SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF INFORMATION TECHNOLOGY

UNIT - V

Design and Analysis of Algorithm – SCSA1403

Backtracking and Branch and Bound **9 Hrs.**

Backtracking:- 8 Queens - Hamiltonian Circuit Problem - Branch and Bound - Assignment Problem - Knapsack Problem:- Travelling Salesman Problem - NP Complete Problems - Clique Problem - Vertex Cover Problem.

Backtracking

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

8 Queens Problem

You are given an 8x8 chessboard; find a way to place 8 queens such that no queen can attack any other queen on the chessboard. A queen can only be attacked if it lies on the same row, or same column, or the same diagonal of any other queen. Print all the possible configurations.

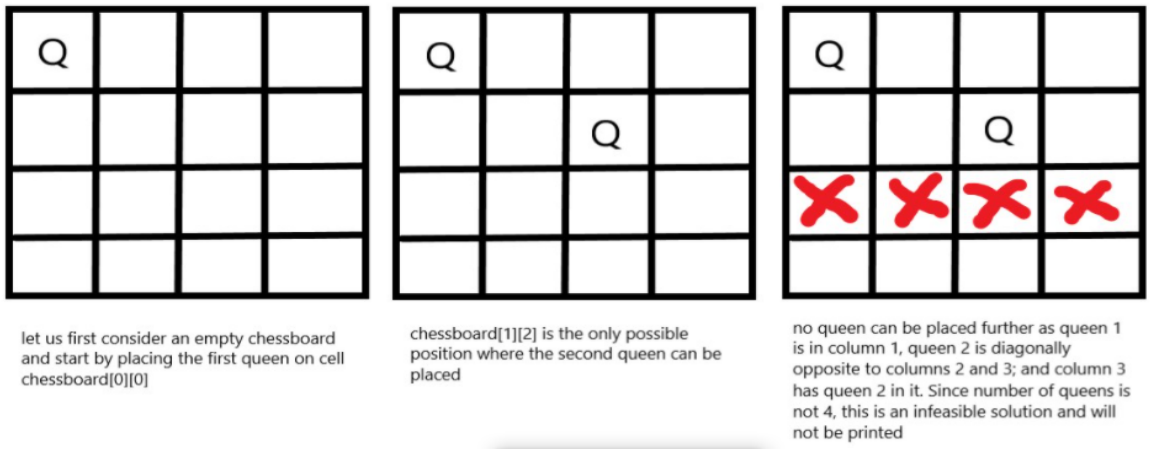
To solve this problem, we will make use of the Backtracking algorithm. The backtracking algorithm, in general checks all possible configurations and test whether the required result is obtained or not. For this given problem, we will explore all possible positions the queens can be relatively placed at. The solution will be correct when the number of placed queens = 8. The time complexity of this approach is O(N!).

Input - the number 8, which does not need to be read, but we will take an input number for the sake of generalization of the algorithm to an NxN chessboard.

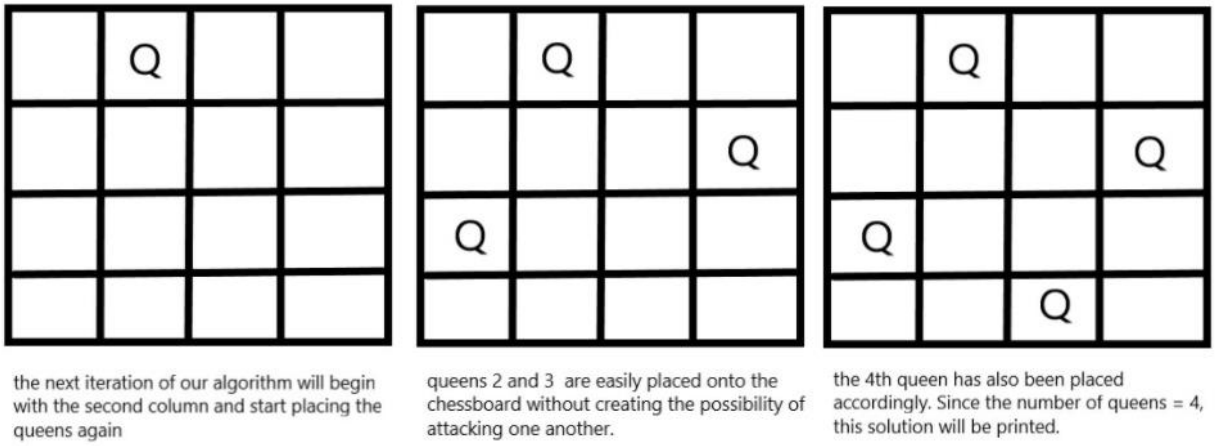
Output - all matrices that constitute the possible solutions will contain the numbers 0(for empty cell) and 1(for a cell where queen is placed). Hence, the output is a set of binary matrices.

Visualization from a 4x4 chessboard solution:

In this configuration, we place 2 queens in the first iteration and see that checking by placing further queens is not required as we will not get a solution in this path. Note that in this configuration, all places in the third rows can be attacked.



As the above combination was not possible, we will go back and go for the next iteration. This means we will change the position of the second queen.



In this, we found a solution. Now let's take a look at the backtracking algorithm and see how it works: The idea is to place the queen's one after the other in columns, and check if previously placed queens cannot attack the current queen we're about to place. If we find such a row, we return true and put the row and column as part of the solution matrix. If such a column does not exist, we return false and backtrack.

