UNIT IV
Backtracking

Syllabus

Introduction –

- Introduction to backtracking - branch and bound
  - N queen's problem - backtracking
  - Sum Of Subsets,
- Graph Introduction,
  - Hamiltonian's Circuit,
  - Travelling Salesman Problem,
- Graph algorithms
  - Depth first search and Breadth first search
  - Shortest path introduction
    - Floyd-Warshall Introduction

Session Learning Outcomes

- To learn about backtracking approach
- To know the applications of backtracking algorithm
- To solve problems or puzzle using backtracking approach
- To know about the concepts of N-Queens problem and Sum of subset problem
- To analyze the time complexities of N-Queens and Sum of subset problems

Backtracking

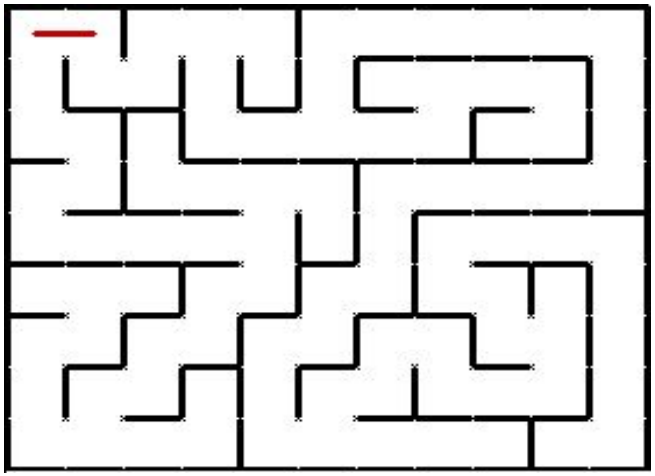
- Recursion is the key in backtracking programming.

  - ***We start with one possible move out of many available moves and try to solve the problem***
  - ***if we are able to solve the problem with the selected move then we will print the solution***
  - ***else we will backtrack and select some other move and try to solve it.***
  - ***If none if the moves work out, we will claim that there is no solution for the problem.***

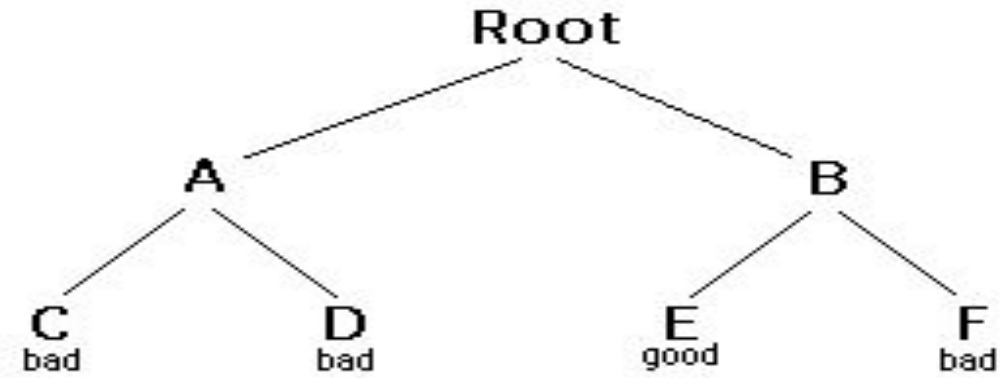
Backtracking

- Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

Introduction to Backtracking



Simple Example



1.Starting at Root, your options are A and B. You choose A.
2.At A, your options are C and D. You choose C.
3.C is bad. Go back to A.
4.At A, you have already tried C, and it failed. Try D.
5.D is bad. Go back to A.
6.At A, you have no options left to try. Go back to Root.
7.At Root, you have already tried A. Try B.
8.At B, your options are E and F. Try E.
9.E is good. Congratulations!

When to use a Backtracking algorithm?

- When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:
  - A piece of sufficient information is not available to make the best choice, so we use the backtracking strategy to try out all the possible solutions.
  - Each decision leads to a new set of choices. Then again, we backtrack to make new decisions. In this case, we need to use the backtracking strategy.

Generalized Algorithm:

Pick a starting point.

while(Problem is not solved)

For each path from the starting point.

check if selected path is safe, if yes select it

and make recursive call to rest of the problem

If recursive calls returns true, then return true.

else undo the current move and return false.

End For

If none of the move works out, return false,

NO SOLUTON.

Branch and Bound technique

- The tree organization of the solution space is referred to as the **state space tree.**

Live node:

- A node which has been generated and all of whose children have not yet been generated is called alive node

E node:

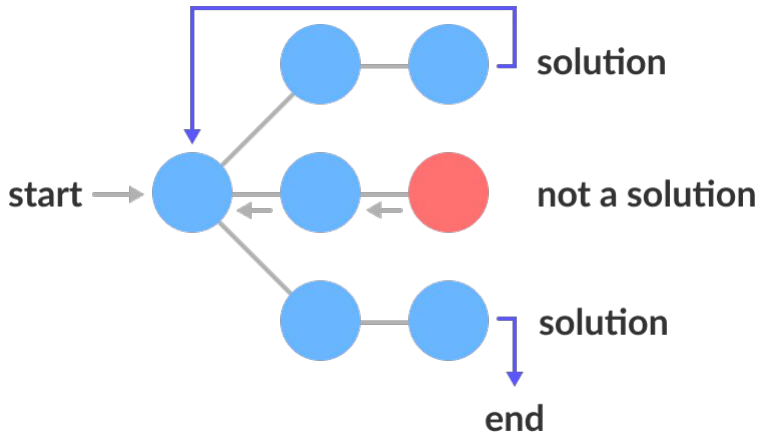
- The live node whose children are currently being generated is called the E-node (node being expanded).

Dead node:

- A dead node is a generated node, which is not to be expanded further or all of whose children have been generated.
- Bounding functions are used to kill live nodes without generating all their children.

State Space Tree

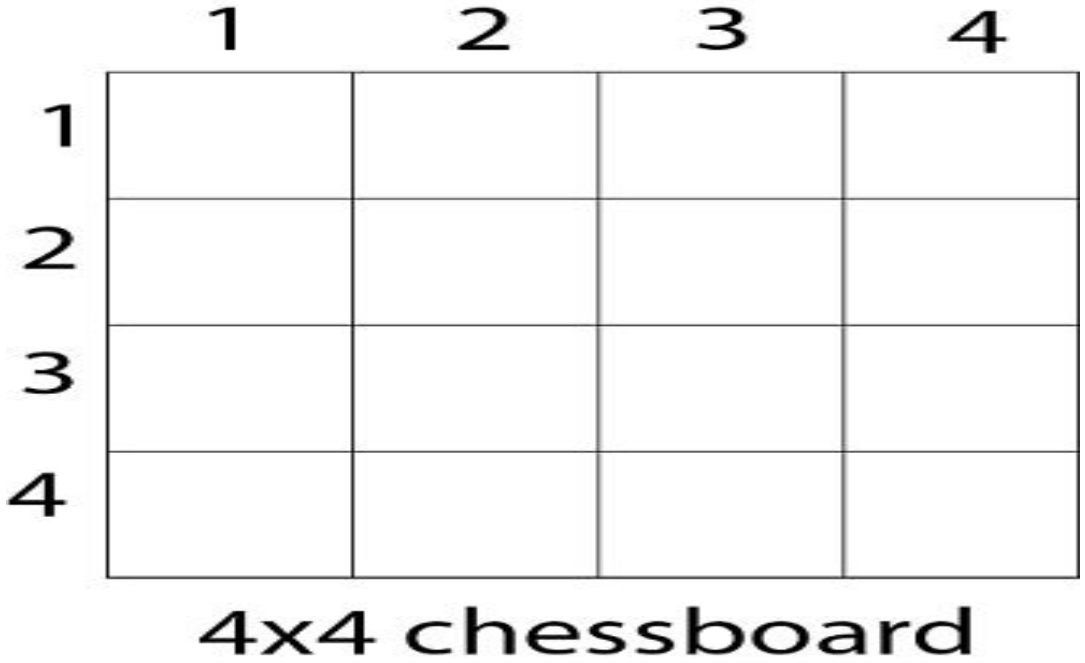
A space state tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.




```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

N-Queens Problem

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



4x4 chessboard

N-Queens Problem

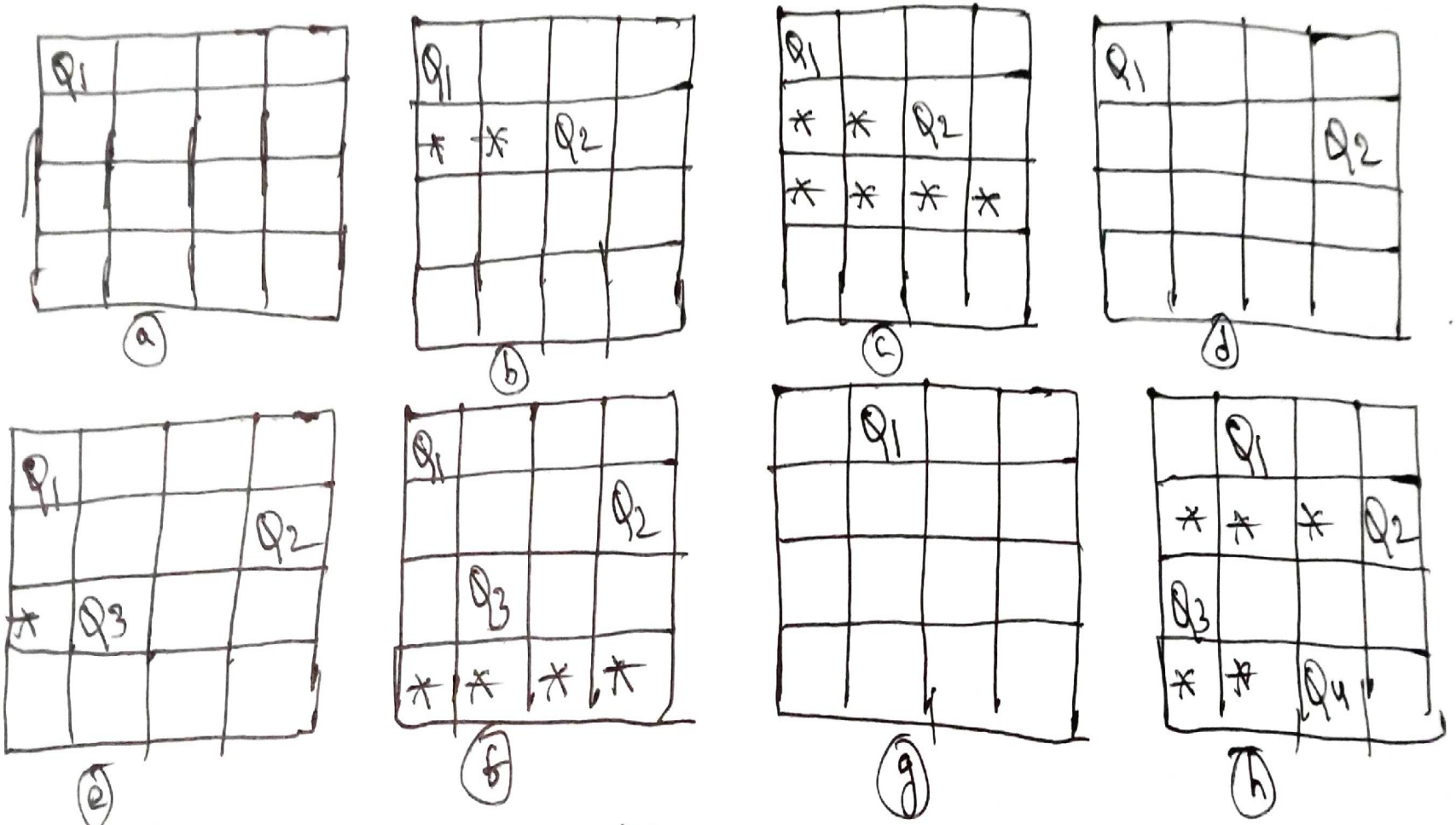
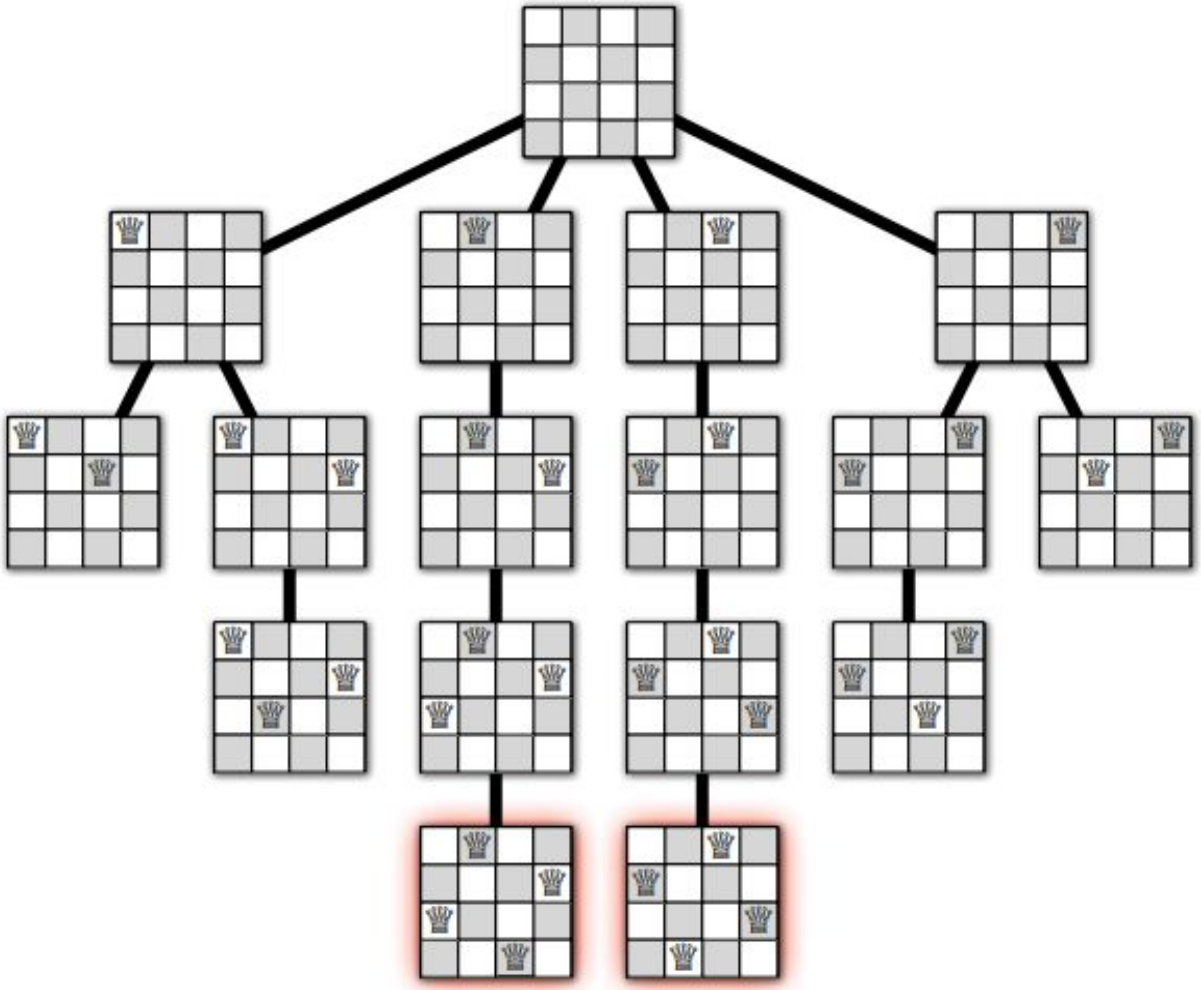


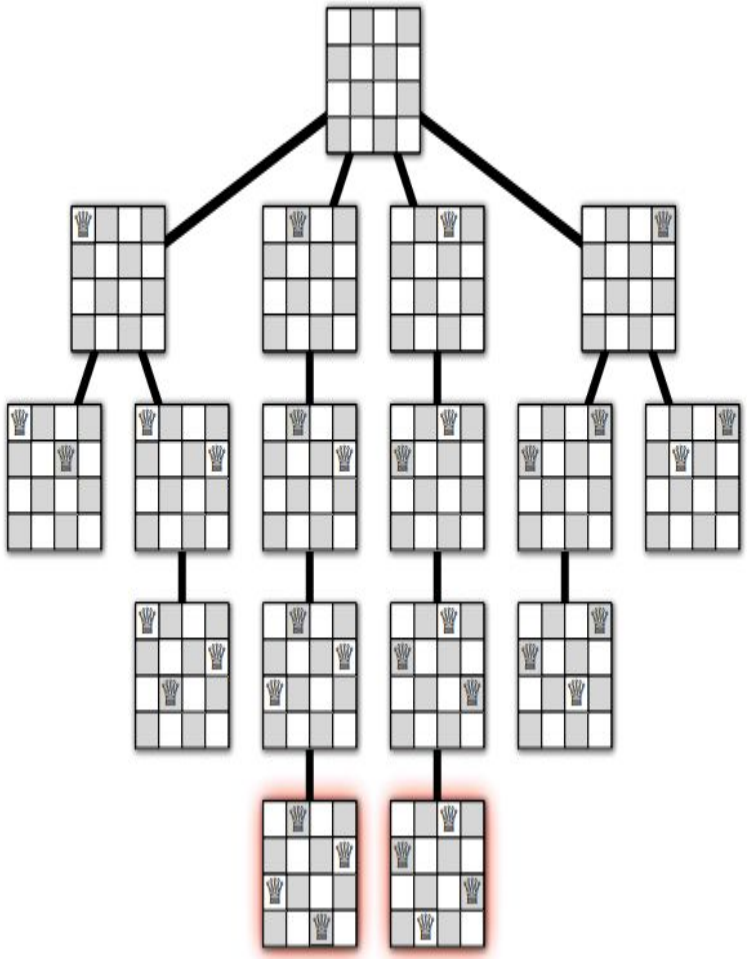
fig (2) Back Tracking solution to 4-queens problem

N-Queens Problem

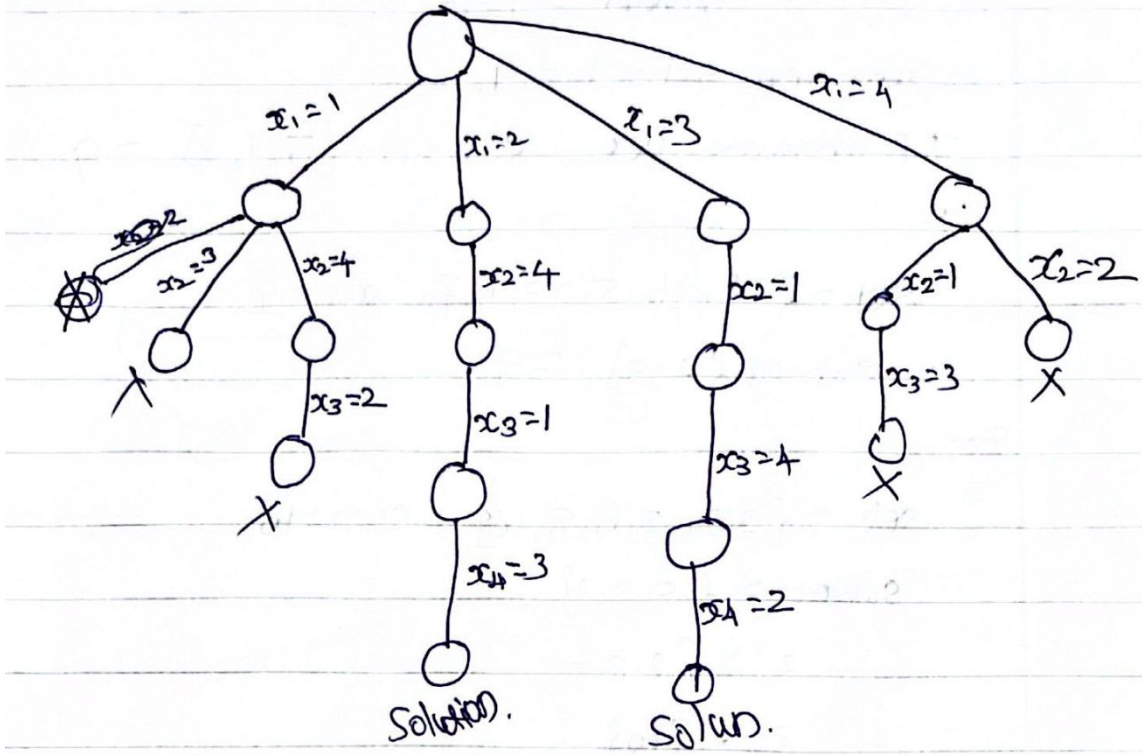
- Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i.“
- Now, we place queen q_1 in the very first acceptable position (1, 1).
- Next, we put queen q_2 so that both these queens do not attack each other.
- We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely.
- So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2).
- But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.

State Space Tree



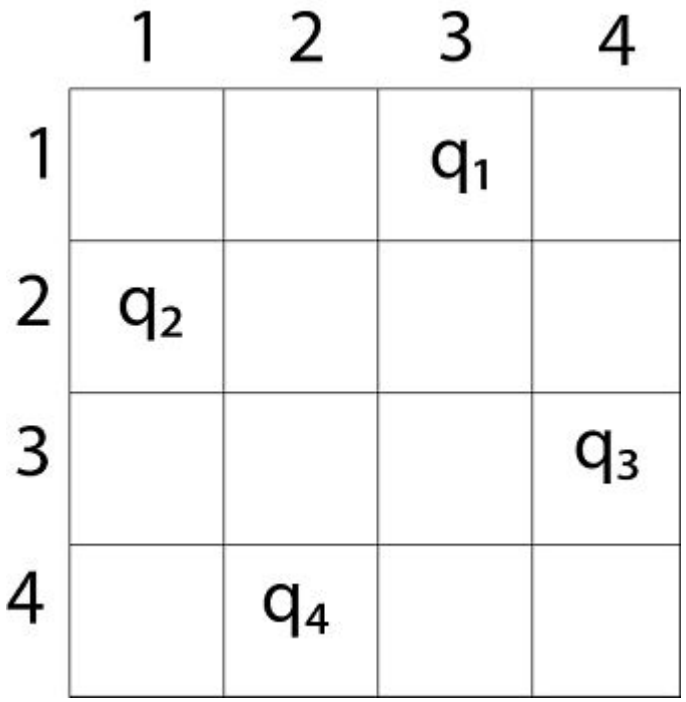


State space tree.



$n + n(n-1) + n(n-1)(n-2) + \cdots 1 = n!$

N-Queens Problem



N-Queens Problem

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 the 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.

N-Queens Problem

