UNIT -3

GREEDY METHOD

The greedy method is a problem-solving approach used in algorithms, where the best choice is made at each step based on a specific criterion, with the hope of finding an optimal solution. The method follows these key principles:

Greedy Choice Property: Make the best possible choice at each step without considering the global solution.

Optimal Substructure: The globally optimal solution can be constructed by choosing local optimal solutions at each step.

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

JOB SEQUENCING WITH DEADLINES

The **Job Sequencing with Deadlines** problem is another classic problem where the greedy method provides an optimal solution. It's used to maximize profit by selecting the most profitable jobs that can be completed within their deadlines.

Problem Statement:

- You are given n jobs, each with a deadline and a profit.
- Each job takes exactly one unit of time.
- You can only complete one job at a time, and a job must be completed before or on its deadline.
- The goal is to schedule the jobs in such a way that you **maximize the total profit**.

Greedy Strategy:

The greedy strategy involves scheduling jobs with the highest profit first, but only if they can be completed within their deadlines.

Greedy Algorithm:

1. **Sort the jobs by profit** in descending order.

2. **Initialize a time slot array** to keep track of which time slots are free (this array will be of length equal to the maximum deadline). (r-1,r) or (r-2,r-1) ..
3. **Iterate through the jobs** in sorted order (from the highest profit to the lowest).
    - For each job, try to schedule it at its latest possible time (which is its deadline). If that slot is already taken, check earlier time slots until an available one is found.
    - If a slot is found, assign the job to that slot.
4. **Return the scheduled jobs and the total profit**.

Example:

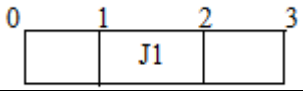
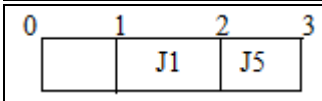
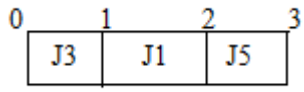
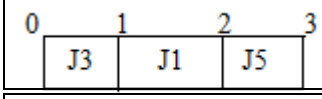
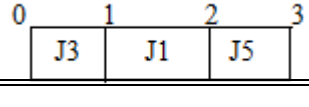
Let's take an example where we have 5 jobs with the following deadlines and profits:

Job	Deadline d_i	Profit p_i
1	2	100
2	1	19
3	2	38
4	1	27
5	3	52

1. **Sort jobs by profit** in descending order:

Jobs sorted by profit: Job 1 (100), Job 5 (52), Job 3 (38), Job 4 (27), Job 2 (19).

2. **Initialize the time slots** (maximum deadline is 3, so the array has 3 slots: [0, 0, 0]).
3. **Schedule the jobs**:

Assigned Slot	Job selected	Action	Profit	Time slot
None	J1	[1,2]	100	
[1,2]	J5	[2,3]	100+52=152	
[1,2], [2,3]	J3	[0,1]	152+38=190	
[1,2], [2,3], [0,1]	J4	rejected	190	
[1,2], [2,3], [0,1]	J2	rejected	190	

Job sequence = {J1, J5, J3}

Total profit = 190

Algorithm:-

```
Algorithm greedy_job(n)
// p[1..n], d[1..n], p[ i ] ≥ p[ i+1]
 profit=0;
count=0;      // how many jobs are executed

dmax=0
for i  =1 to n do
{
if( d[ i ] > dmax) then
dmax = d[ i ]
}

for i  =1 to dmax do
{
slot = -1
}

for i = 1 to n do
{
for r = d[ i ] down to 1
if (slot [ r ] = = -1) then
{
slot [ r ] = i
profit + = p[i]
count++
break
}
}
if (count = = dmax)
break
}
```

KNAPSACK PROBLEM

The selection of items, each with profit and weight values, to be packed into one or more knapsacks with capacity is the fundamental idea behind all families of knapsack problems.
The fractional **Knapsack problem using the Greedy Method** is an efficient method to solve it, where you need to sort the items according to their ratio of value/weight. In a fractional

knapsack, we can break items to maximize the knapsack's total value. This problem in which we can break an item is also called the **Fractional knapsack problem**.
In this method, the Knapsack's filling is done so that the maximum capacity of the knapsack is utilized so that maximum profit can be earned from it. The knapsack problem using the Greedy Method is referred to as:

Given a list of n objects, say $\{I_1, I_2, \ldots\ldots, I_n)$ and a knapsack (or bag). The capacity of the knapsack is M. Each object I_j has a weight w_j and a profit of p_j If a fraction x_j (where $x \in \{0\ldots., 1)$) of an object I_j is placed into a knapsack, then a profit of $p_j x_j$ is earned. The problem (or Objective) is to fill the knapsack (up to its maximum capacity M), maximizing the total profit earned.

$$\text{Maximize (the profit)} = \sum_{j=1}^{n} p_j x_j$$

$$= \sum_{j=1}^{n} w_j x_j \le M \text{ and } x_j \in \{0,\ldots.,1\}, 1 \le j \le n$$

Algorithm:-

```
greedy fractional-knapsack (P[1…n], W[1…n], X[1..n]. M)
//*P[1…n] and W[1…n] contain the profit and weight of the n-objects ordered such that X[1…n]
is a solution set and M is the capacity of knapsack*/
{

For j ← 1 to n do
X[j]← 0
profit ← 0 // Total profit of item filled in the knapsack
weight ← 0 // Total weight of items packed in knapsacks
j ← 1
While (Weight < M) // M is the knapsack capacity
{
if (weight + W[j] =< M)
X[j] = 1
weight = weight + W[j]
else
{
X[j] = (M – weight)/w[j]
weight = M
}
Profit = profit + p[j] * X[j]
j++;
```