Pseudo code:

START

1. Begin from the leftmost column
2. If all the queens are place return true/ print configuration
3. Check for all rows in the current column

a) if queen placed safely, mark row and column; and recursively check if we approach in the current configuration, do we obtain a solution or not

b) if placing yields a solution, return true

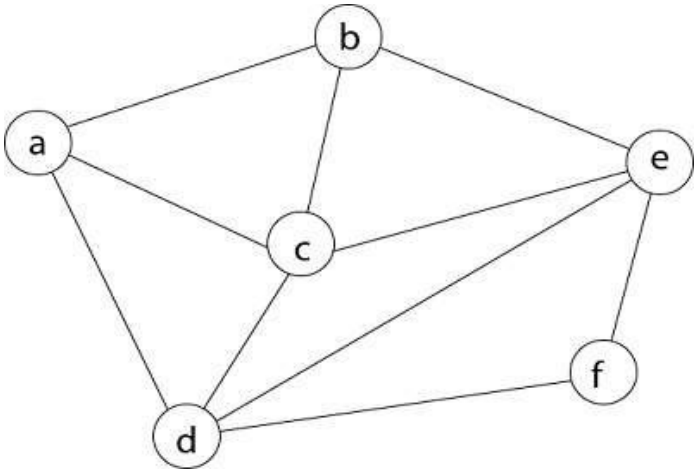
c) if placing does not yield a solution, unmark and try other rows

4. if all rows tried and solution not obtained, return false and backtrack

END

Hamiltonian Circuit Problem

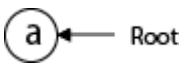
Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach. We start our search from any arbitrary vertex say 'a'. This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected by alphabetical



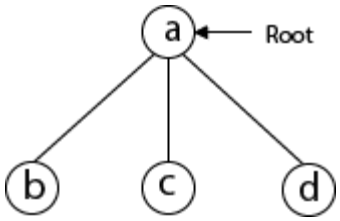
order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that dead end is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian Cycle is obtained.

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

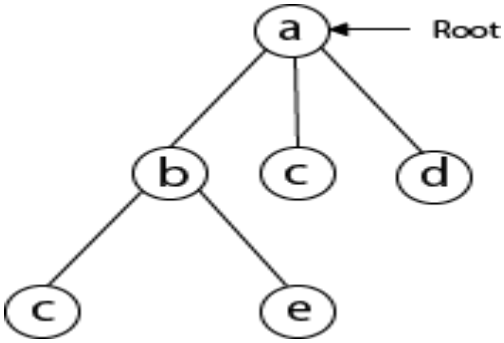
Solution: Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.



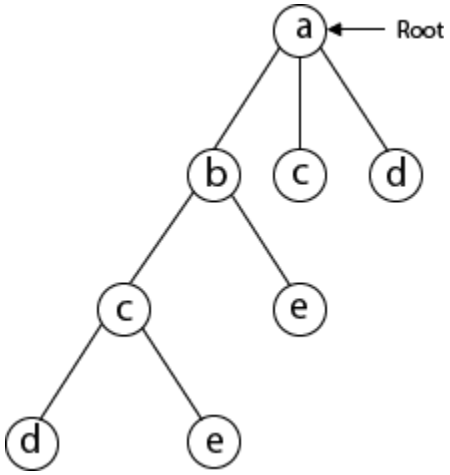
Next, we choose vertex 'b' adjacent to 'a' as it comes first in lexicographical order (b, c, d).



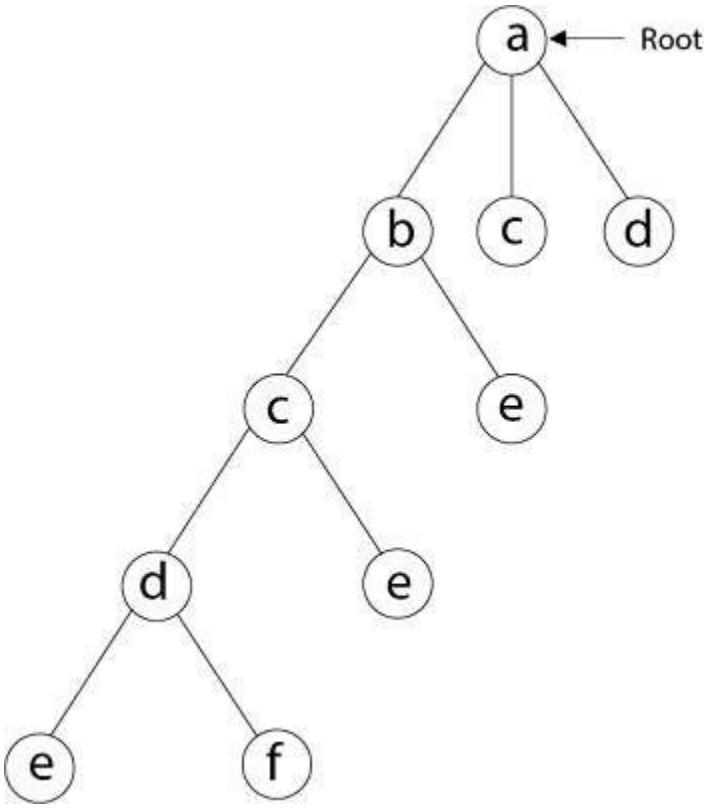
Next, we select 'c' adjacent to 'b.'