```
Algorithm N_QUEEN (k, n)
// Description : To find the solution of n x n queen problem using backtracking
// Input :
n: Number of queen
k: Number of the queen being processed currently, initially set to 1.

// Output : n x 1 Solution tuple

for i ← 1 to n do
  if PLACE(k , i) then
    x[k] ← i
    if k == n then
      print X[1…n]
    else
      N_QUEEN(k + 1, n)
    end
  end
end
```

Complexity Analysis

- The constraint that no two queens in an attacking position reduces the calculation to

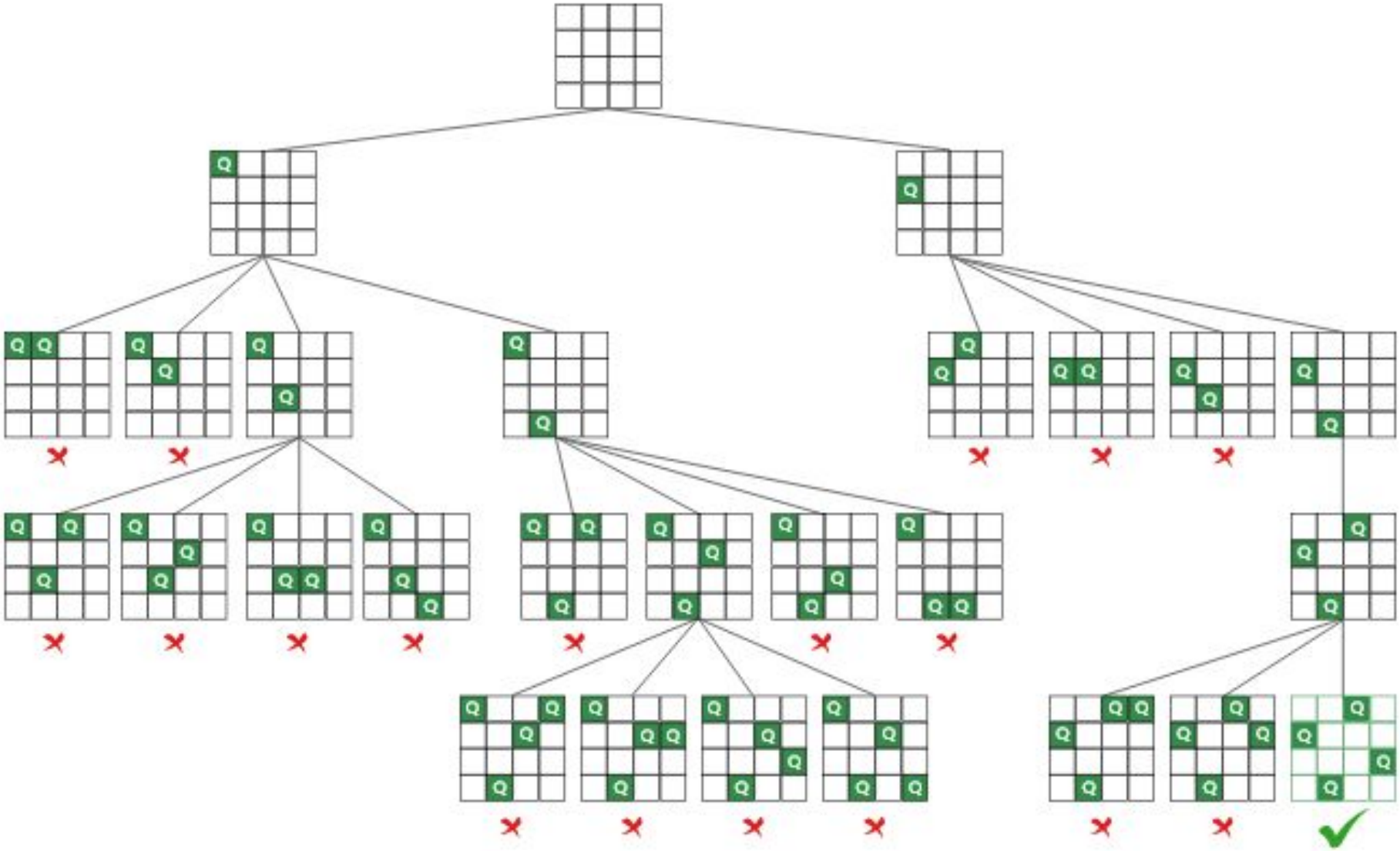
  (For four queens) 1+4*3+4*3*2+4*3*2*1 promising nodes

  In general,

  n+n(n-1)+n(n-1)(n-2)+…….+1 = n! promising nodes are possible.

- The best, average and worst case time complexity of this algorithm is O(N!) where N is number of queens.

N-Queens Problem Column Wise



Subset Sum Problem

- It is one of the most important problems in complexity theory. The problem is given an A set of integers a1, a2,…., an upto n integers. The question arises that is there a non-empty subset such that the sum of the subset is given as M integer?.
- For example, the set is given as [5, 2, 1, 3], and the sum of the subset is 9; the answer is YES as the sum of the subset [5, 3, 1] is equal to 9. It is the special case of knapsack

Let's understand this problem through an example.

- **Statement:** Given a set of positive integers, and a value sum, determine that the sum of the subset of a given set is equal to the given sum.
- **Or**
- Given an array of integers and a sum, the task is to have all subsets of given array with sum equal to the given sum.

Example 1:

Input: set[] = {4, 16, 5, 23, 12}, sum = 9

Output = true

Subset {4, 5} has the sum equal to 9.

Example 2:

Input: set[] = {2, 3, 5, 6, 8, 10}, sum = 10

Output = true

There are three possible subsets that have the sum equal to 10.

Subset1: {5, 2, 3}

Subset2: {2, 8}

Subset3: {10}

Sum of Subsets problem

Sum of Subsets Problem

Suppose we are given n-distinct positive numbers (weights) we desire to find all combinations of these numbers whose sums are m. This is called the Sum of the Subsets problem.

We formulate using either "fixed" or "variable sized" tuple size strategy. In this case the element x_i of the solution vector is either 0 or 1 depending on whether the weight w_i is included or not.

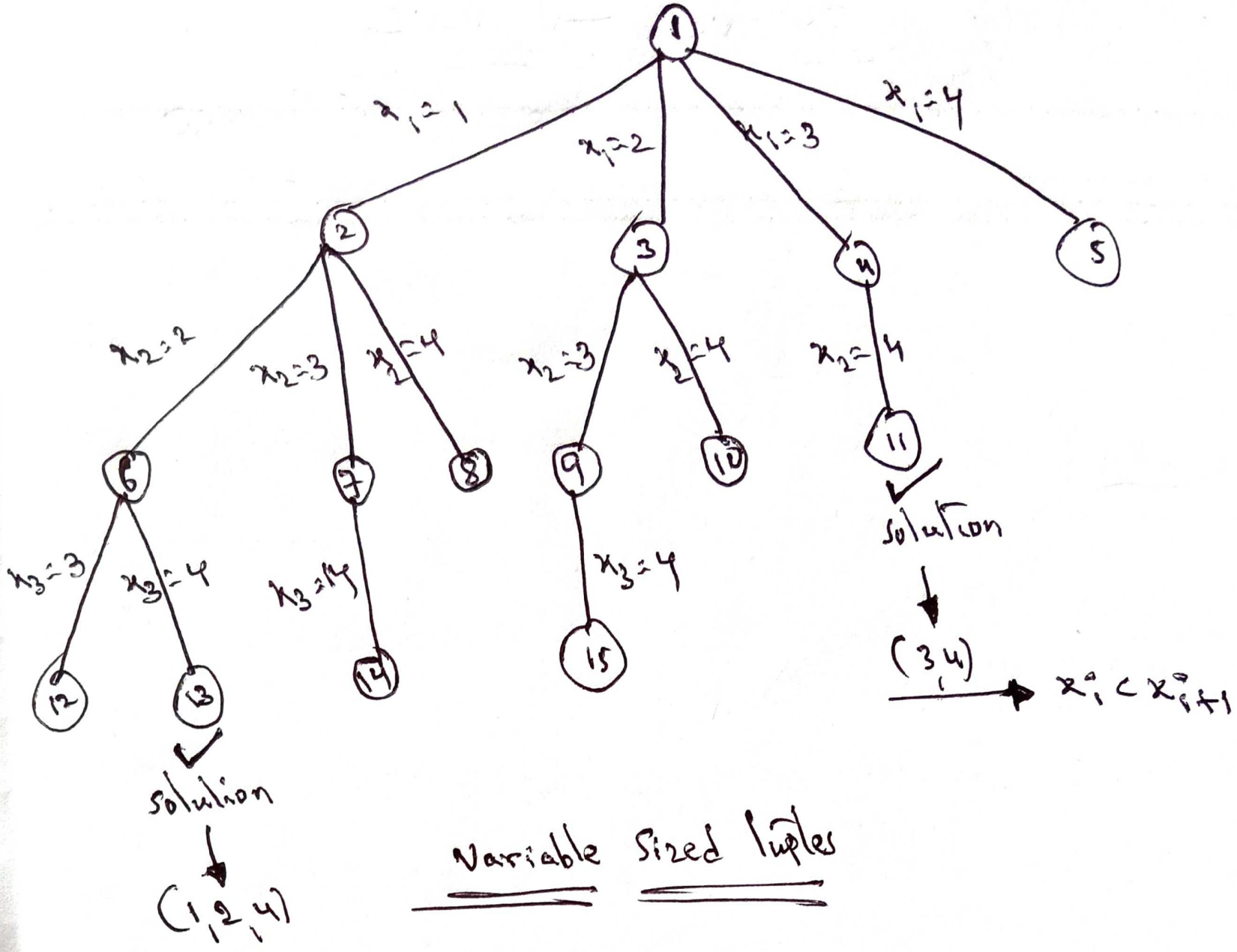
Ex: Given positive numbers w_i, $1 \le i \le n$ and m, the problem calls for finding all subsets of the w_i whose sums are m. For ex: $n = 4$ and $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ and $m = 31$, then the desired solutions are $(1, 2, 4)$ and $(3, 4)$. In general all solutions are n-Tuple $(x_1, x_2, \ldots x_n)$

Different solutions have different sized tuples — variable sized tuples

Let's understand that how can we solve the problem using recursion. Consider the array which is given below:

arr = [11, 13, 24, 7] , sum = 31

Sum of Subsets problem

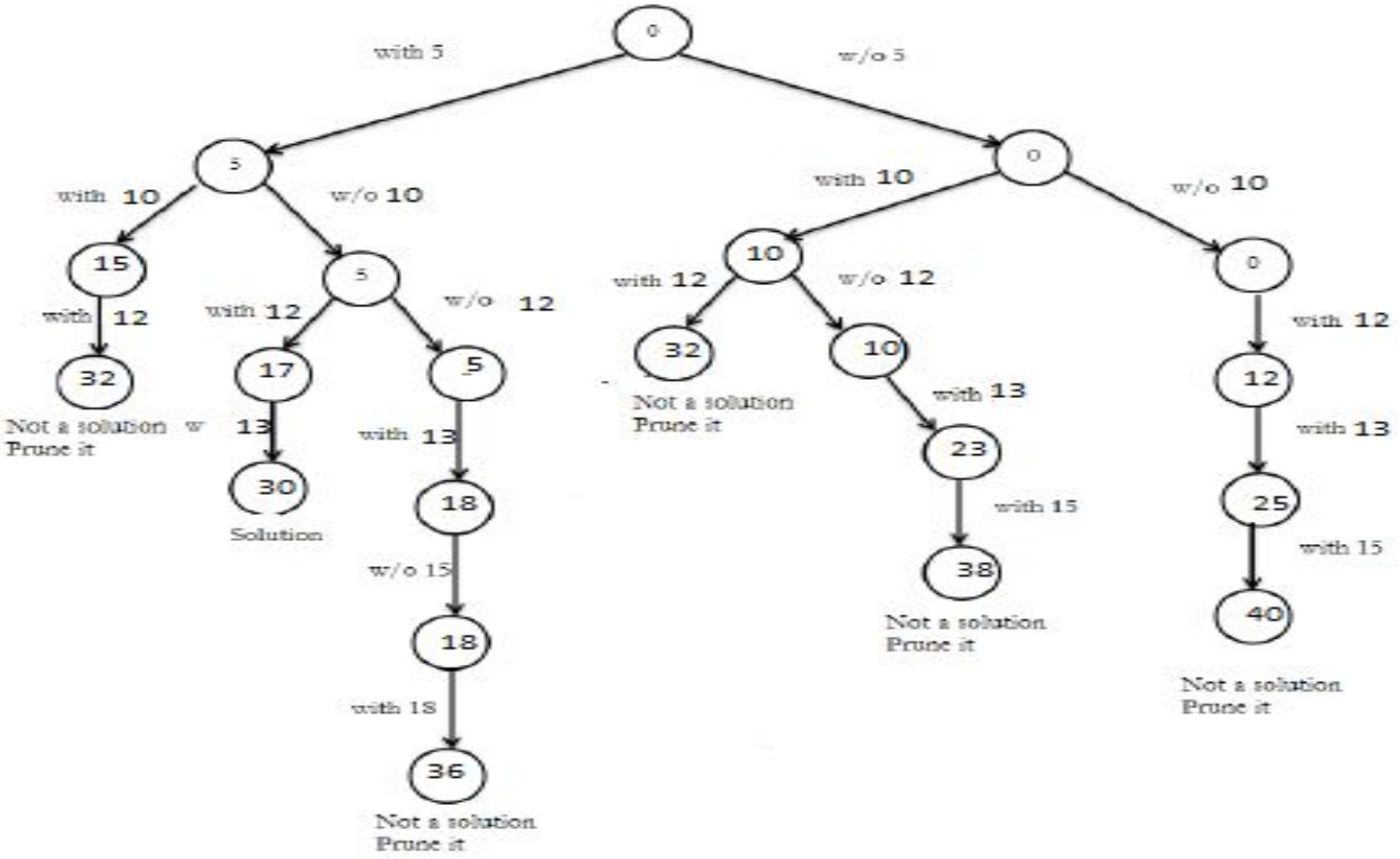


Example

- Example: Solve following problem and draw portion of state space tree M=30,W ={5, 10, 12, 13, 15, 18}

Initially subset = {}	Sum = 0	Description
5	5	Then add next element.
5, 10	15 i.e. 15 < 30	Add next element.
5, 10, 12	27 i.e. 27 < 30	Add next element.
5, 10, 12, 13	40 i.e. 40 < 30	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 15	42	Sum exceeds M = 30. Hence backtrack.
5, 10, 12, 18	45	Sum exceeds M = 30. Hence backtrack.
5, 12, 13	30	Solution obtained as M = 30

State Space Tree



Steps for Subset Sum Problem

- **Steps**:

1. Start with an empty set
2. Add the next element from the list to the set
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 (sum of subset < M) then go to 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.

Pseudocode for Subset Sum Problem

