JGi JAIN DEEMED-TO-BE UNIVERSITY | FACULTY OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

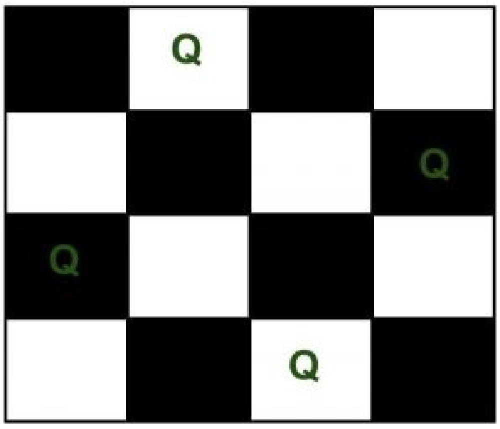
MODULE-5 QUESTION BANK

1. **What is backtracking algorithm with example?**

Problem Statement
A classic example of backtracking is the n-Queens problem, first proposed by German chess enthusiast Max Bezzel in 1848. Given a chessboard of size n, the problem is to place n queens on the n × n chessboard, so no two queens are attacking each other.
It is clear that for this problem, we need to find all the arrangements of the positions of the n queens on the chessboard, but there is a constraint: no queen should be able to attack another queen.

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

```
1) Start in the leftmost column
2) If all queens are placed
   return true
3) Try all rows in the current column.  Do following for every tried row.
   a) If the queen can be placed safely in this row then mark this [row,
      column] as part of the solution and recursively check if placing
      queen here leads to a solution.
   b) If placing 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.
```

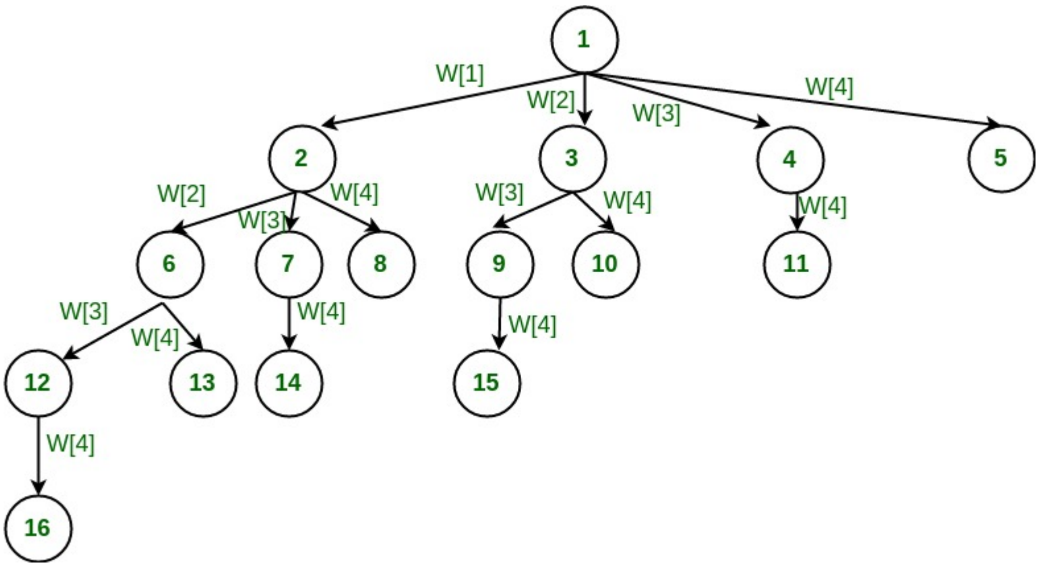


2. **What is backtracking solve the sum of subsets problem using backtracking technique with example?**

Subset sum problem is to find subset of elements that are selected from a given set whose sum adds up to a given number K. We are considering the set contains non-negative values. It is assumed that the input set is unique (no duplicates are presented).

Backtracking Algorithm for Subset Sum
Using exhaustive search we consider all subsets irrespective of whether they satisfy given constraints or not. Backtracking can be used to make a systematic consideration of the elements to be selected.
Assume given set of 4 elements, say w[1] … w[4]. Tree diagrams can be used to design backtracking

algorithms. The following tree diagram depicts approach of generating variable sized tuple.



3. **What are the steps to color the graph using backtracking algorithm?**

Given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.