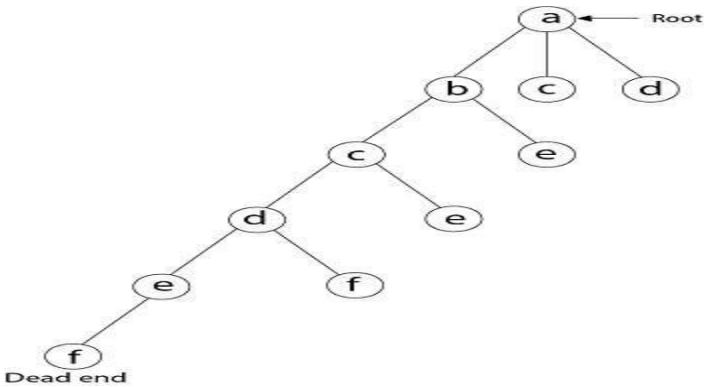
Next, we select 'd' adjacent to 'c.'



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

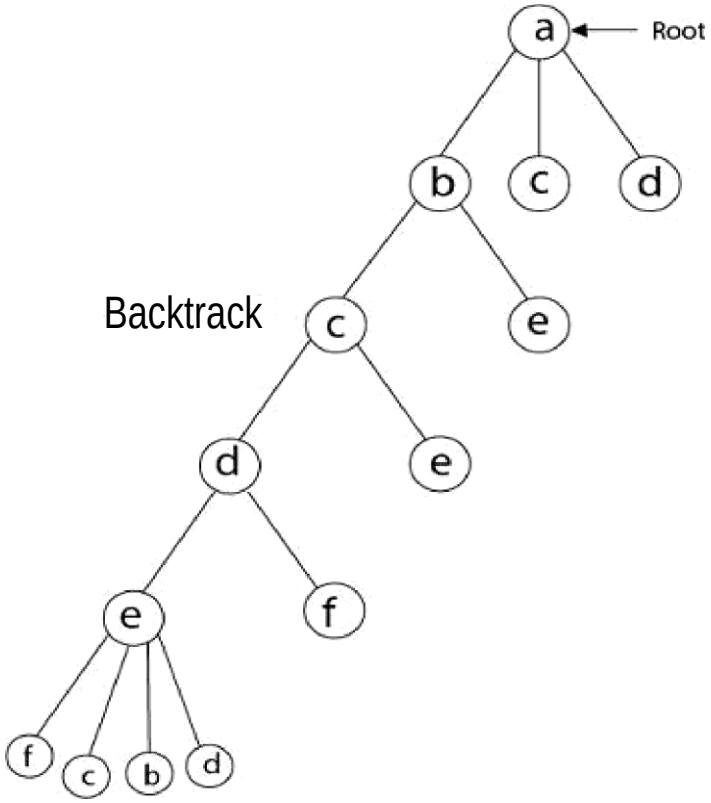


Next, we select vertex 'f' adjacent to 'e.' The vertex adjacent to 'f' is d and e, but they have already visited. Thus, we get the dead end, and we backtrack one step and remove the vertex 'f' from partial solution.

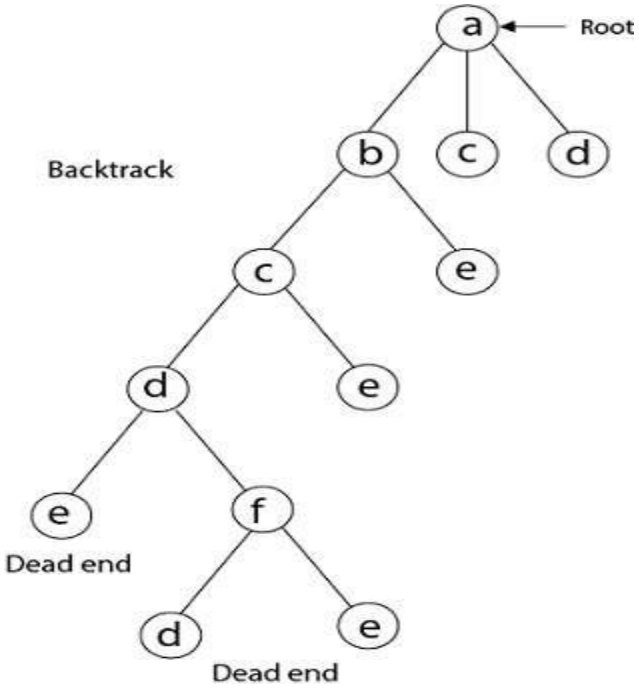


From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex, revisited them we get a dead state. So again we backtrack one step.