```
void subset_sum(int list[], int sum, int starting_index, int target_sum)
{
        if( target_sum == sum )
        {
            subset_count++;
            if(starting_index < list.length)
                subset_sum(list, sum - list[starting_index-1], starting_index, target_sum);
        }
        else
        {
            for( int i = starting_index; i < list.length; i++ )
            {
                subset_sum(list, sum + list[i], i + 1, target_sum);
            }
        }
    }
```

Analysis

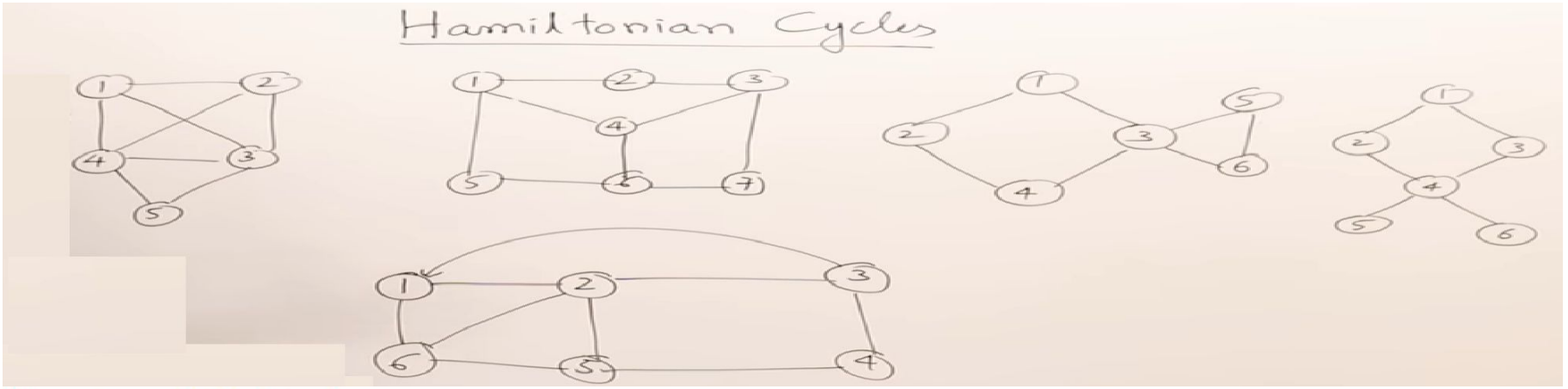
- The time complexity of solving Subset sum problem using backtracking approach is O(2^N) where N is the given number elements in the set.

Hamiltonian Circuit Problems

Hamiltonian Circuit Problems

- Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach. We start our search from any arbitrary vertex say **'a.'** This vertex **'a'** becomes the root of our implicit tree.
- The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed.
- The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex **'a'** then we say that **dead end** is reached.
- In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian Cycle is obtained.

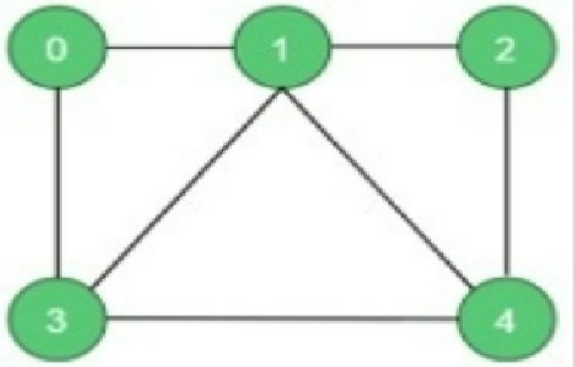
Hamiltonian Circuit Problems



Input and Output

Input:
The adjacency matrix of a graph G(V, E).

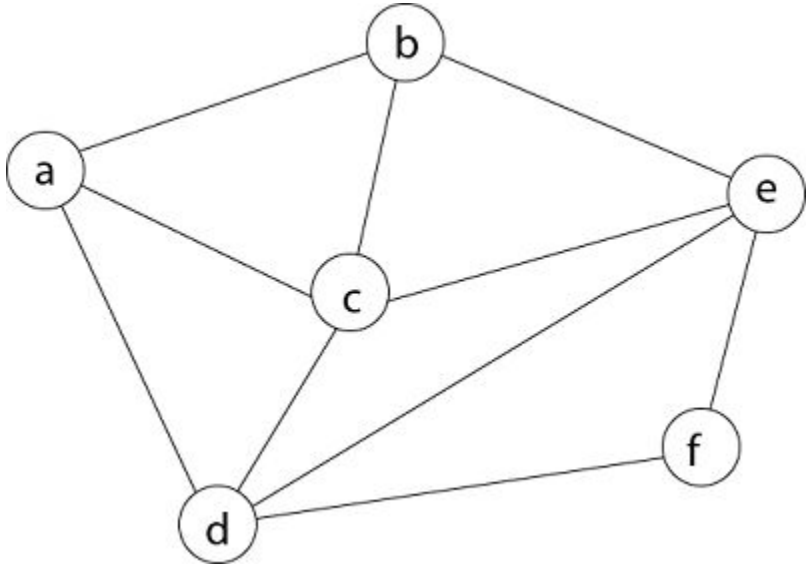
0	1	0	1	0
1	0	1	1	1
0	1	0	0	1
1	1	0	0	1
0	1	1	1	0



Output:
The algorithm finds the Hamiltonian path of the given graph. For this case it is (0, 1, 2, 4, 3, 0). This graph has some other Hamiltonian paths.
If one graph has no Hamiltonian path, the algorithm should return false.

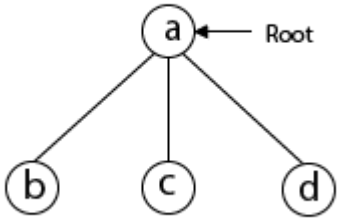
Hamiltonian Circuit Problems

- **Example:** Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.

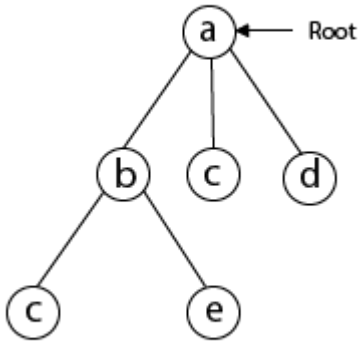


- **Solution:** Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.

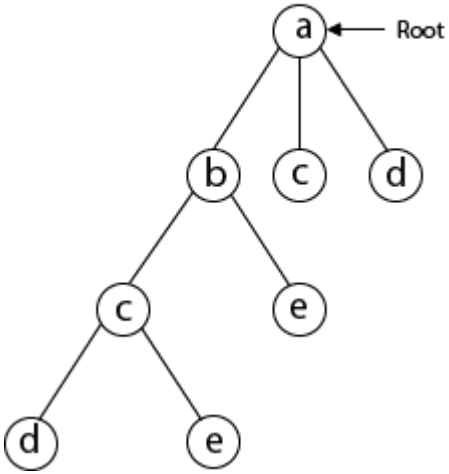
- Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



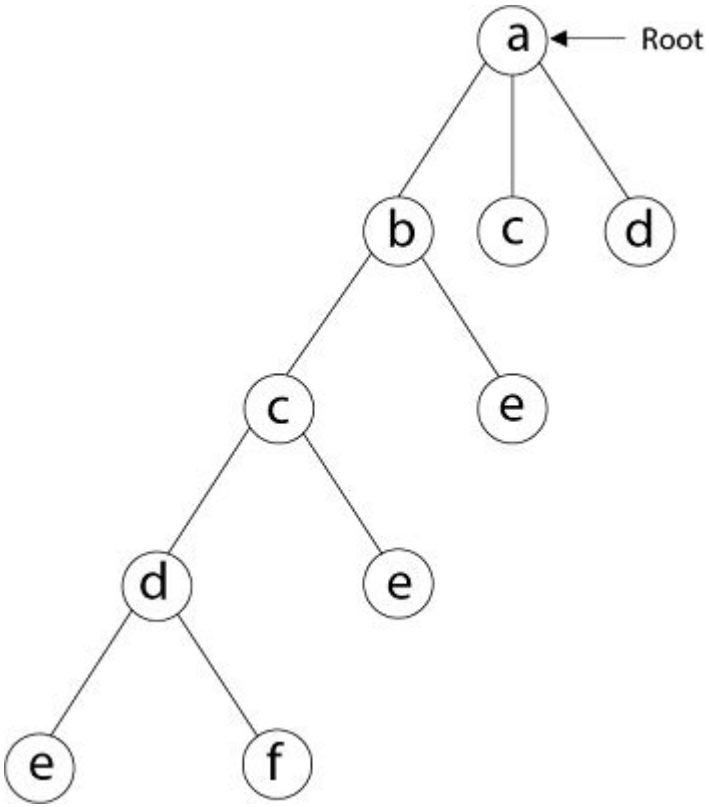
- Next, we select 'c' adjacent to 'b.'

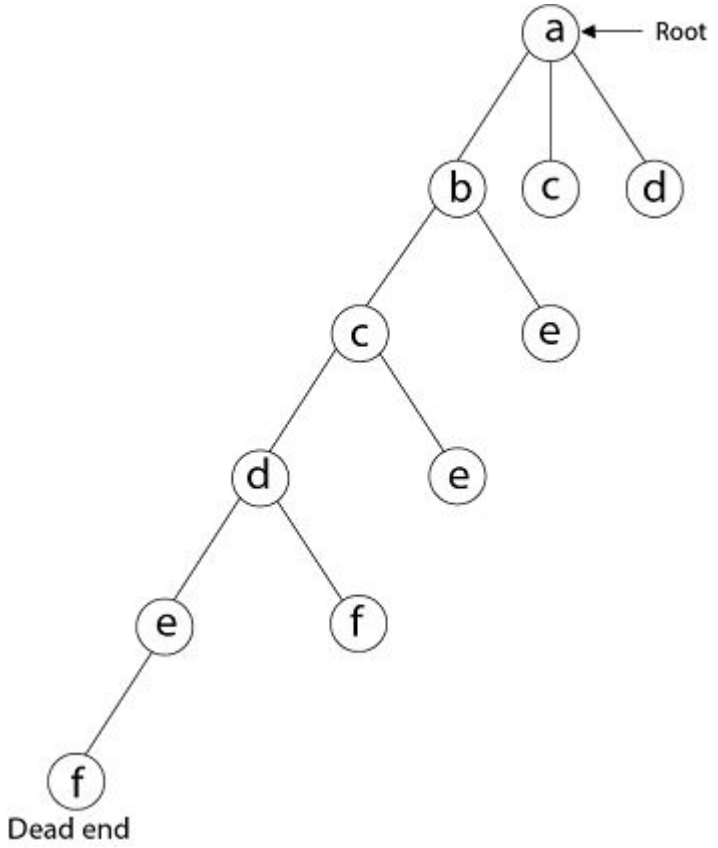


- Next, we select 'd' adjacent to 'c.'



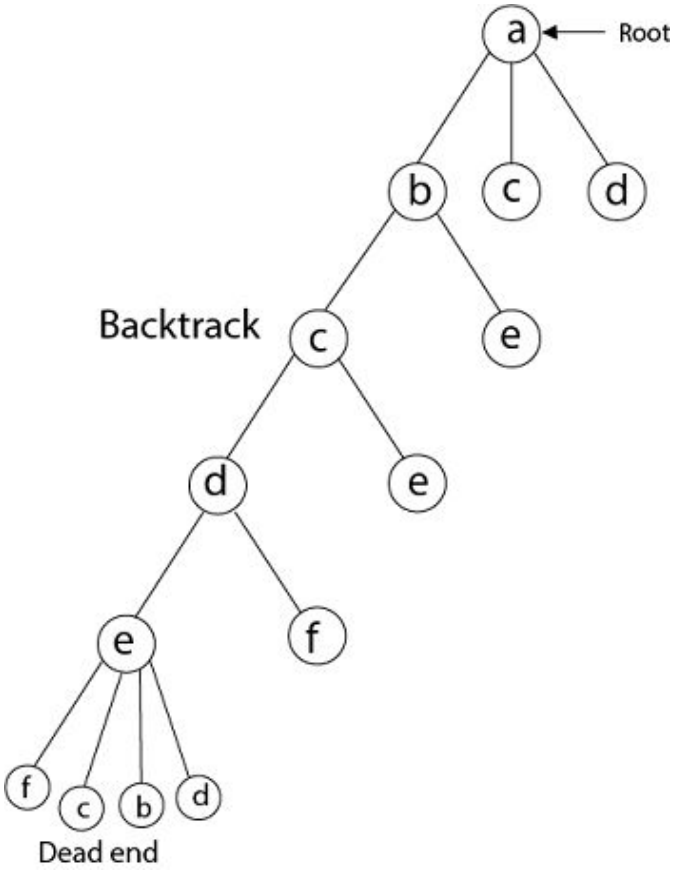
- Next, we select 'e' adjacent to 'd.'



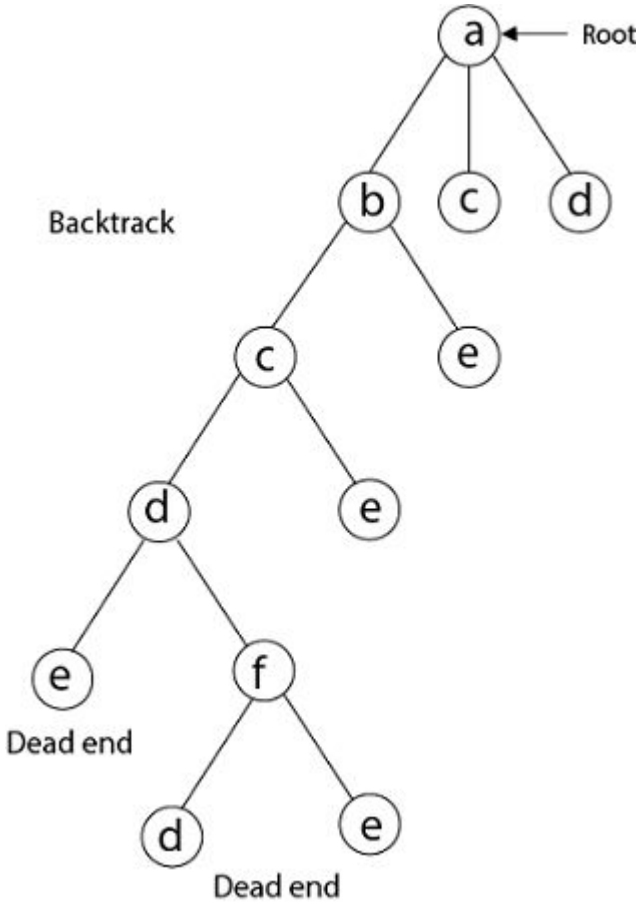


- Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.

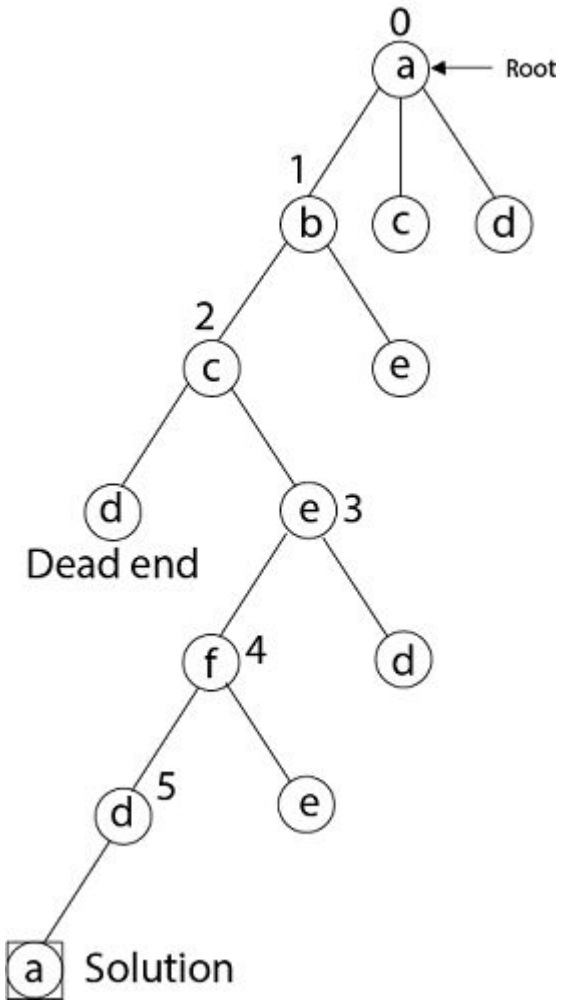
- From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.
- Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).



- Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.
-



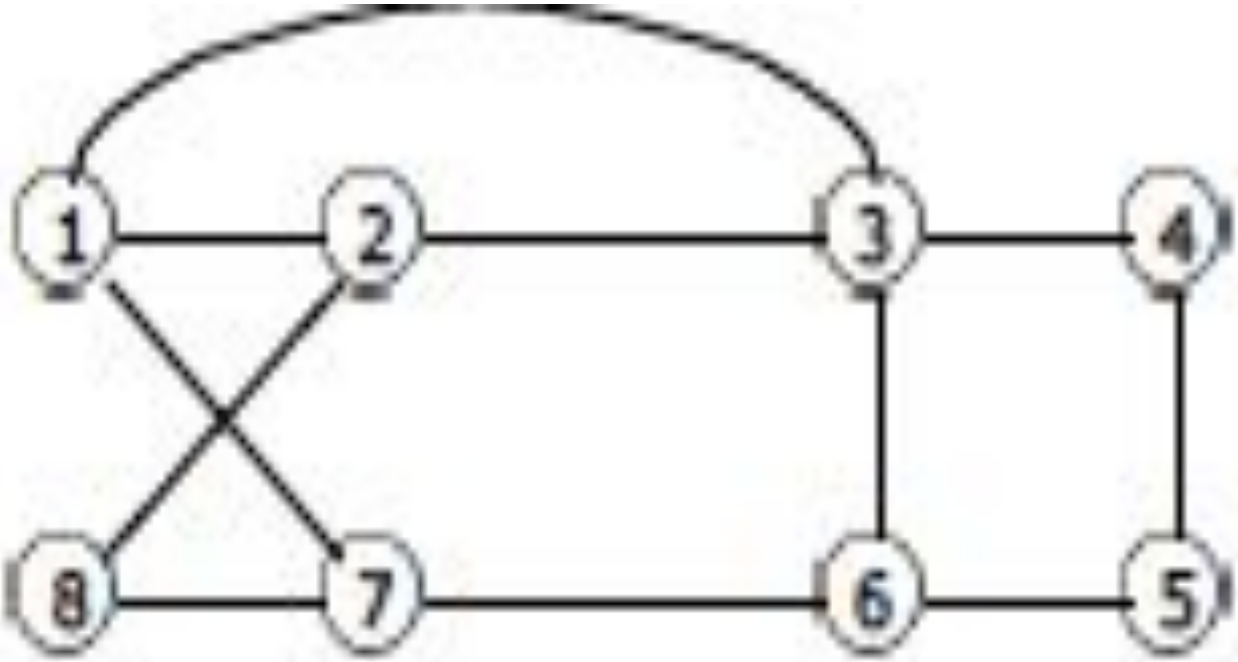
Again Backtrack



- **Time Complexity**
- Time Complexity for finding the **Hamiltonian Cycle** using the naive approach is $O(N!)$ or O(n^n), where N is the number of **vertices** in the graph.

Consider an example graph G1.

- The graph G1 has Hamiltonian cycles:

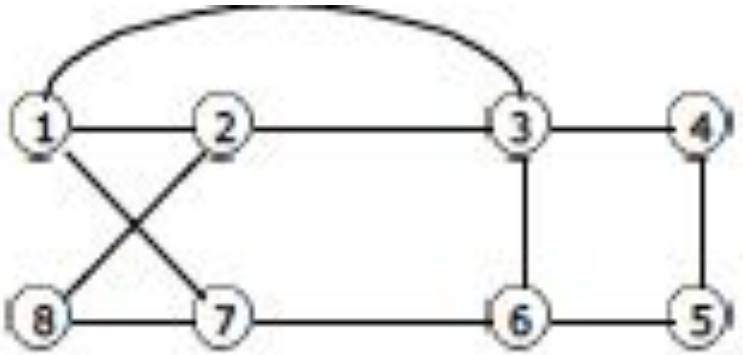


Consider an example graph G1.

- The graph G1 has Hamiltonian cycles:

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

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



•Algorithm Hamiltonian Circuit Problems

isValid(v, k)

Input – Vertex v and position k.

Output – Checks whether placing v in the position k is valid or not.