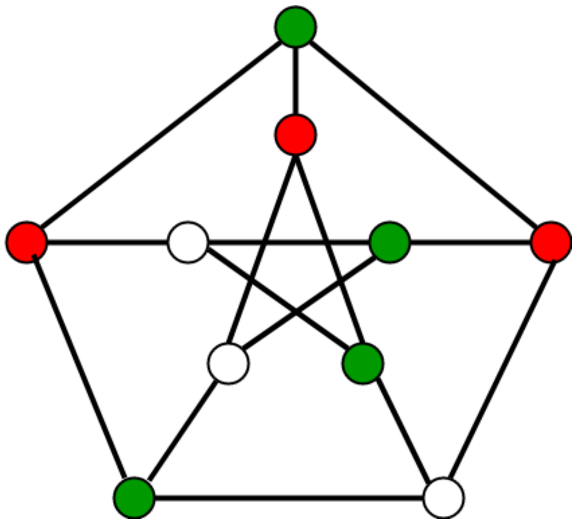
Input-Output format:

Input:

A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is an adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
An integer m is the maximum number of colors that can be used.
Output:
An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Example:

Input:
graph = {0, 1, 1, 1},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {1, 0, 1, 0}
Output:
Solution Exists:
Following are the assigned colors
 1 2 3 2
Explanation: By coloring the vertices with following colors, adjacent vertices does not have same colors


example of a graph that can be coloured with 3 different colours.

Input:
graph = {1, 1, 1, 1},
        {1, 1, 1, 1},
        {1, 1, 1, 1},
        {1, 1, 1, 1}
Output: Solution does not exist.
Explanation: No solution exits.

4. **What is Hamiltonian circuit problem what is the procedure to find Hamiltonian circuit of a graph?**

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.
Input:
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
Output:
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

Naive Algorithm
Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

```
while there are untried conflagrations
{
  generate the next configuration
  if ( there are edges between two consecutive vertices of this
    configuration and there is an edge from the last vertex to
    the first ).
  {
    print this configuration;
    break;
  }
}
```

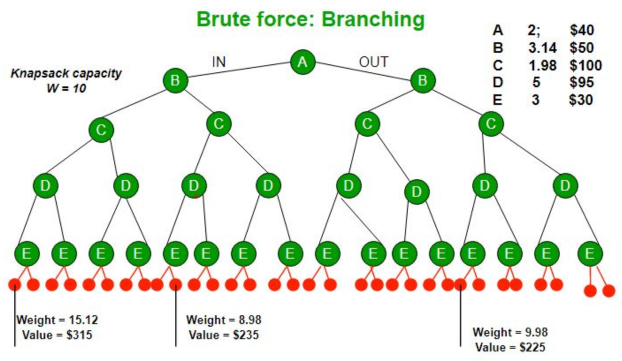
Backtracking Algorithm
Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

5. **How do you calculate the bound value for the knapsack problem in branch and bound?**

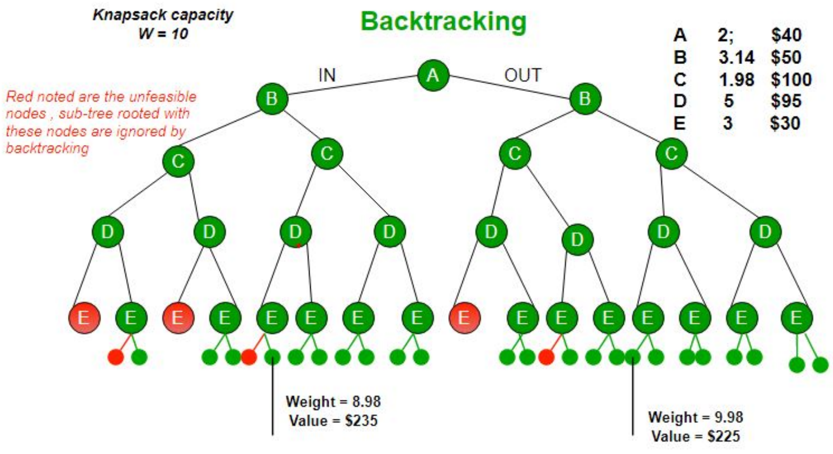
Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly. Let us consider below 0/1 Knapsack problem to understand Branch and Bound. 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. Let us explore all approaches for this problem.