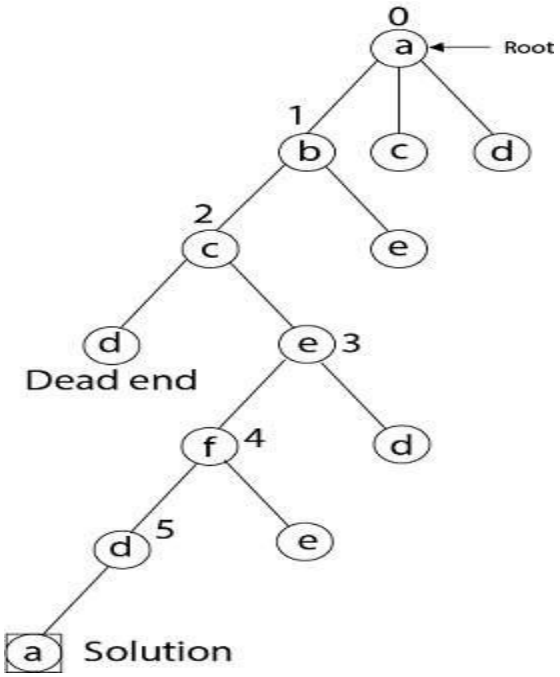
Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once. (a - b - c - e - f -d - a).



Dead end



Again Backtrack



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

Branch and bound

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

Job Assignment Problem:

Problem Statement:

Let's first define a job assignment problem. In a standard version of a job assignment problem, there can be jobs and workers. To keep it simple, we're taking jobs and workers in our example:

	Job 1	Job 2	Job 3
A	9	3	4
B	7	8	4
C	10	5	2

We can assign any of the available jobs to any worker with the condition that if a job is assigned to a worker, the other workers can't take that particular job. We should also notice that each job has some cost associated with it, and it differs from one worker to another.

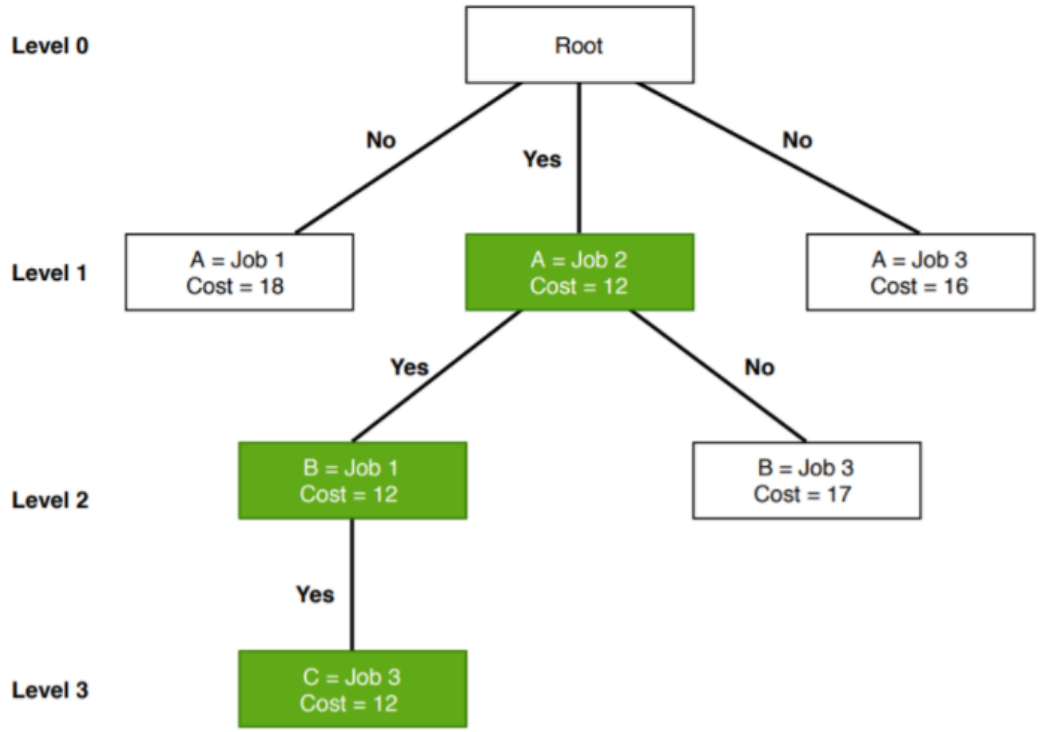
Here the main aim is to complete all the jobs by assigning one job to each worker in such a way that the sum of the cost of all the jobs should be minimized.

Pseudocode:

Algorithm 1: Job Assignment Problem Using Branch And Bound

```
Data: Input cost matrix M[][]
Result: Assignment of jobs to each worker according to optimal
        cost
Function MinCost(M[][])
while True do
    E = LeastCost();
    if E is a leaf node then
        print();
        return;
    end
    for each child S of E do
        Add(S);
        S → parent = E;
    end
end
```

Here, M[][] is the input cost matrix that contains information like the number of available jobs, a list of available workers, and the associated cost for each job. The function MinCost() maintains a list of active nodes. The function LeastCost() calculates the minimum cost of the active node at each level of the tree. After finding the node with minimum cost, we remove the node from the list of active nodes and return it. We are using the Add() function in the pseudocode, which calculates the cost of a particular node and adds it to the list of active nodes. In the search space tree, each node contains some information, such as cost, a total number of jobs, as well as a total number of workers.



Initially, we have 3 jobs available. The worker A has the option to take any of the available jobs. So at level 1, we assigned all the available jobs to the worker A and calculated the cost. We can see that when we assigned jobs 2 to the worker A, it gives the lowest cost in level 1 of the search space tree. So we assign the job 2 to worker A and continue the algorithm. “Yes” indicates that this is currently optimal cost.