```
Begin
   if there is no edge between node(k-1) to v, then
      return false
   if v is already taken, then
      return false
   return true; //otherwise it is valid
End
```

Algorithm

- Algorithm Hamiltonian (k)

  Loop
  - Next value (k)
  - If (x (k)=0) then return;
    - If k=n then Print (x)
    - Else Hamiltonian (k+1);
    - End if

    Repeat

Algorithm Nextvalue (k)

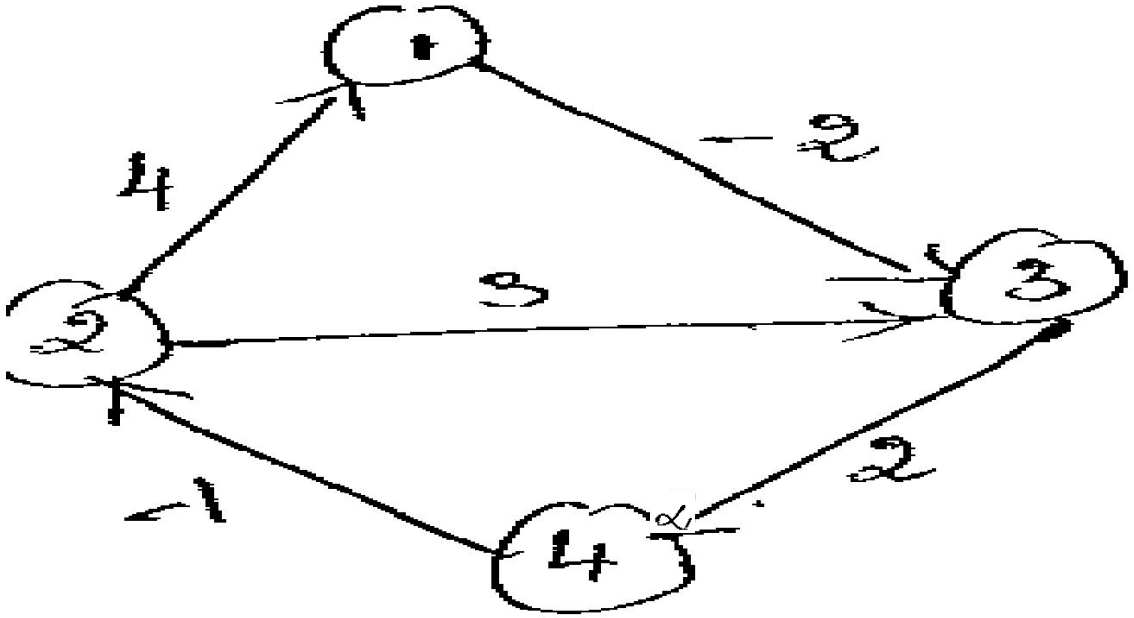
```
{
 Repeat
        X [k]=(X [k]+1) mod (n+1); //next vertex
       If (X [k]=0) then return;
        If (G [X [k-1], X [k]] ≠0) then
              For j=1 to k-1 do if (X [j]=X [k]) then break;  // Check for distinction.
                    If (j=k) then        //if true then the vertex is distinct.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

Warshall’s algorithm

Warshall's algorithm

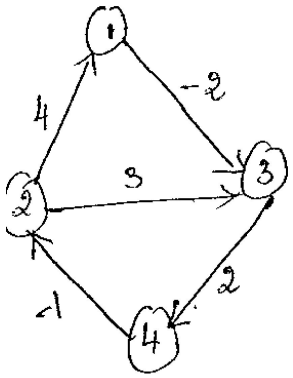
- Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.
- It will find the minimum distance for the given graph.

Find the minimum distance for the given graph



- If no edge then give alpha .
- But for (1,1) (2,2) (3,3) (4,4) then give 0.

Find the minimum distance for the given graph



dist [1][3] = -2

d [2][1] = 4

d [2][3] = 3

d [3][4] = 2

d [4][2] = -1

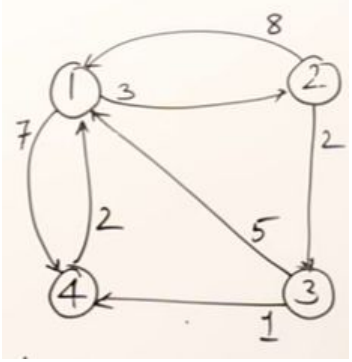
i.e

	1	2	3	4
1	0	∝	-2	∝
2	4	0	3	∝
3	∝	∝	0	2
4	∝	-1	∝	0

- If no edge then give alpha ∝
- But for (1,1) (2,2) (3,3) (4,4) then give 0.

How Floyd-Warshall Algorithm Works?

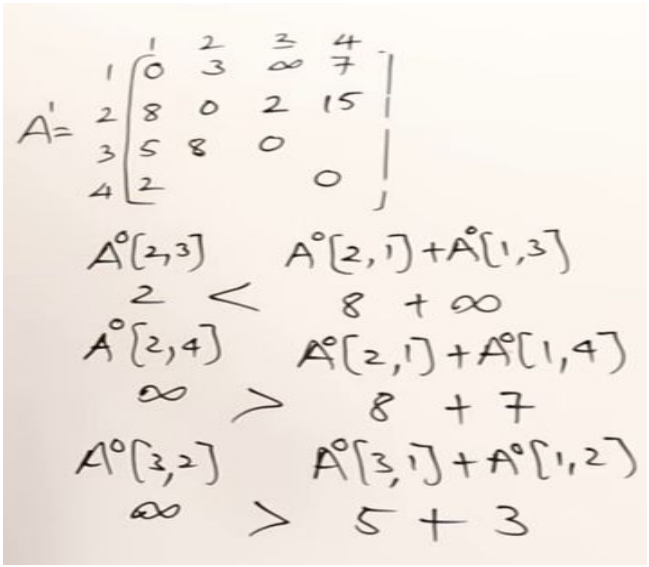
- Let the given graph be:

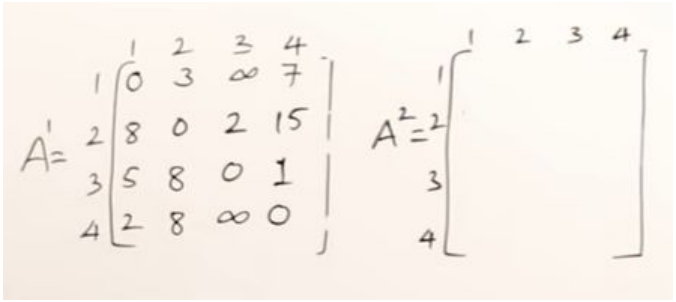


- Follow the steps below to find the shortest path between all the pairs of vertices.
  - Create a matrix A0 of dimension n*n where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph.

$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

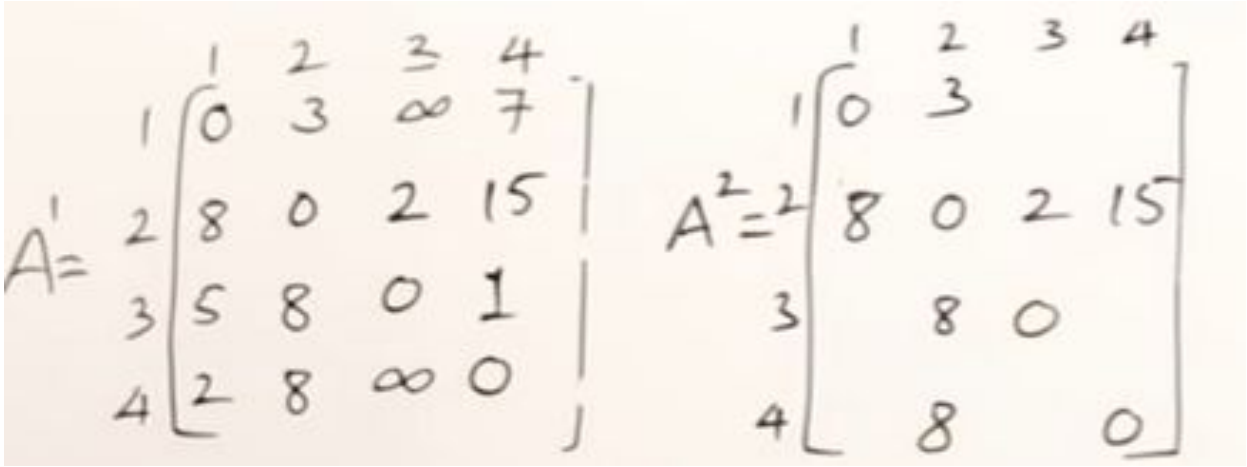
- Now, create a matrix A1 using matrix A0. The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way.
- Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. A[i][j] is filled with
- (A[i][k] + A[k][j]) if (A[i][j] > A[i][k] + A[k][j]).



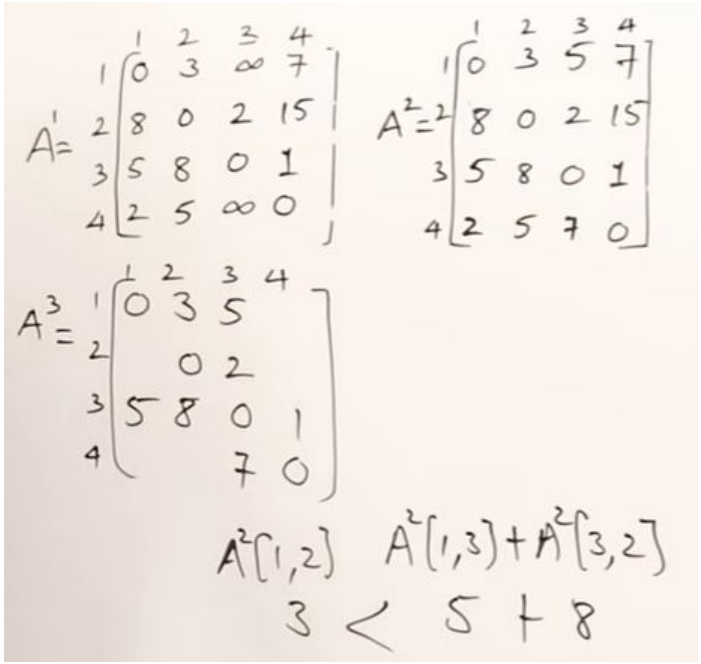


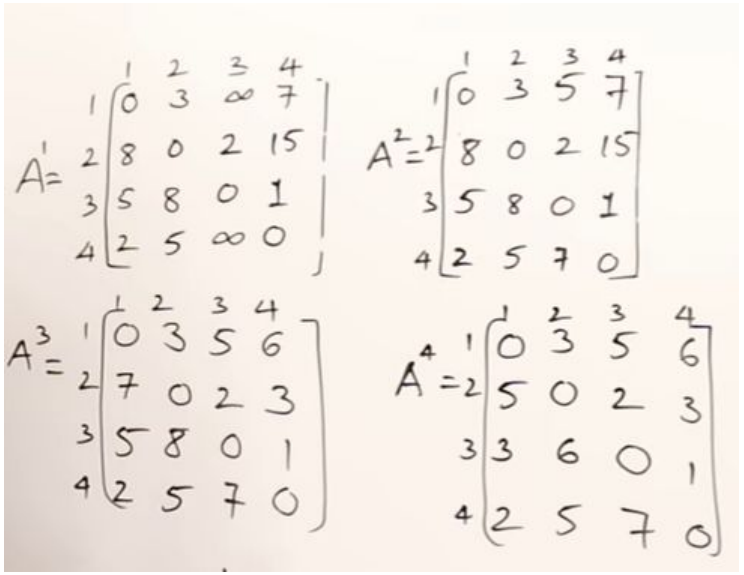
- Similarly, A2 is created using A1. The elements in the second column and the second row are left as they are.