A Greedy approach is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.
We can use Dynamic Programming (DP) for 0/1 Knapsack problem. In DP, we use a 2D table of size n x W. The DP Solution doesn't work if item weights are not integers.
Since DP solution doesn't always work, a solution is to use Brute Force. With n items, there are 2n solutions to be generated, check each to see if they satisfy the constraint, save maximum solution that

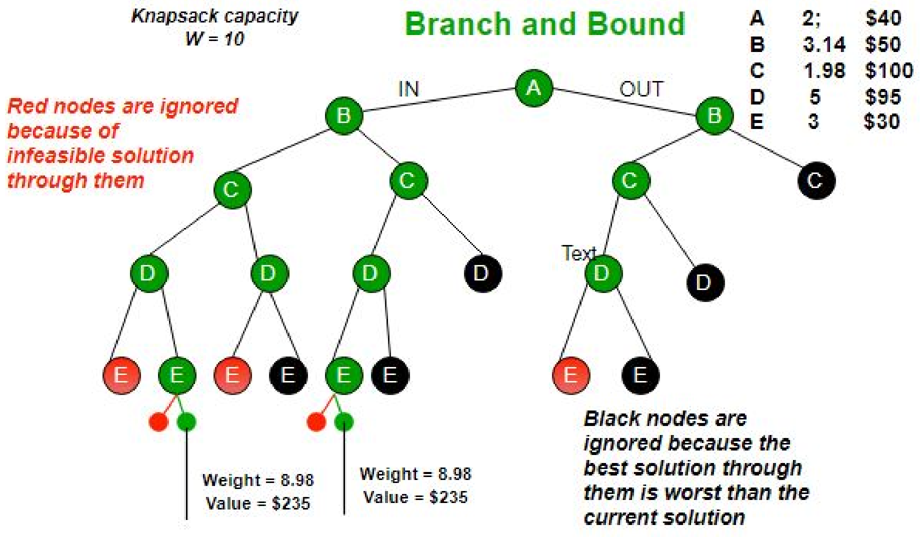
satisfies constraint. This solution can be expressed as tree.



We can use Backtracking to optimize the Brute Force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.

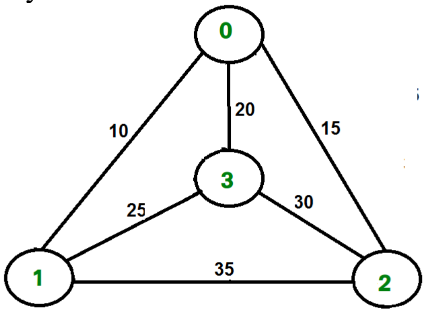

Knapsack capacity W = 10
Backtracking
A 2; $40
B 3.14 $50
C 1.98 $100
D 5 $95
E 3 $30
Red noted are the unfeasible nodes , sub-tree rooted with these nodes are ignored by backtracking
IN
OUT
Weight = 8.98 Value = $235
Weight = 9.98 Value = $225

Branch and BoundThe backtracking based solution works better than brute force by ignoring infeasible solutions. We can 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. Example bounds used in below diagram are, A down can give $315, B down can $275, C down can $225, D down can $125 and E down can $30.


Knapsack capacity W = 10
Branch and Bound
A 2; $40
B 3.14 $50
C 1.98 $100
D 5 $95
E 3 $30
Red nodes are ignored because of infeasible solution through them
IN
OUT
Text
Weight = 8.98 Value = $235
Weight = 8.98 Value = $235
Black nodes are ignored because the best solution through them is worst than the current solution

6. **What is the travelling salesman problem and find the optimal path using branch and bound strategy?**

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is 10+25+30+15 which is 80.

Branch and Bound Solution
As seen in the previous articles, in Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.
Note that the cost through a node includes two costs.
1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)
2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

In cases of a maximization problem, an upper bound tells us the maximum possible solution if we follow the given node. For example in 0/1 knapsack we used Greedy approach to find an upper bound.
In cases of a minimization problem, a lower bound tells us the minimum possible solution if we follow the given node. For example, in Job Assignment Problem, we get a lower bound by assigning least cost job to a worker.
In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for Traveling salesman problem.
Cost of any tour can be written as below.

Cost of a tour T = (1/2) * ∑ (Sum of cost of two edges
adjacent to u and in the
tour T)
where u ∈ V
For every vertex u, if we consider two edges through it in T, and sum their costs. The overall sum for all vertices would be twice of cost of tour T (We have considered every edge twice.)