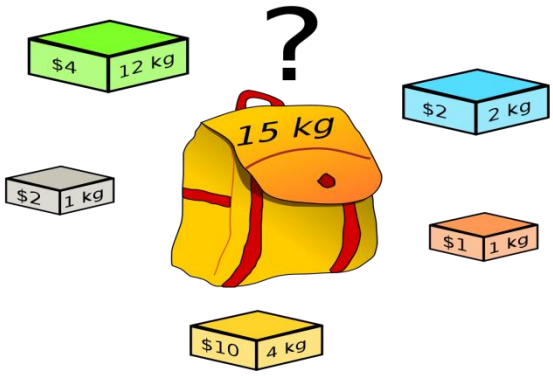
After assigning the job 2 to worker A, we still have two open jobs. Let’s consider worker B now. We are trying to assign either job 1 or 3 to worker B to obtain optimal cost.

Either we can assign the job 1 or 3 to worker B. Again we check the cost and assign job 1 to worker B as it is the lowest in level 2.

Finally, we assign the job 3 to worker C, and the optimal cost is 12.

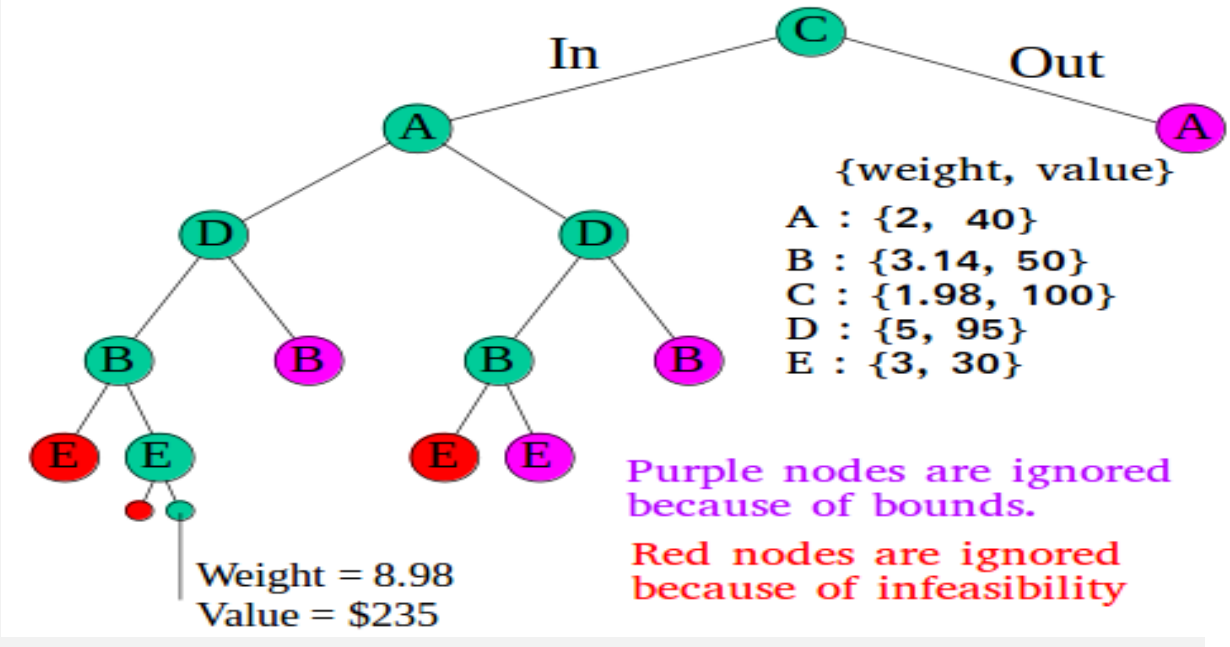
Knapsack Problem

Given two arrays **v[]** and **w[]** that represent values and weights associated with n items respectively. Find out the maximum value subset (**Maximum Profit)** of **v[]** such that sum of the weights of this subset is smaller than or equal to Knapsack capacity Cap(W).



Branch and bound (**BB)** is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization. Combinatorial optimization problems are mostly exponential in terms of time complexity. Also it may require to solve all possible permutations of the problem in worst case. So, by using Branch and Bound it can be solved quickly.

The backtracking based solution works better than brute force by ignoring infeasible solutions. To do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.



To find bound for every node for Knapsack:

To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using Greedy method. If the solution computed by Greedy approach is more than the best until now , then we can't get a better solution through the node.

Algorithm:

1. Sort all items in decreasing order of V/W so that upper bound can be computed using Greedy Approach.(The nodes taken in the image are accordingly.)
2. Initialize profit, max = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
5. Do while (Q is not empty).
    - Extract an item from Q. Let the item be x.
    - Compute profit of next level node. If the profit is more than max, then update max. (Profit from root to this node (include this node)).
    - Compute bound of next level node. If bound is more than max, then add next level node to Q.(Upper Bound of the maximum Profit in subtree of this node)
    - 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.

Branch and Bound Method

The branch and bound method are similar to the backtracking method except that, this method searches the nodes of the solution space in the Breadth First Search method. This branches to various nodes in search of the solution, but if an infeasible solution is encountered, then we bound back and try the next adjacent branch.

Travelling Salesman Problem

We have already solved this problem using the dynamic programming method. Now let us see how to solve this problem using branch and bound technique.

Given:

- A graph showing n number of cities connected by edges.
- Cost of each edge is given using the cost matrix.