- In this step, k is the second vertex (i.e. vertex 2). The remaining steps are the same as in step 2.



- Similarly, A3 and A4 is also created.





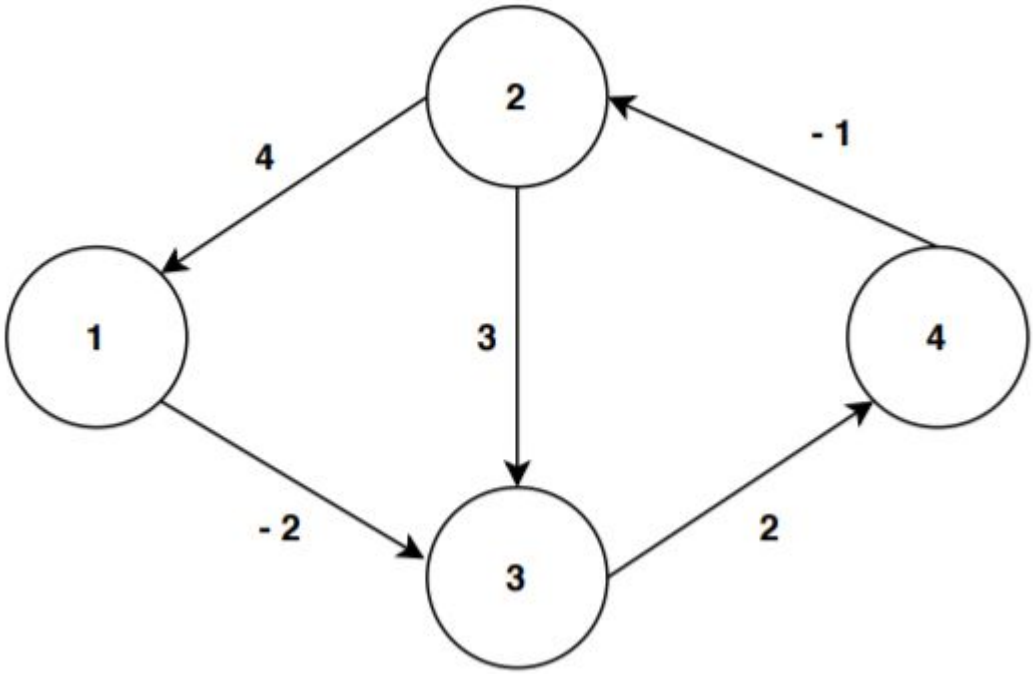
- A4 gives the shortest path between each pair of vertices.

Warshall's algorithm

- **Input –** The cost matrix of given Graph.
- **Output:** Matrix to for shortest path between any vertex to any vertex.

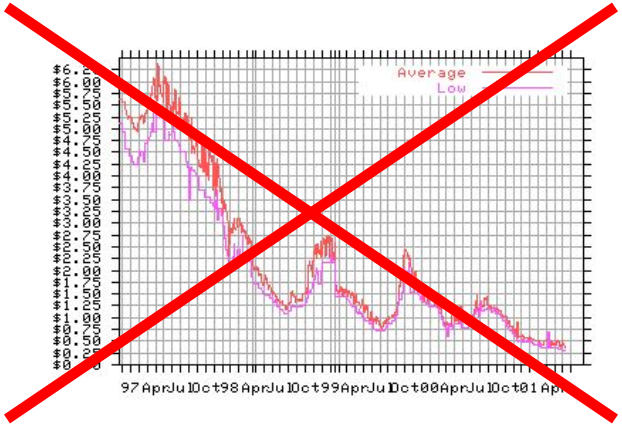
```
Begin
   for k := 0 to n, do
      for i := 0 to n, do
         for j := 0 to n, do
            if cost[i,k] + cost[k,j] < cost[i,j], then
               cost[i,j] := cost[i,k] + cost[k,j]
         done
      done
   done
   display the current cost matrix
End
```

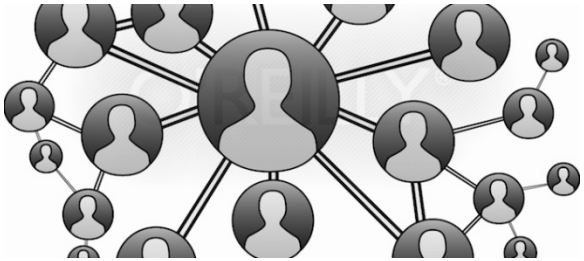
Solve using **Floyd-Warshall Algorithm**

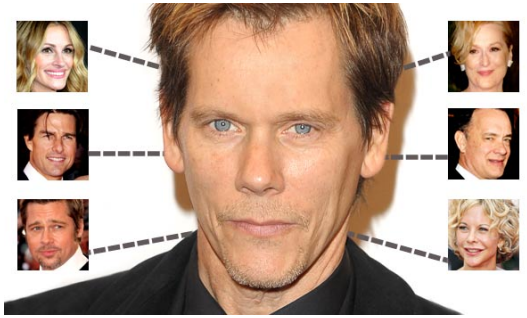


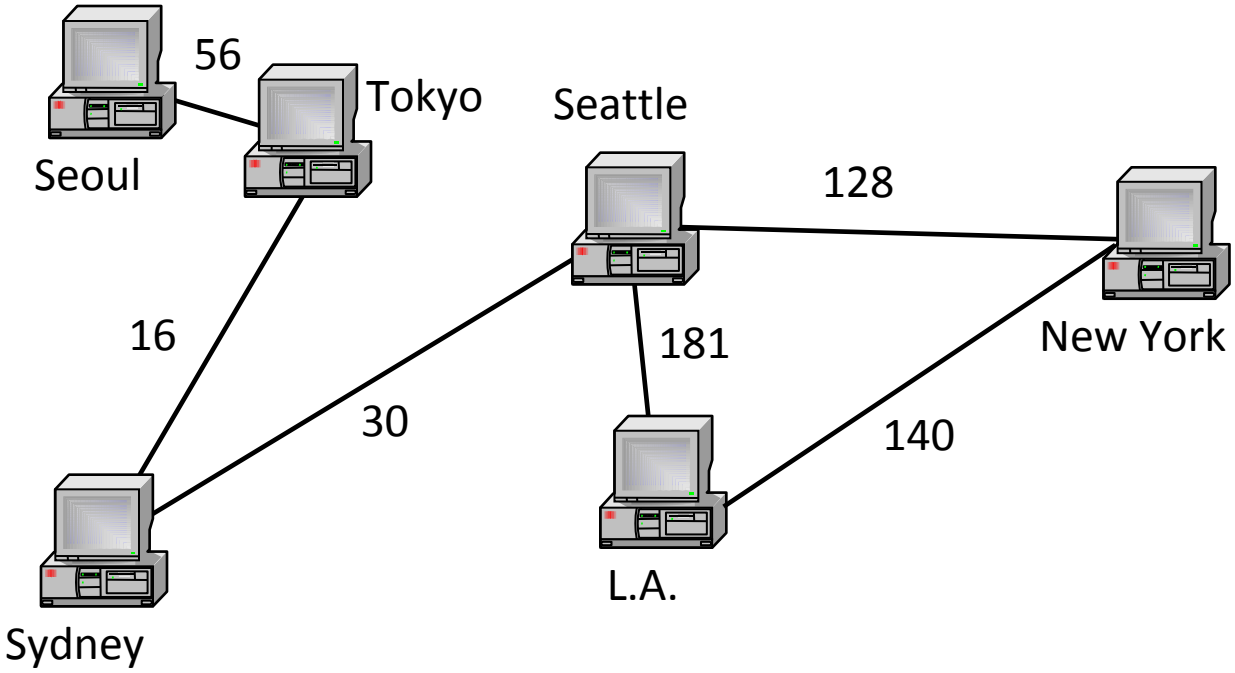
Graph

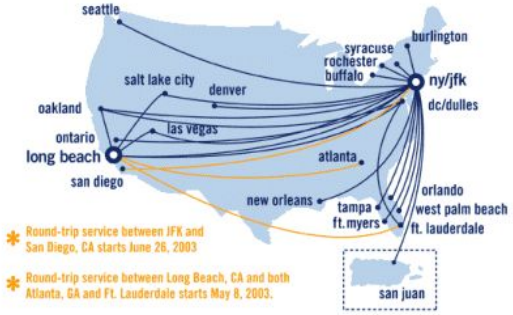
What is a graph?





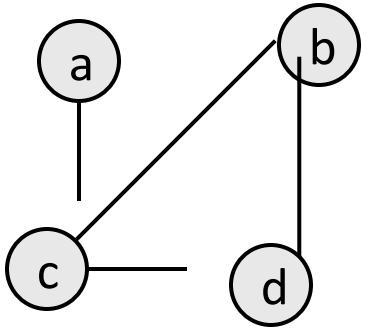






Graphs

- **graph**: A data structure containing:
  - a set of **vertices** *V*, *(sometimes called nodes)*
  - a set of **edges** *E*, where an edge represents a connection between 2 vertices.
    - Graph *G* = (*V*, *E*)
    - an edge is a pair (*v*, *w*) where *v*, *w* are in *V*

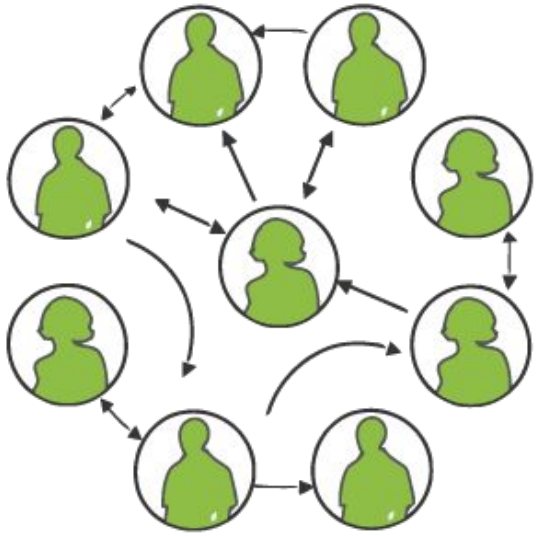


- the graph at right:
  - *V* = {a, b, c, d}
  - *E* = {(a, c), (b, c), (b, d), (c, d)}

- **degree**: number of edges touching a given vertex.
  - at right: a=1, b=2, c=3, d=2

Graph examples

- For each, what are the vertices and what are the edges?
  - Web pages with links
  - Methods in a program that call each other
  - Road maps (e.g., Google maps)
  - Airline routes
  - Facebook friends
  - Course pre-requisites
  - Family trees
  - Paths through a maze

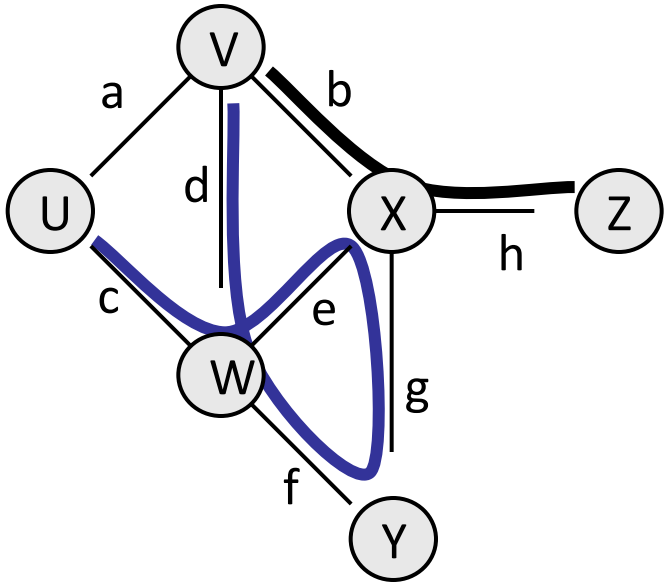


Paths

- **path**: A path from vertex *a* to *b* is a sequence of edges that can be followed starting from *a* to reach *b*.
  - can be represented as vertices visited, or edges taken
  - example, one path from *V* to *Z*: {b, h} or {V, X, Z}
  - What are two paths from U to Y?

- **path length**: Number of vertices or edges contained in the path.

- **neighbor** or **adjacent***:* Two vertices connected directly by an edge.
  - example: V and X

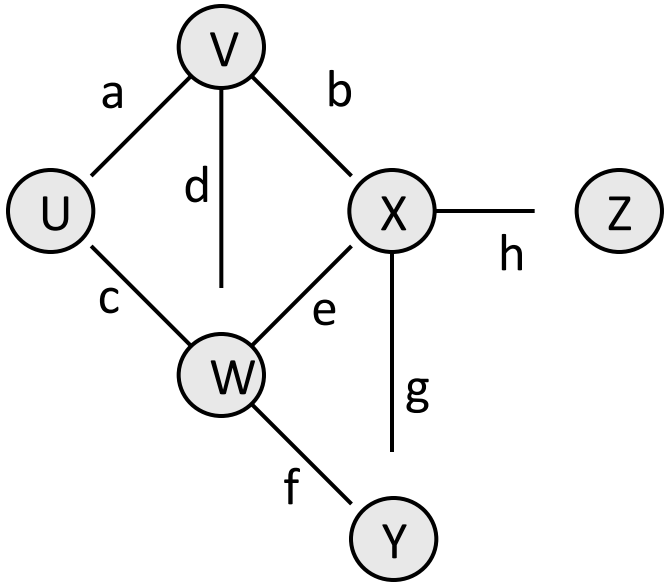


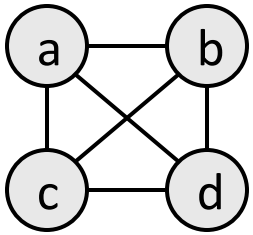
Reachability, connectedness

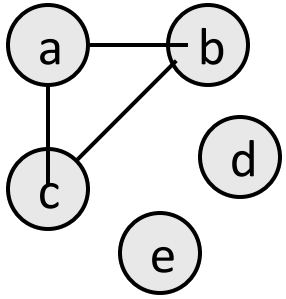
- **reachable**: Vertex *a* is *reachable* from *b* if a path exists from *a* to *b*.

- **connected**: A graph is *connected* if every vertex is reachable from any other.
  - Is the graph at top right connected?

- **strongly connected**: When every vertex has an edge to every other vertex.

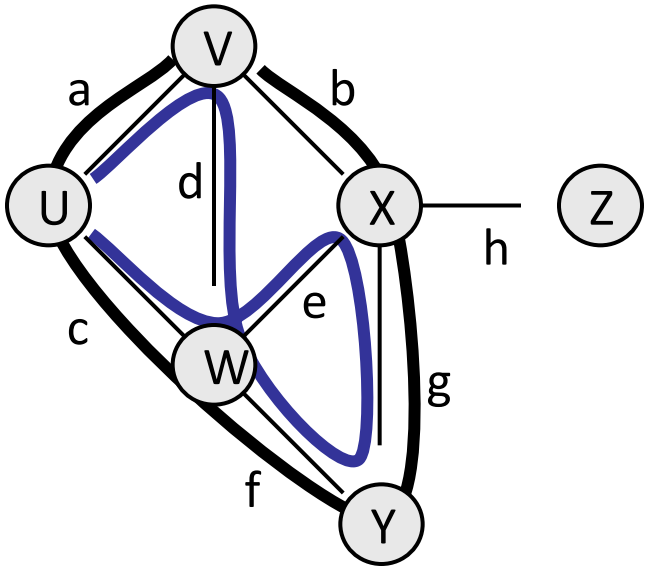






Loops and cycles

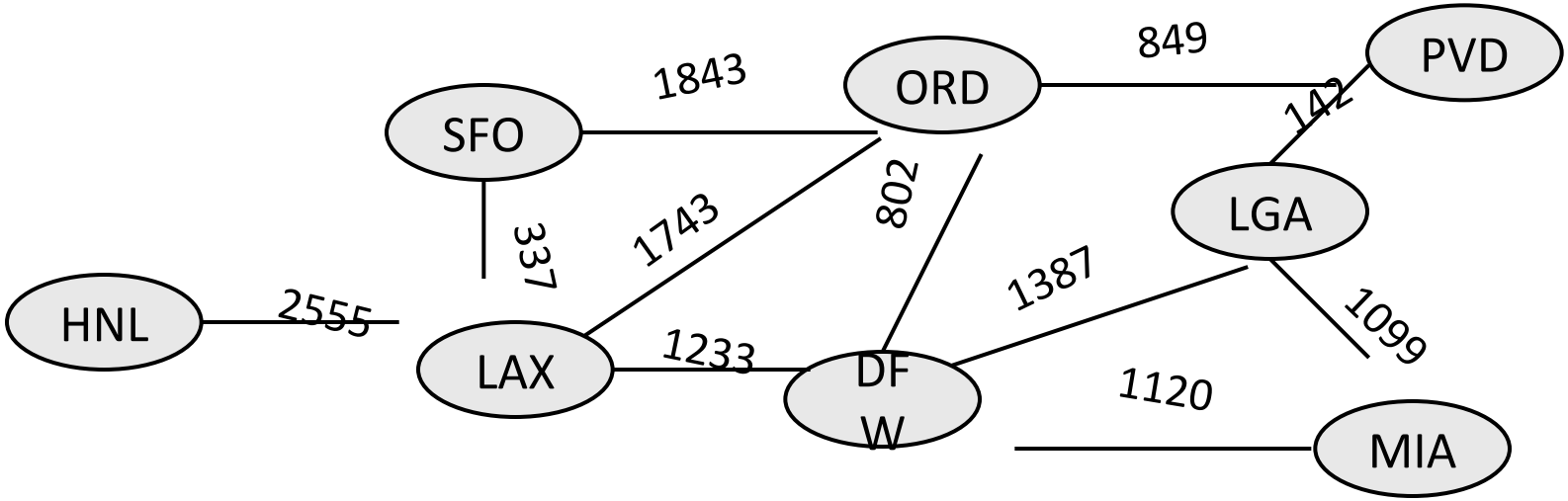
- **cycle**: A path that begins and ends at the same node.
  - example: {b, g, f, c, a} or {V, X, Y, W, U, V}.
  - example: {c, d, a} or {U, W, V, U}.
  - **acyclic graph**: One that does not contain any cycles.
- **loop**: An edge directly from a node to itself.
  - Many graphs don't allow loops.