(Sum of two tour edges adjacent to u) >= (sum of minimum weight
two edges adjacent to
u)

Cost of any tour >= 1/2) * ∑ (Sum of cost of two minimum
weight edges adjacent to u)
where u ∈ V
For example, consider the above shown graph. Below are minimum cost two edges adjacent to every node.

Node	Least cost edges	Total cost
0	(0, 1), (0, 2)	25
1	(0, 1), (1, 3)	35
2	(0, 2), (2, 3)	45
3	(0, 3), (1, 3)	45

Thus a lower bound on the cost of any tour =
$1/2(25 + 35 + 45 + 45)$
$= 75$
Refer this for one more example.
Now we have an idea about computation of lower bound. Let us see how to how to apply it state space search tree. We start enumerating all possible nodes (preferably in lexicographical order)

7. **What do you mean by P NP NP-hard and NP-complete problems give an example of each category?**
P Class

The P in the P class stands for Polynomial Time. It is the collection of decision problems(problems with a "yes" or "no" answer) that can be solved by a deterministic machine in polynomial time.

Features:

The solution to P problems is easy to find.
P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.
This class contains many natural problems like:

Calculating the greatest common divisor.
Finding a maximum matching.
Decision versions of linear programming.

NP Class

The NP in NP class stands for Non-deterministic Polynomial Time. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:
The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
Problems of NP can be verified by a Turing machine in polynomial time.

Example:
Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to some personal reasons.
This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

Boolean Satisfiability Problem (SAT).
Hamiltonian Path Problem.
Graph coloring.

NP-hard class
An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.

Features:

All NP-hard problems are not in NP.
It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.
Some of the examples of problems in Np-hard are:

Halting problem.
Qualified Boolean formulas.
No Hamiltonian cycle.

NP-complete class
A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hardest problems in NP.
Features:
NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.
Some example problems include:

0/1 Knapsack.
Hamiltonian Cycle.
Satisfiability.
Vertex cover.

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

8 **Difference between NP hard and NP complete problem?**

NP-hard	NP-Complete
NP-Hard problems (say X) can be solved if and only if there is a NP-Complete problem (say Y) that can be reducible into X in polynomial time.	NP-Complete problems can be solved by a non-deterministic Algorithm/Turing Machine in polynomial time.
To solve this problem, it do not have to be in NP.	To solve this problem, it must be both NP and NP-hard problems.
Do not have to be a Decision problem.	It is exclusively a Decision problem.
Example: Halting problem, Vertex cover problem, etc.	Example: Determine whether a graph has a Hamiltonian cycle, determine whether a Boolean formula is satisfiable or not, Circuit-satisfiability problem, etc.

9 **What is LC branch and bound?**
Branch and bound is an algorithm to find the optimal solution to various optimization problems. It is very similar to the backtracking strategy, only just in the backtracking method state-space tree is used. The benefit of the branch and bound method is that it does not limit us to only a certain way of traversing the tree. Though it also has certain drawbacks, one being it is much slower and gets an exponential time complexity in the worst cases. The branch and bound method are applicable to many discrete combinational problems.
The branch and bound can be solved using FIFO, LIFO, and least count strategies. The selection rule used in the FIFO or LIFO linked list can sometimes be erroneous, as it does not give preference to certain nodes, and hence may not find the best response as quickly. To overcome this drawback, we use the least cost method, which is also considered to be an intelligent method as it uses an intelligent ranking method. It is also referred to as the approximate cost function "C".

10 **Which data structure is used for implementing a FIFO branch and bound strategy?**

FIFO Branch and Bound solution is one of the methods of branch and bound.
Branch and Bound is the state space search method where all the children E-node that is generated before the live node, becomes an E- node.
FIFO branch and bound search is the one that follows the BFS like method. It does so as the list follows first in and first out.
Some key terms to keep in mind while proceeding further:
What is a live node?
A live node is the one that has been generated but its children are not yet generated.
What is an E node?
An E node is the one that is being explored at the moment.

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

Branch and Bound solution have three types of strategies:

FIFO branch and bound.
LIFO branch and bound.
Least Cost branch and bound.

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

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

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

Let us understand it through an example:

We start by taking an empty queue:

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

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

2 3 4

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

3 4 5 6

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

4 5 6

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

5 6

Similarly, we proceed further, but the children of nodes 5, meaning the nodes 10, and 11 are also generated and killed by boundary function. The last standing node in the queue is 6, the children of

	node 6 are 12, and it satisfies all the conditions for the problem, therefore it is the answer node.