Aim of the problem:

Find the shortest path to cover all the cities and come back to the same city.

Constraints:

- All the cities should be covered.
- Each city should be visited only once.
- The starting and the ending point should be the same.

Solution:

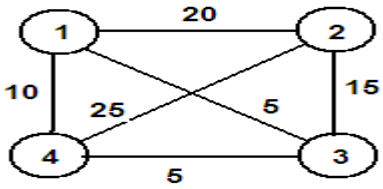
The given graph is the solution space for this problem. Hence, we perform a BFS in this graph to find the solution.

Start from node 1.

- Find all the adjacent nodes of the current node.
- Select the node which is not yet visited and has a least cost edge.
- Move to the selected node and repeat the above steps.

The solution is obtained when al the nodes are visited and we come back to the same city.

Example:



The cost matrix for the given graph is

	1	*2*	*3*	*4*
1	∞	20	5	10
2	20	∞	15	25
3	5	15	∞	5
4	10	25	5	∞

In the given graph, we start from node 1. To find out TSP path, first convert given cost matrix into cost reduced matrix. If all the rows and columns of the matrix having at least one zero than the matrix is cost reduced matrix. Take minimum element from each row and column and subtract all the elements of respective row and column by the minimum element.

	1	*2*	*3*	*4*	
1	∞	15	0	5	5
2	5	∞	0	10	15
3	0	10	∞	0	5
4	5	20	0	∞	5

Since all the row in the above matrix is having one zero , this matrix is called row reduced matrix.

	1	*2*	*3*	*4*	
1	∞	5	0	5	5
2	5	∞	0	10	15
3	0	0	∞	0	5
4	5	10	0	∞	5
		10			

R = sum of all subtraction = 5+15+5+5+10 = 40

Procedure to find out TSP,do following steps

1.Make all the entries of i^{th} row and j^{th} column to α

2. set A(j,1) to α

3. Convert cost reduced matrix

4. C(s) = R(s) + A(i,j)+R

Path from 1 to 2

	1	*2*	*3*	*4*
1	∞	∞	∞	∞
2	∞	∞	0	10
3	0	∞	∞	0
4	5	∞	0	∞

C(2) = R(1) + A(1,2)+R

= 40+5+0 = 45

Path from 1 to 3

	1	*2*	*3*	*4*
1	∞	∞	∞	∞
2	5	∞	∞	10
3	∞	0	∞	0
4	5	10	∞	∞

Make above matrix is cost reduced matrix

	1	*2*	*3*	*4*	
1	∞	∞	∞	∞	
2	0	∞	∞	5	5
3	∞	0	∞	0	
4	0	5	∞	∞	5

R = 5+5 = 10

C(3) = R(1) + A(1,3)+R

= 40+0+10 =50

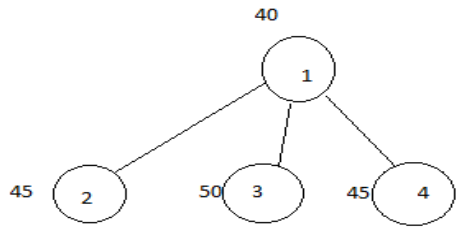
Path from 1 to 4

	1	*2*	*3*	*4*
1	∞	∞	∞	∞
2	5	∞	0	∞
3	0	0	∞	∞
4	∞	10	0	∞

C(4) = R(1) + A(1,4)+R

= 40+5+0=45

State space tree is



Since path from 1 to 2 is minimum, next node to be visited is 2 and now considers A matrix as

	1	*2*	*3*	*4*
1	∞	∞	∞	∞
2	∞	∞	0	10
3	0	∞	∞	0
4	5	∞	0	∞

Now find out path from 2 to 3

	1	*2*	*3*	*4*
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	0
4	5	∞	∞	∞

Make above matrix cost reduced matrix

	1	*2*	*3*	*4*
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	∞	∞	∞	0
4	0	∞	∞	∞

