamiltonian cycle begins at some vertex V1 belongs to G and the vertices of G are visited in the order of V1,V2…….Vn+1,then the edges (Vi,Vi+1) are in E,1<=i<=n, and the Vi are distinct except for V1 and Vn+1 which are equal.
- Consider an example 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(X_1……..X_n) 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,b’coz 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,

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

   2. 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.
8. Repeat the above steps to generate possible Hamiltonian cycle.

Algorithm :(finding all Hamiltonian cycle)

```
Algorithm Hamiltonian(k)
{
  repeat
  {
   // generate values for X[k].
      Nextvalue(k);    // Assign a legal next value to  X[k] .
      if (X[k]=0) then return;
      if (k=n) then
            Write(x[1:n]);
      else Hamiltonian(k+1);
   } until(false);
}
```

```
Algorithm Nextvalue (k)
{
   repeat
  {
         X [k]=(X [k]+1) mod (n+1); //next vertex
         if (X [k]=0) then return;
         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)) then return;
         }
 } until (false);
}
```

- It begin by $X_1=1$; X_k (for k=2 to n-1) can be any vertex v that is distinct from $X_1, X_2, \ldots X_{k-1}$ and V is connected by an edge to X_{k-1}.
- This algorithm is started by initializing matrix G[1:n,1:n] then setting x[2:n] to zero, and X[1] to 1.
- Hamiltonian (2); is called first.

5.4. Knapsack Problems

0/1 KNAPSACK PROBLEM:

- This problem is similar to ordinary knapsack problem but we may not take a fraction of an object.
- We are given ' N ' object with weight W_i and profits P_i where I varies from l to N and also a knapsack with capacity ' M '.
- The problem is, we have to fill the bag with the help of ' N ' objects and the resulting profit has to be maximum
- Formally, the problem can be started as, maximize $\sum_{i=1}^{n} X_i P_i$

$$\text{subject to } \sum_{i=l}^{n} X_i W_i \text{ L } M$$

- Where X_i are constraints on the solution $X_i \in \{0,1\}$. (u) X_i is required to be 0 or 1. if the object is selected then the unit in 1. if the object is rejected than the unit is 0. That is why it is called as 0/1, knapsack problem.
- To solve the problem by dynamic programming we up a table T[1…N, 0…M] (ic) the size is N. where 'N' is the no. of objects and column starts with 'O' to capacity (ic) 'M'.

5.5. Traveling Salesman Problems

1. TRAVELLING SALESMAN PROBLEM

- Let G(V,E) be a directed graph with edge cost c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \propto$,if $<i,j> \notin E$.

  Let $|V| = n$ and assume n>1.
- The traveling salesman problem is to find a tour of minimum cost.
- A tour of G is a directed cycle that include every vertex in V.
- The cost of the tour is the sum of cost of the edges on the tour.
- The tour is the shortest path that starts and ends at the same vertex (ie) 1.

APPLICATION:

1. Suppose we have to route a postal van to pick up mail from the mail boxes located at 'n' different sites.
2. An n+1 vertex graph can be used to represent the situation.
3. One vertex represents the post office from which the postal van starts and returns.
4. Edge <i,j> is assigned a cost equal to the distance from site 'i' to site 'j'.
5. The route taken by the postal van is a tour and we are finding a tour of minimum length.
6. Every tour consists of an edge <1,k> for some $k \in V-\{\}$ and a path from vertex k to vertex 1.
7. The path from vertex k to vertex 1 goes through each vertex in V-{1,k} exactly once.
8. the function which is used to find the path is

   $g(1,V-\{1\}) = \min\{ c_{ij} + g(j,S-\{j\})\}$
9. g(i,s) be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1.

The function g(1,v-{1}) is the length of an optimal tour

Reading assignment

Chapter 6: Introduction to Probabilistic Algorithms - Parallel Algorithms (2hr)