Weighted graphs

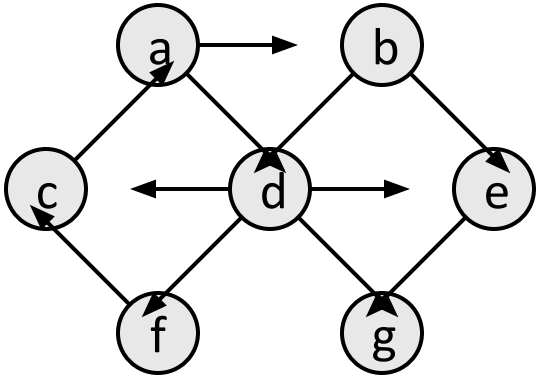
- **weight**: Cost associated with a given edge.
  - Some graphs have weighted edges, and some are unweighted.
  - Edges in an unweighted graph can be thought of as having equal weight (e.g. all 0, or all 1, etc.)
  - Most graphs do not allow negative weights.

- *example*: graph of airline flights, weighted by miles between cities:



Directed graphs

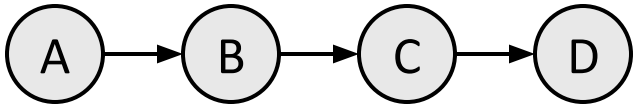
- **directed graph** ("digraph"): One where edges are *one-way* connections between vertices.
  - If graph is directed, a vertex has a separate in/out degree.
  - A digraph can be weighted or unweighted.
  - Is the graph below connected? Why or why not?

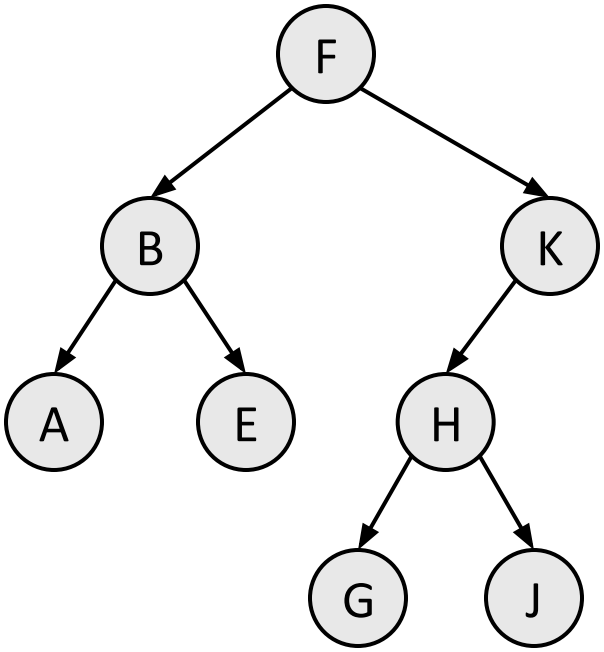


Linked Lists, Trees, Graphs

- A *binary tree* is a graph with some restrictions:
  - The tree is an unweighted, directed, acyclic graph (DAG).
  - Each node's in-degree is at most 1, and out-degree is at most 2.
  - There is exactly one path from the root to every node.

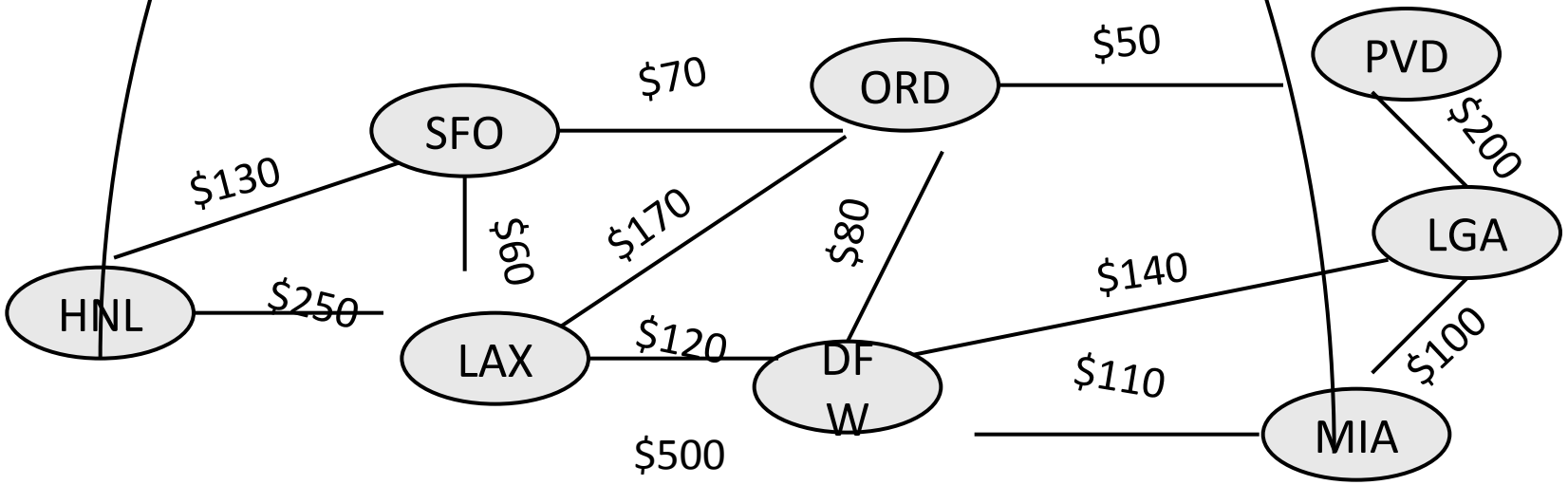
- A *linked list* is also a graph:
  - Unweighted DAG.
  - In/out degree of at most 1 for all nodes.





Searching for paths

- Searching for a path from one vertex to another:
  - Sometimes, we just want *any* path (or want to know there *is* a path).
  - Sometimes, we want to minimize path *length* (# of edges).
  - Sometimes, we want to minimize path *cost* (sum of edge weights).
- What is the shortest path from MIA to SFO? Which path has the minimum cost?



Representation of Graph

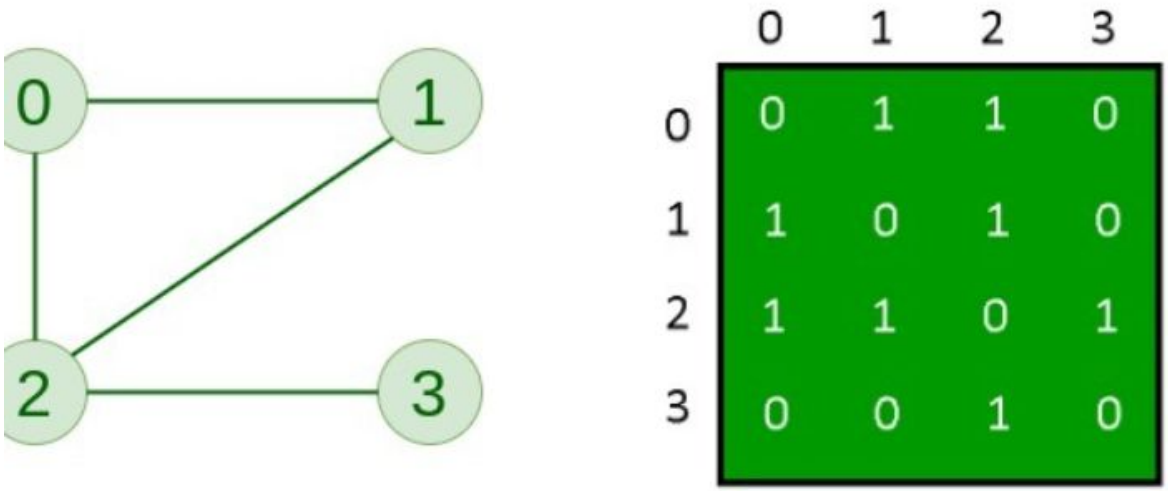
There are two ways to store a graph:

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

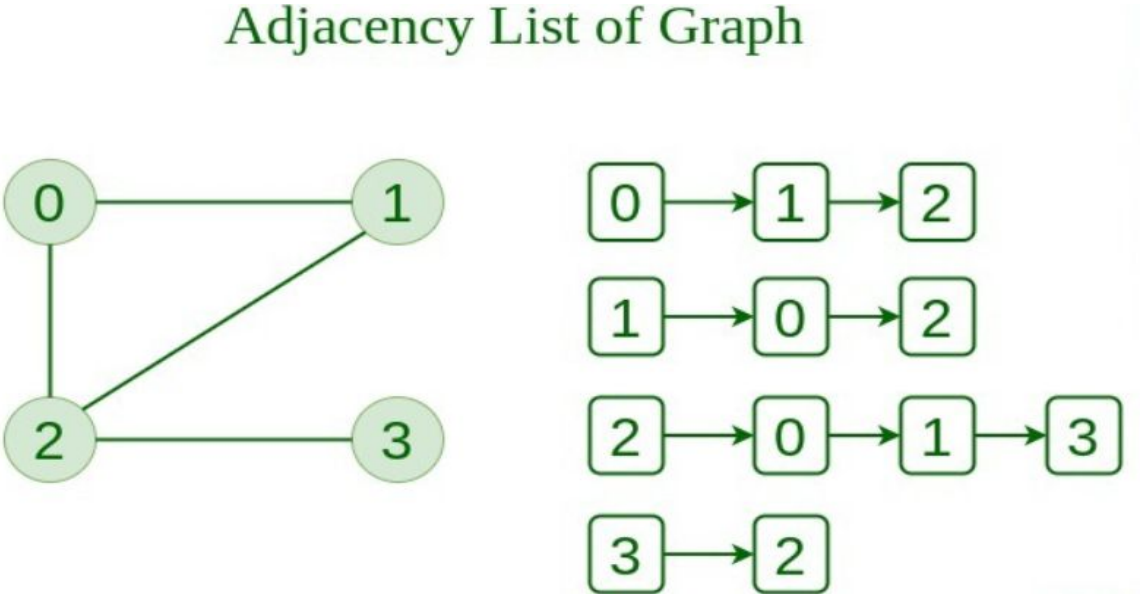
- In this method, the graph is stored in **the form of the 2D matrix** where rows and column denote vertices.

Adjacency Matrix of Graph



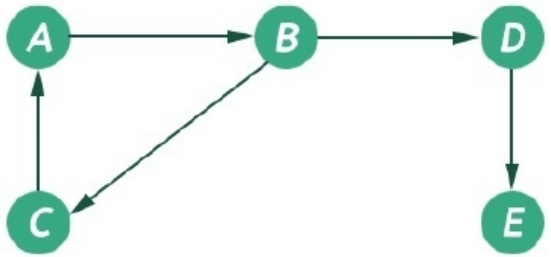
Adjacency List

- This graph is represented as a **collection of linked lists**.
  There is an array of pointer which points to the edges connected to that vertex



Traversing a Graph

- The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be directed or undirected. This graph can be represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})



- The graph has two types of traversal algorithms.
- These are called the
  - Breadth First Search and
  - Depth First Search.

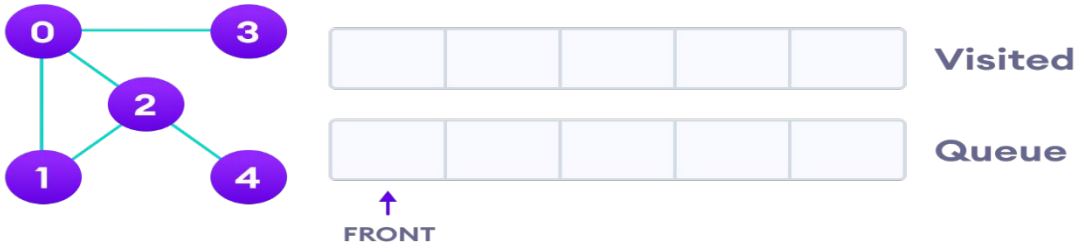
Breadth-first search

- Here we use **Queue** data structure.
- The traversal start from source node, its visit all the children of the root, then start with first then visits all its children, this process will repeat until no more element to visit.
- A standard BFS implementation puts each vertex of the graph into one of two categories:

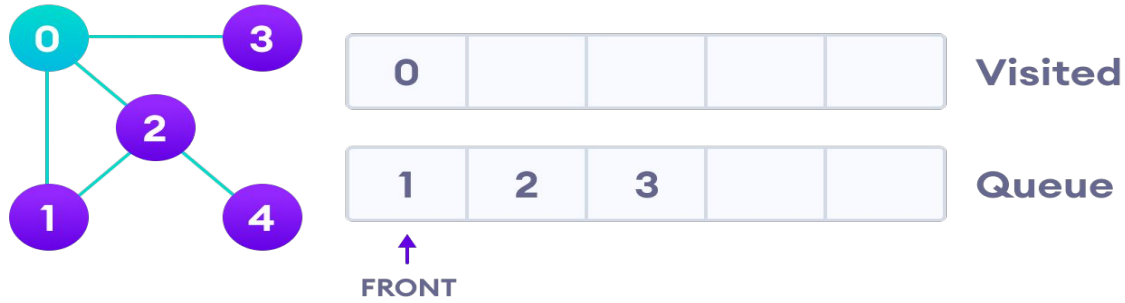
1. Visited
2. Not Visited

Breadth-first search (BFS)

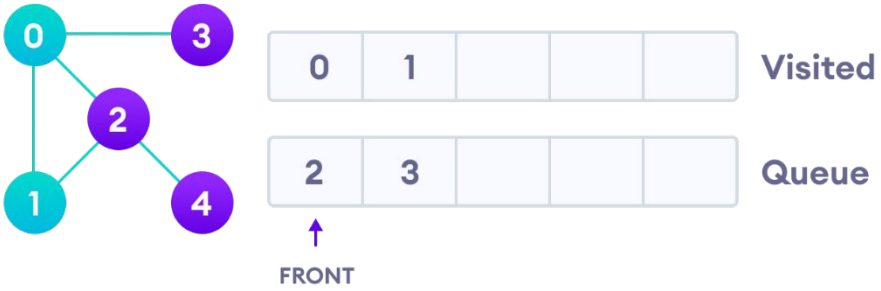
- Let's see how the Breadth First Search algorithm works with an example. We use an undirected graph with 5 vertices.



- We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the Queue.

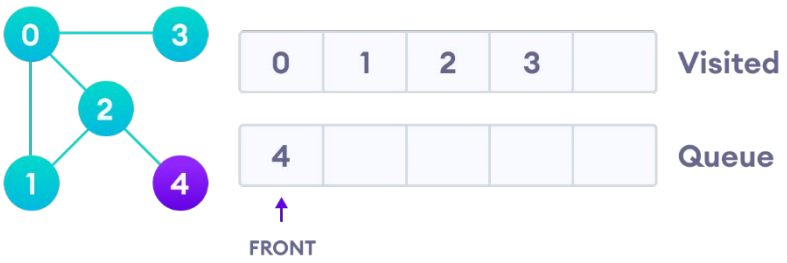


- Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.

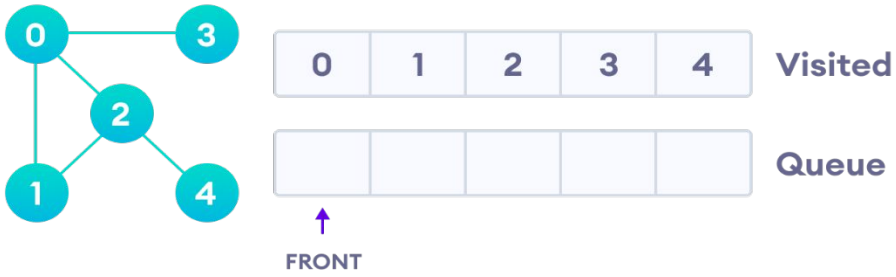


- Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.