5

R = 5

C(3) = R(2) + A(2,3)+R

= 45+0+5 =45

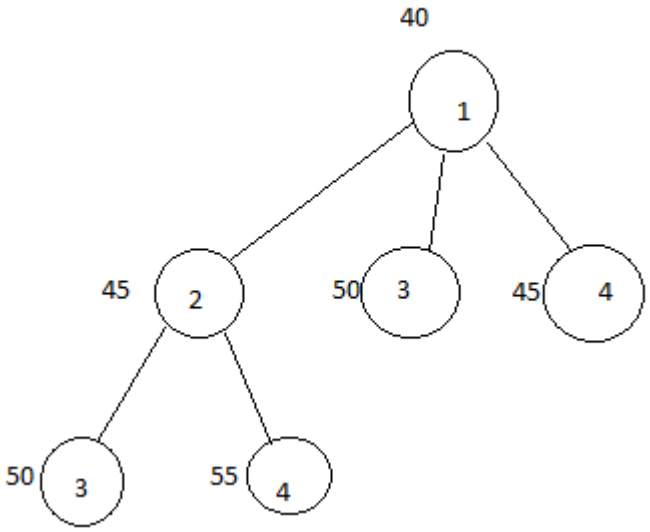
Path from 2 to 4

	1	*2*	*3*	*4*
1	∞	∞	∞	∞
2	∞	∞	∞	∞
3	0	∞	∞	∞
4	∞	∞	0	∞

C(4) = R(2) + A(2,4)+R

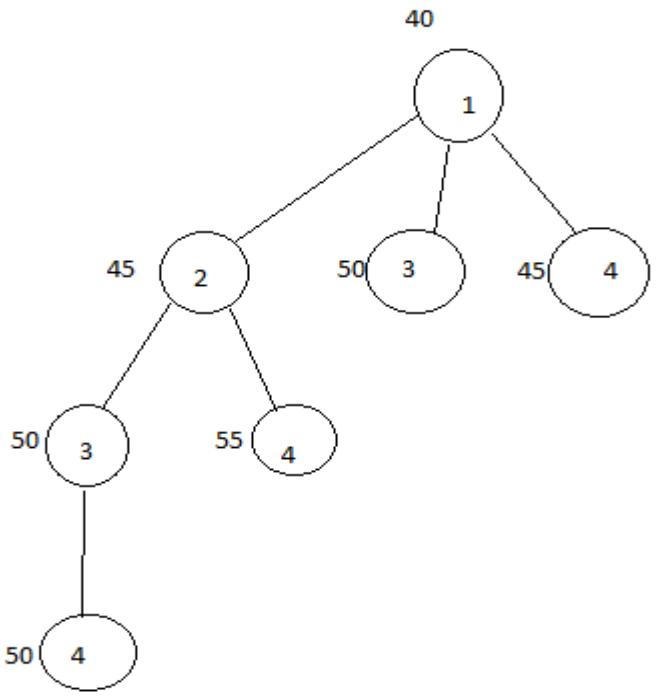
= 45+10+0 = 55

State space tree is



Since path from 2 to 3 is minimum next node to be visited is 3. Once we visited node 3 remaining to be visited is node 4.

The State space tree



The shortest path → 1 234 1

The cost of the path is 50.

Algorithm

BBTRAVEL(cost[],n)

U=city 1

Repeat while(all cities visited)

Find all cities w adjacent from u

If (cost of edge is minimum) and (city not yet visited)

Move to that city and mark it visited

u = current city

End if

End Repeat

Calculate path cost

Print Shortest path and its cost

End TRAVEL

NP-Complete Problem

NP-Complete is a decision problem in which there is a question in some formal system that can be posed as a yes-no question, dependent on the input values.

For example, the problem "given two numbers *x* and *y*, does *x* evenly divide *y*?" is a decision problem. The answer can be either 'yes' or 'no', and depends upon the values of *x* and *y*.

A method for solving a decision problem, given in the form of an algorithm, is called a decision procedure for that problem.

P-Problem(Polynomial- Problems)

The set of polynomially solvable problems are known as P-problems.

A problem is assigned to the P (polynomial time) class if there exists at least one algorithm to solve that problem, such that the number of steps of the algorithm is bounded by a polynomial in n, where n is the length of the input.

- Polynomial-time algorithms
  - Worst-case running time is $O(n^k)$, for some constant k
- Examples of polynomial time:
  - $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
- Examples of non-polynomial time:
  - $O(2^n)$, $O(n^n)$, $O(n!)$

NP-Problem(Non Deterministic polynomial problems)

It is the set of decision problems solvable in polynomial time by a non deterministic Turing machine. P problems are always NP problems. All problems whose answers can be verified in polynomial time are NP.NP problems includes problems with exponential algorithms but have not proved that they cannot have polynomial time algorithms. These are the problems that we have yet to find efficient algorithms in polynomial time.

Nondeterministic algorithm = two stage procedure:

1) Nondeterministic ("guessing") stage:

   Generate randomly an arbitrary string that can be thought of as a candidate solution ("certificate")

1) Deterministic ("verification") stage:

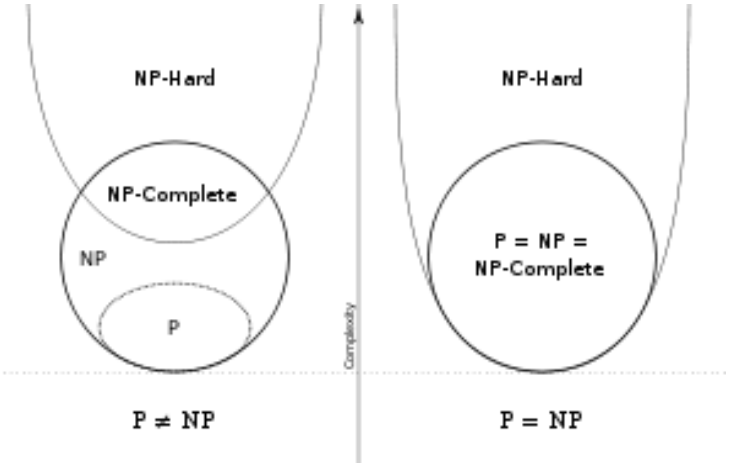
   Take the certificate and the instance to the problem and returns YES if the certificate represents a solution

NP algorithms (Nondeterministic polynomial)

verification stage is polynomial

NP-Hard Problems:

A problem is said to be NP-hard if an algorithm for solving it can be translated into one for solving any other NP problem.



NP Complete problems:

NP complete Problems have below two properties

1. Any given solution to the problem can be verified quickly.
2. If the problem is solved quickly, then every problem in NP can be solved quickly.

NP complete is a subset of NP. If every problem in NP can be quickly solved, then we call P=NP problem. If a problem is not solvable in polynomial time then P≠NP and all NP complete problems are not polynomial time solvable.

- Need to be in NP
- Need to be in NP-Hard

If both are satisfied then it is an NP complete problem

Solving NP Complete Problems

Given NP-Complete problems, what should do?

1. Use Brute Force may be the algorithm performance is acceptable for small input sizes.
2. Use time limit: terminates the algorithm after time limit.
3. Use approximate algorithms for optimization problems: find a good solution, but not necessary the best solution.

Clique problem

The clique problem is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph. It has several different formulations depending on which cliques, and what information about the cliques, should be found.

A Clique is a subgraph of graph such that all vertcies in subgraph are completely connected with each other.

- Undirected graph G = (V, E)
- **Clique:** a subset of vertices in V all connected to each other by edges in E (i.e., forming a complete graph)
- **Size of a clique:** number of vertices it contains

A maximal clique is a clique that cannot be extended by including one more adjacent vertex, that is, a clique which does not exist exclusively within the vertex set of a larger clique. Some authors define cliques in a way that requires them to be maximal, and use other terminology for complete subgraphs that are not maximal.

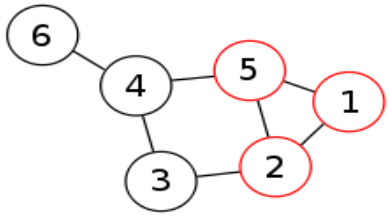
A maximum clique of a graph, *G*, is a clique, such that there is no clique with more vertices.

The clique number ω(*G*) of a graph *G* is the number of vertices in a maximum clique in *G*.

The intersection number of *G* is the smallest number of cliques that together cover all edges of *G*.

The clique cover number of a graph *G* is the smallest number of cliques of *G* whose union covers *V(G)*.

A maximum clique transversal of a graph is a subset of vertices with the property that each maximum clique of the graph contains at least one vertex in the subset.[2]



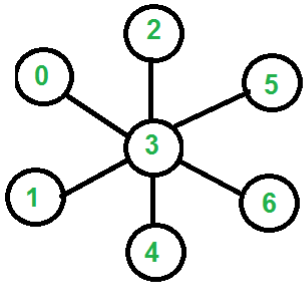
The graph shown has one maximum clique, the triangle {1,2,5}, and four more maximal cliques, the pairs {2,3},}3,4}, {4,5} and {4,6}.

Vertex Cover Problem:

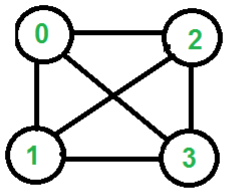
Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial time solution for this unless P = NP.

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. *Given an undirected graph, the vertex cover problem is to find minimum size vertex cover*.

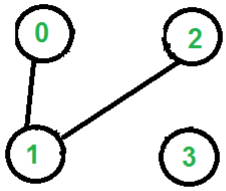
Following are some examples.



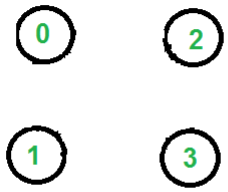
Minimum Vertex Cover is {3}



Minimum Vertex Cover is {0, 1, 2} or
{0, 1, 3} or {1, 2, 3}



Minimum Vertex Cover is {1}

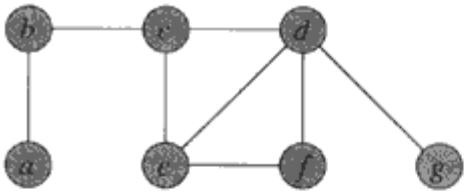


Minimum Vertex Cover is empty {}

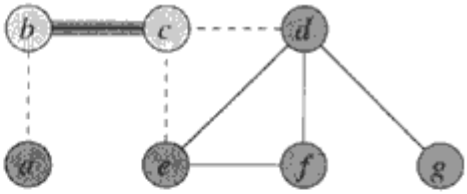
.

Approximate Algorithm for Vertex Cover:

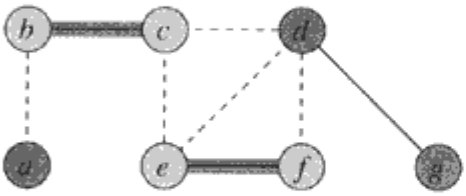
1. Initialize the result as {}
2. Consider a set of all edges in given graph. Let the set be E.
3. Do the following while E is not empty.
    a) Pick an arbitrary edge (u,v) from set E and add ‘u’ and ‘v’ to result.
    b) Remove all edges from E which are either incident on u or v.
4. Return result.



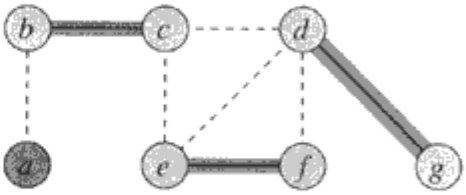
(a)



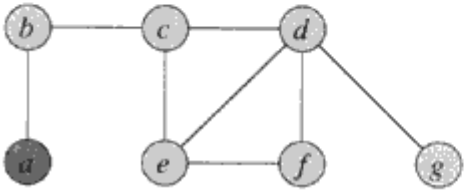
(b)



(c)



(d)



(e)

(f)