- 4 remains in the queue



- Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited. We visit it.



- Since the queue is empty, we have completed the Breadth First Traversal of the graph.

Breadth-first search (BFS) - Algorithm

Algorithm:

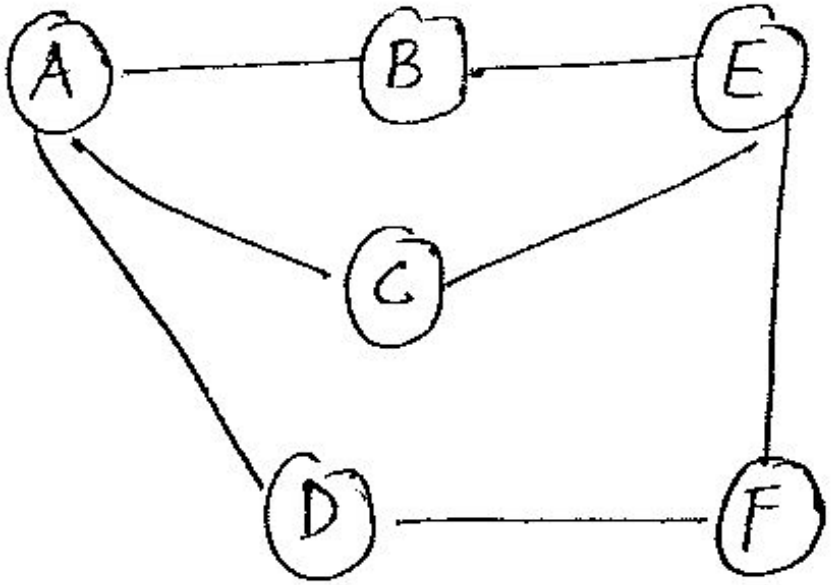
1. All nodes are initialized to ready state and queue is initially empty.
2. Begin with source node (or any node) which is in ready state and put into queue. Mark the status of that node to waiting.
3. while queue is not empty do
   begin
4. Delete first node K from queue and Mark the status of it as visited.
5. Add all adjacent nodes of K which are in ready state to the rear side of the queue and mark the status of those nodes to waiting.

   end
6. If the graph still contains nodes which are in ready state then goto step 2
7. Return.

Algorithm

```
BFS (G, s)                    //Where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s )            //Inserting s in queue until all its neighbour vertices
are marked.
    mark s as visited.
    while ( Q is not empty) //Removing that vertex from queue,whose
  neighbour will be visited now
    v = Q.dequeue( )          //processing all the neighbours of v
    for all neighbours w of v in Graph G
    if w is not visited
            Q.enqueue( w )  //Stores w in Q to further visit its neighbour
            mark w as visited.
```

Traverse the below graph using BFS



step1: Mask all nodes to ready state S

A	B	C	D	E	F
0	0	0	0	0	0

step2: Traverse A and insert it into queue. [A]

step3: Delete A, change its state to 2 and insert all its neighbours to queue, and change the state from ready to waiting.

A [B | C | D]

step4: delete B, change state to 2, insert all its neighbour to queue → change state to 1 (waiting)

A, B [C | D | E]

step5: delete C, change state to 2, insert all its neighbours, change state to 1 (waiting)

A, B, C [D | E]

step6: delete D, change D's state to 2, insert all its neighbours, change state to 1 (waiting)

A, B, C, D [E | F]

step7: delete E, change its state from 1 to 2, since no neighbours no need to insert

A, B, C, D, E [F]

step8: delete F

A, B, C, D, E, F [ ]

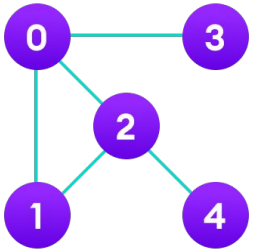
Depth First Search

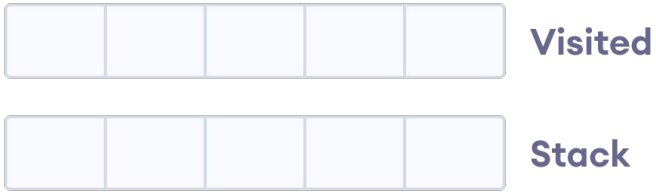
- Here we use **stack** data structure.
- The traversal start from source node and goes till all the successor of the first neighbor, then continues with the remaining nodes and so on.
- A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

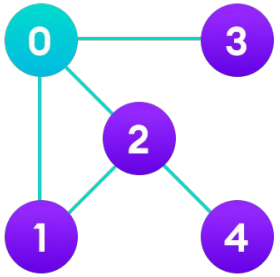
Depth First Search Example (DFS)

- Let's see how the Depth First Search algorithm works with an example. We use an undirected graph with 5 vertices.

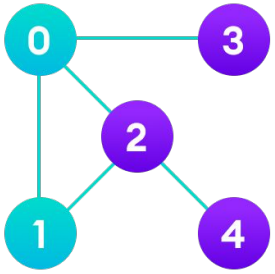




- We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the stack.

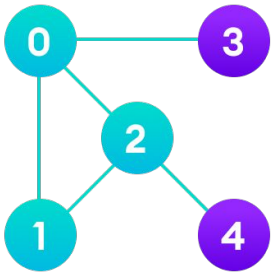


- Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes. Since 0 has already been visited, we visit 2 instead.



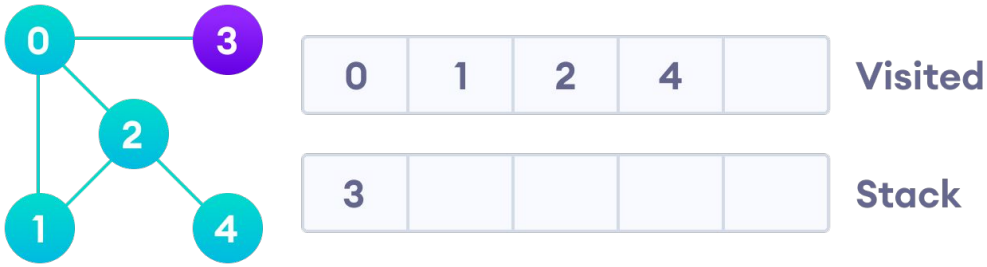
0	1				Visited
2	3				Stack

- Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.

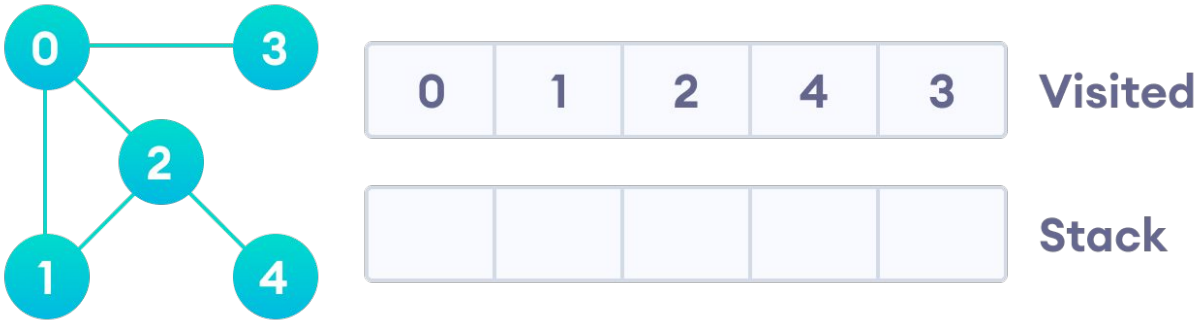


0	1	2			Visited
4	3				Stack

- Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



- After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



Depth-first search (DFS) - Algorithm

Algorithm:

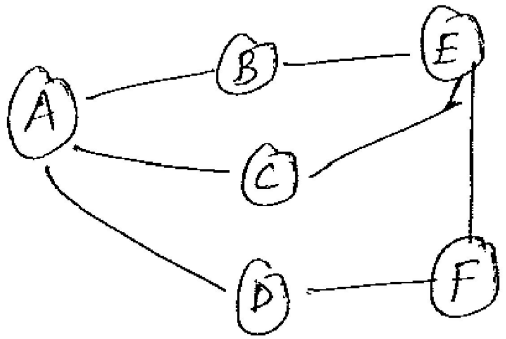
1. All nodes are initialized to ready state (0) and stack is initially empty.
2. Begin with source node which is in ready state and push it into stack. change its state to waiting (1)
3. while stack is not empty do
   begin
4. pop the top node K of stack and process it. and change the state to visited (2).
5. push all adjacent nodes of K which are in ready state into stack and mark the status of those nodes end to waiting.
6. If the graph still contains nodes which are in ready state then goto step 2.

Algorithm

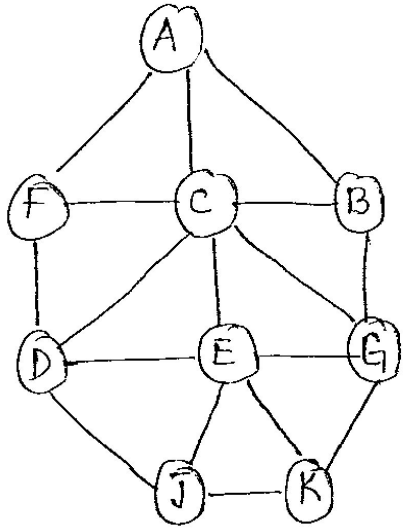
```
DFS-iterative (G, s):    //Where G is graph and s is source vertex
    let S be stack
    S.push( s ) //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
            //Pop a vertex from stack to visit next
            v = S.top( )
            S.pop( )
            //Push all the neighbours of v in stack that are not visited
            for all neighbours w of v in Graph G :
            if w is not visited :
```

```
            S.push( w )
        mark w as visited
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
        DFS-recursive(G, w)
```

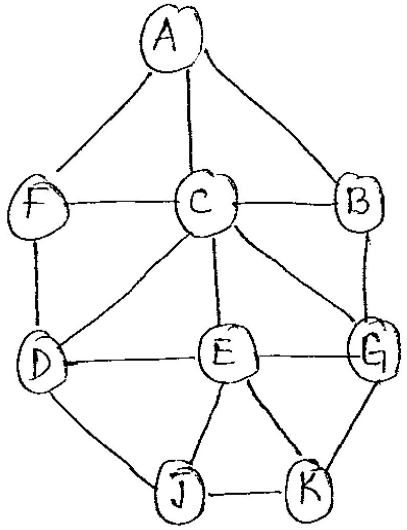
Traverse the below graph using DFS



Solve BFS and DFS Or the below graph



Solve BFS and DFS Or the below graph



Answers:

BFS: A F C B D E G J K

DFS: A F D J K G E C B

Quiz

Which of the following algorithms can be used to most efficiently determine the presence of a cycle in a given graph ?

- [ ] Depth First Search
- [ ] Breadth First Search
- [ ] Prim\'s Minimum Spanning Tree Algorithm
- [ ] Kruskal\' Minimum Spanning Tree Algorithm

Which of the following algorithms can be used to most efficiently determine the presence of a cycle in a given graph ?

- [x] Depth First Search
- [ ] Breadth First Search
- [ ] Prim\'s Minimum Spanning Tree Algorithm
- [ ] Kruskal\' Minimum Spanning Tree Algorithm

Traversal of a graph is different from tree because

- [ ] There can be a loop in graph so we must maintain a visited flag for every vertex
- [ ] DFS of a graph uses stack, but inorder traversal of a tree is recursive
- [ ] BFS of a graph uses queue, but a time efficient BFS of a tree is recursive.
- [ ] All of the above

Traversal of a graph is different from tree because

- [x] There can be a loop in graph so we must maintain a visited flag for every vertex
- [ ] DFS of a graph uses stack, but inorder traversal of a tree is recursive
- [ ] BFS of a graph uses queue, but a time efficient BFS of a tree is recursive.
- [ ] All of the above

What are the appropriate data structures for following algorithms?

1) Breadth-First Search

2) Depth First Search

3) Prim's Minimum Spanning Tree

4) Kruskal' Minimum Spanning Tree

- [ ]
  - Stack
  - Queue
  - Priority Queue
  - Union Find

- [ ]
  - Queue
  - Stack
  - Priority Queue
  - Union Find

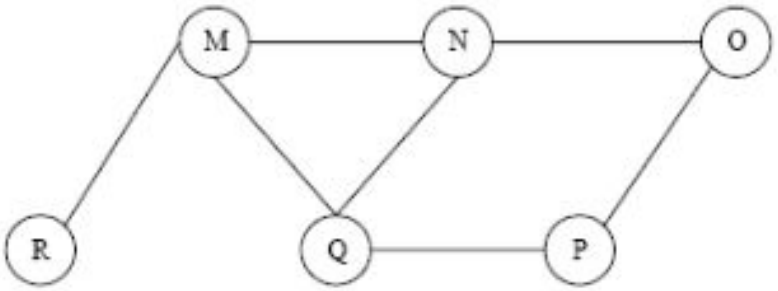
- [ ]
  - Stack
  - Queue
  - Union Find
  - Priority Queue

What are the appropriate data structures for following algorithms?

1) Breadth-First Search
2) Depth First Search
3) Prim's Minimum Spanning Tree
4) Kruskal' Minimum Spanning Tree

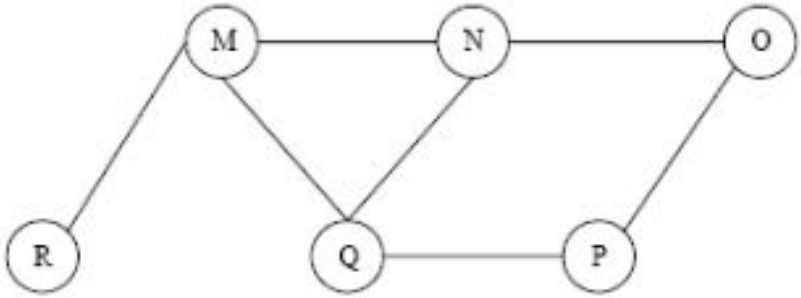
- [ ] Stack, Queue, Priority Queue, Union Find
- [x] Queue, Stack, Priority Queue, Union Find
- [ ] Stack, Queue, Union Find, Priority Queue

The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is



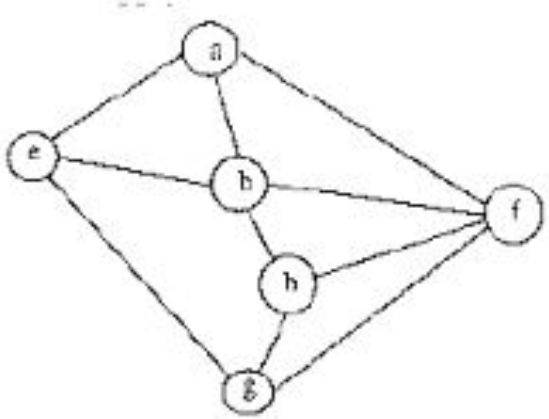
- [ ] MNOPQR
- [ ] NQMPOR
- [ ] QMNPRO
- [ ] QMNPOR

The Breadth First Search algorithm has been implemented using the queue data structure. One possible order of visiting the nodes of the following graph is



- [ ] MNOPQR
- [ ] NQMPOR
- [x] QMNPRO
- [ ] QMNPOR

Consider the following graph,



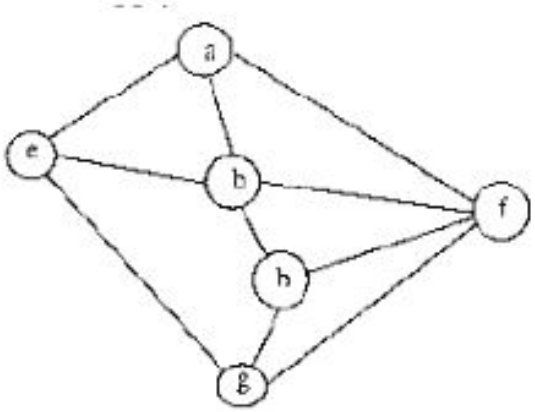
Among the following sequences:

(I) a b e g h f
(II) a b f e h g
(III) a b f h g e
(IV) a f g h b e

Which are depth first traversals of the above graph?

- [ ] I, II and IV only
- [ ] I and IV only
- [ ] II, III and IV only
- [ ] I, III and IV only

Consider the following graph,



Among the following sequences:

(I) a b e g h f
(II) a b f e h g
(III) a b f h g e
(IV) a f g h b e

Which are depth first traversals of the above graph?

- [ ] I, II and IV only
- [ ] I and IV only
- [ ] II, III and IV only
- [x] I, III and IV only

Given two vertices in a graph s and t, which of the two traversals (BFS and DFS) can be used to find if there is path from s to t?

- [ ] Only BFS
- [ ] Only DFS
- [ ] Both BFS and DFS
- [ ] Neither BFS nor DFS

Given two vertices in a graph s and t, which of the two traversals (BFS and DFS) can be used to find if there is path from s to t?

- [ ] Only BFS
- [ ] Only DFS
- [x] Both BFS and DFS
- [ ] Neither BFS nor DFS

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

BRANCH AND BOUND
-knapsack

0/1 Knapsack Problem

Given two integer arrays val[0..n-1] and wt[0..n-1] that represent values and weights associated with n items respectively. Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W. We have ‘n’ items with value v1 , v2 . . . vn and weight of the corresponding items is w1 , w2 . . . Wn . Max capacity is W . We can either choose or not choose an item. We have x1 , x2 . . . xn. Here xi = { 1 , 0 }. xi = 1 , item chosen xi = 0 , item not chosen

Different approaches of this problem :

- ✔ Dynamic programming
- ✔ Brute force
- ✔ Backtracking
- ✔ Branch and bound

ALGORITHM

- Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
- Initialize maximum profit, maxProfit = 0
- Create an empty queue, Q.
- Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.

Do following while Q is not empty

- Extract an item from Q. Let the extracted item be u. Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
- Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

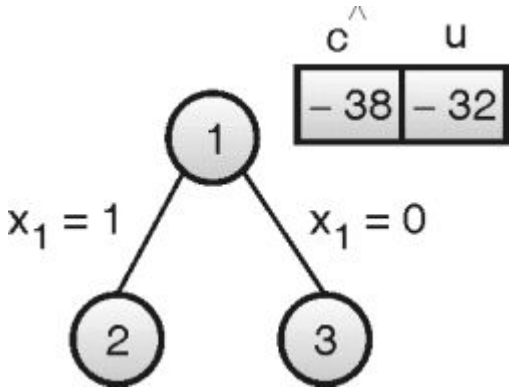
Algorithm for maxProfit :

```
knapsack(int W, Item arr[], int n)
    queue<Node> Q
    Node u, v                          //u.level = -1
    Q.push(u)                          //u.profit = u.weight = 0
    while ( !Q.empty() )               //int maxProfit = 0
        u = Q.front()  &  Q.pop()
        v.level = u.level + 1                  // selecting the item
        v.weight = u.weight + arr[v.level].weight
        v.profit = u.profit + arr[v.level].value
        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit
        v.bound = bound(v, n, W, arr)
        if (v.bound > maxProfit)
            Q.push(v)
        v.weight = u.weight                    // not selecting the item
        v.profit = u.profit
        v.bound = bound(v, n, W, arr)
        If (v.bound > maxProfit)
            Q.push(v)
     return (maxProfit)
```

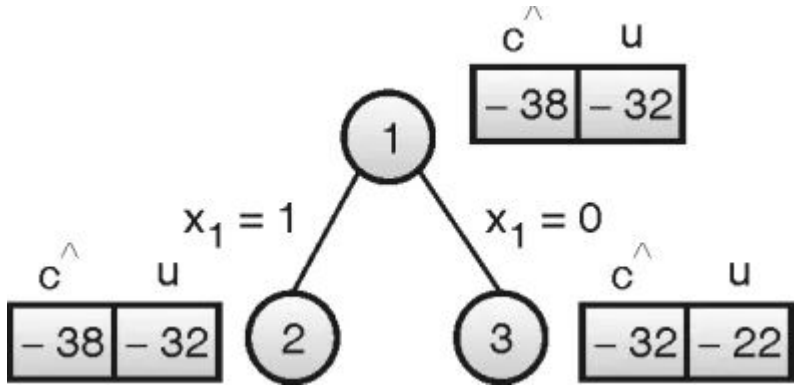
Example: Solve the following instance of knapsack using LCBB for knapsack capacity M = 15.

i	P_i	W_i
1	10	2
2	10	4
3	12	6
4	18	9

State Space Tree:

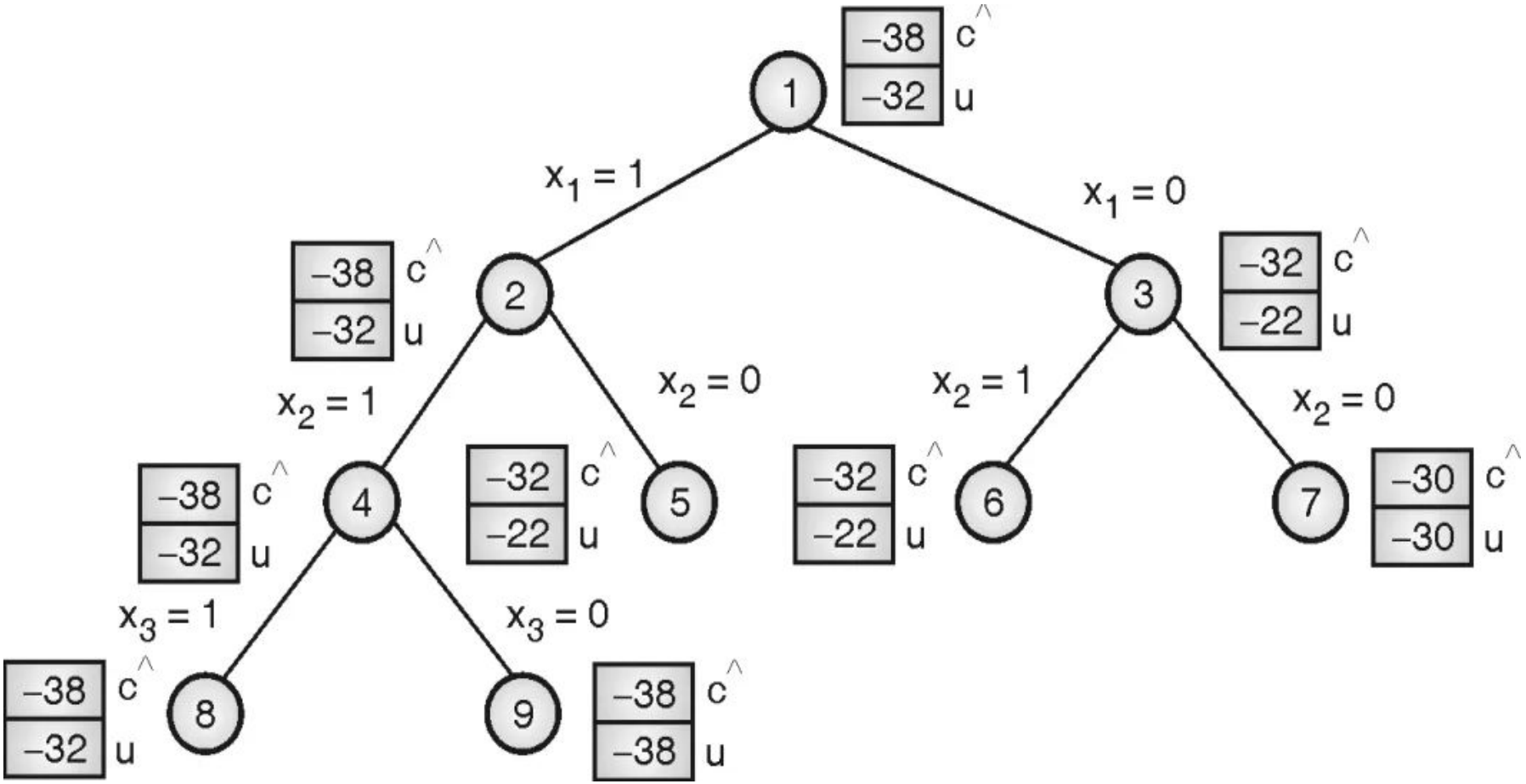


- Find
  - **Node 2 : Inclusion of item 1 at node 1**
  - **Node 3 : Exclusion of item 1 at node 1**



State Space Tree:

- Find
  - **Node 4 : Inclusion of item 2 at node 2**
  - **Node 4 : Inclusion of item 2 at node 2**

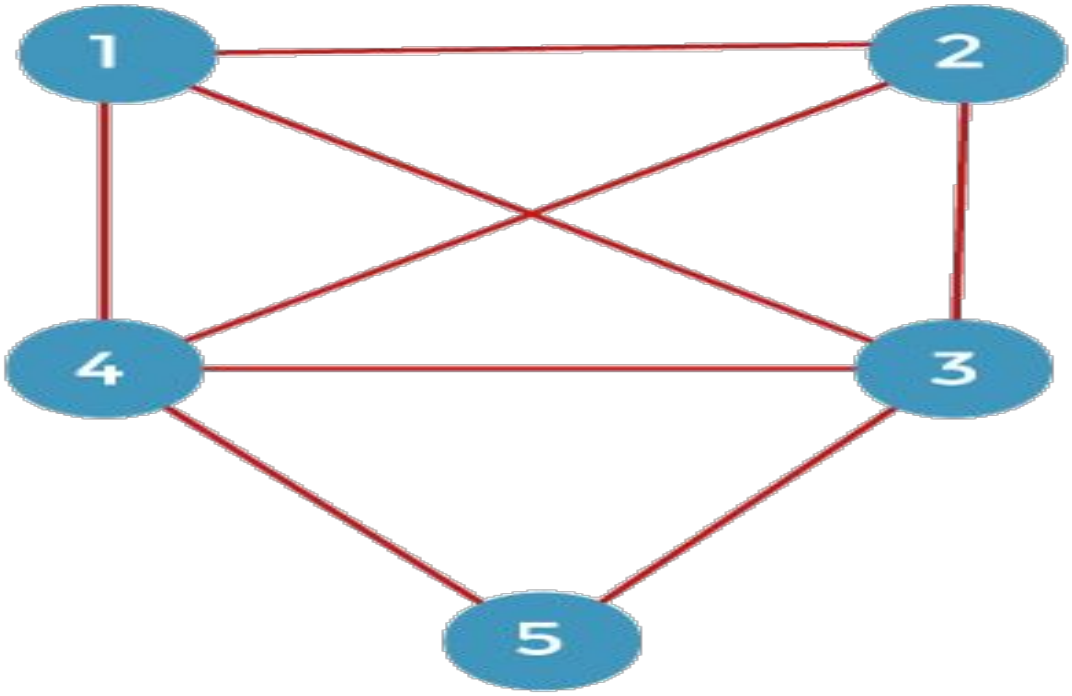


Travelling Salesman Problem

- In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).
- We can model the cities as a complete graph of n vertices, where each vertex represents a city.
- Traveling Salesperson problem using branch and bound
  - Given the vertices, the problem here is that we have to travel each vertex exactly once and reach back to the starting point.

Travelling Salesman Problem

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

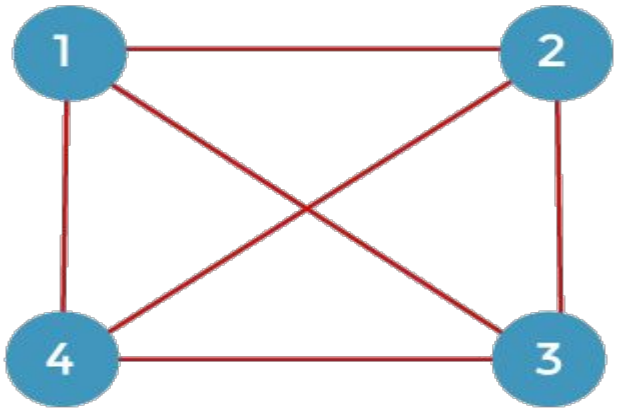


Travelling Salesman Problem

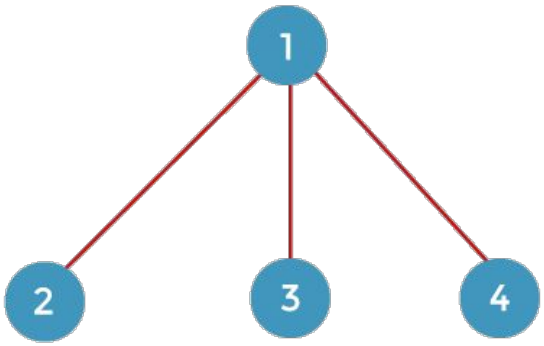
- As we can observe in the above graph that there are 5 vertices given in the graph. We have to find the shortest path that goes through all the vertices once and returns back to the starting vertex. We mainly consider the starting vertex as 1, then traverse through the vertices 2, 3, 4, and 5, and finally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

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

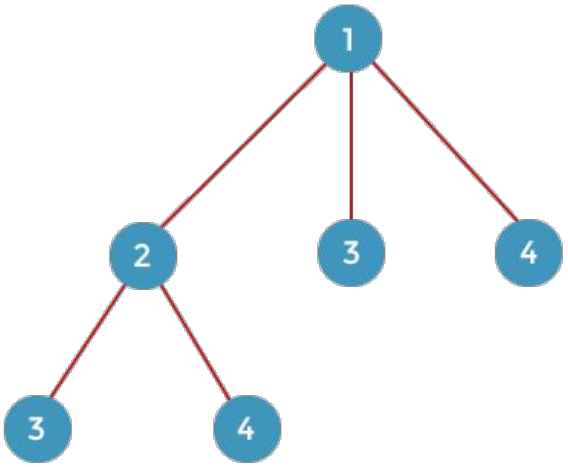
- **Let's first understand the approach then we solve the above problem.**
- The graph is given below, which has four vertices:



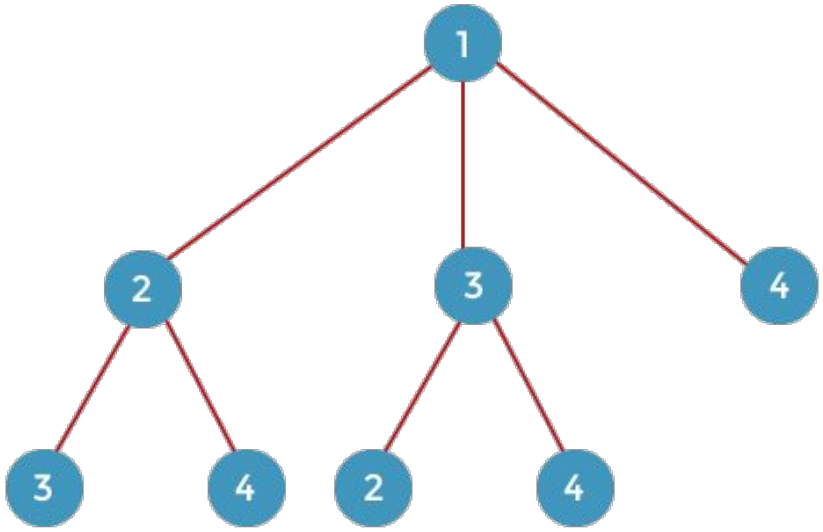
- Suppose we start travelling from vertex 1 and return back to vertex 1. There are various ways to travel through all the vertices and returns to vertex 1. We require some tools that can be used to minimize the overall cost.
- To solve this problem, we make a state space tree. From the starting vertex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



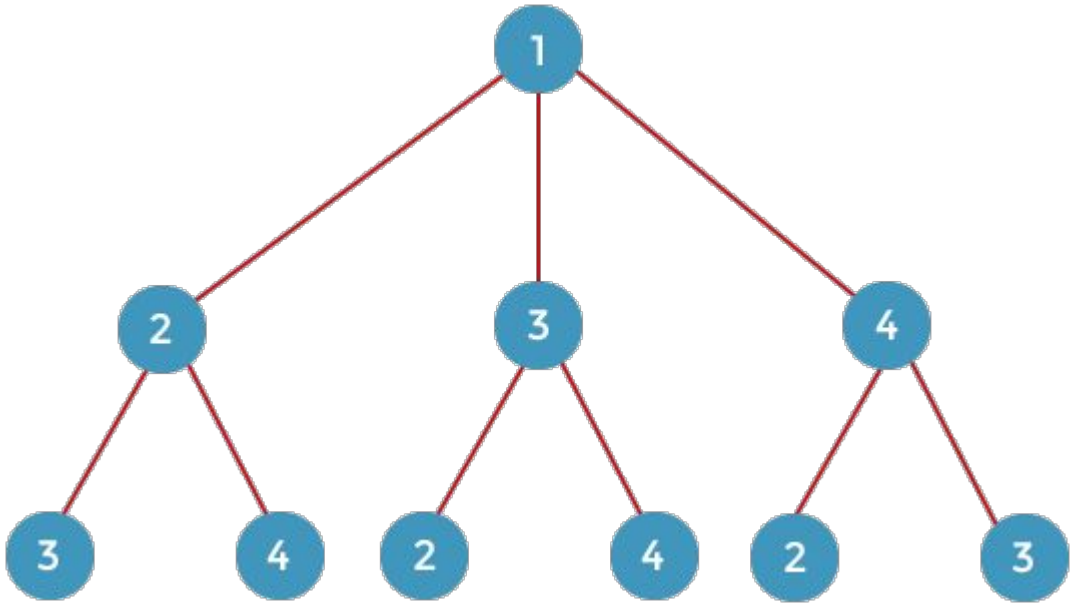
- From vertex 2, we can go either to vertex 3 or 4. If we consider vertex 3, we move to the remaining vertex, i.e., 4. If we consider the vertex 4 shown in the below diagram:



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



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



- The above is the complete state space tree. The state space tree shows all the possibilities. Backtracking and branch n bound both use the state space tree, but their approach to solve the problem is different. Branch n bound is a better approach than backtracking as it is more efficient.

Travelling Salesman Problem

Difference between Backtracking and Branch and Bound

Back tracking	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 function	It involves bounding function
It is more often not applied to non-optimization.	It can be applied only to optimization problems.