} // end of while
} // end of Algorithm

Example:-

Objects:	1	2	3	4	5	6	7
Profit (P):	5	10	15	7	8	9	4
Weight(w):	1	3	5	4	1	3	2

W (Weight of the knapsack): 15
n (no of items): 7

In this case, we first calculate the profit/weight ratio.
Object 1: 5/1 = 5
Object 2: 10/3 = 3. 33
Object 3: 15/5 = 3
Object 4: 7/4 = 1.7
Object 5: 8/1 = 8
Object 6: 9/3 = 3
Object 7: 4/2 = 2

P:w:	**5**	**3.3**	**3**	**1.7**	**8**	**3**	**2**

In this approach, we will select the objects based on the maximum profit/weight ratio. Since the P/W of object 5 is maximum so we select object 5.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 8 = 7

After object 5, object 1 has the maximum profit/weight ratio, i.e., 5. So, we select object 1 shown in the below table:

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13

After object 1, object 2 has the maximum profit/weight ratio, i.e., 3.3. So, we select object 2 having profit/weight ratio as 3.3.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10

After object 2, object 3 has the maximum profit/weight ratio, i.e., 3. So, we select object 3 having profit/weight ratio as 3.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5

After object 3, object 6 has the maximum profit/weight ratio, i.e., 3. So we select object 6 having profit/weight ratio as 3.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2

After object 6, object 7 has the maximum profit/weight ratio, i.e., 2. So we select object 7 having profit/weight ratio as 2.

Object	Profit	Weight	Remaining weight
5	8	1	15 - 1 = 14
1	5	1	14 - 1 = 13
2	10	3	13 - 3 = 10
3	15	5	10 - 5 = 5
6	9	3	5 - 3 = 2
7	4	2	2 - 2 = 0

As we can observe in the above table that the remaining weight is zero which means that the knapsack is full. We cannot add more objects in the knapsack. Therefore, the total profit would be equal to (8 + 5 + 10 + 15 + 9 + 4), i.e., 51.

MINIMUM SPANNING TREES

A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees. We can assign *weights* to each edge (which is a number that represents how unfavorable the edge is), and use it to assign a weight to a spanning tree by calculating the sum of the weights of the edges in that spanning tree. A *minimum spanning tree* (MST) is defined as a spanning tree with weight less than or equal to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Prim's Algorithm

Prim's algorithm is a greedy algorithm that is used to form a minimum spanning tree for a connected weighted undirected graph. In other words, the algorithm builds a tree that includes every vertex and a subset of the edges in such a way that the total weight of all the edges in the tree is minimized. For this, the algorithm maintains three sets of vertices which can be given as below:

***Tree vertices* -** Vertices that are a part of the minimum spanning tree T.

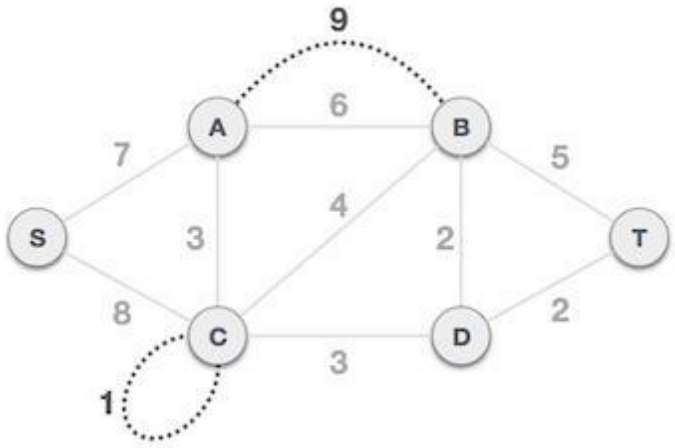
***Fringe vertices* -** Vertices that are currently not a part of T, but are adjacent to some tree vertex.

***Unseen vertices* -** Vertices that are neither tree vertices nor fringe vertices fall under this category.

Choose a starting vertex. Branch out from the starting vertex and during each iteration, select a new vertex and an edge. Basically, during the each iteration of the algorithm, we have to select a vertex from the fringe vertices in such a way that the edge connecting the tree vertex and the new vertex has the minimum weight assigned to it. The running time of Prim's algorithm can be given as O(E log V) where E is the number of edges and V is the number of vertices in the graph.

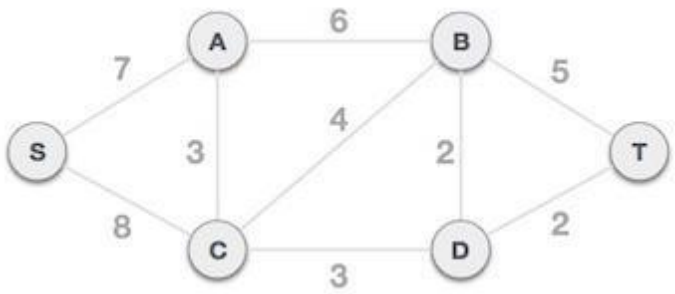
Example:-

Construct a minimum spanning tree of the graph given in Fig.



Step 1 - Remove all loops and parallel edges

Remove all loops and parallel edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

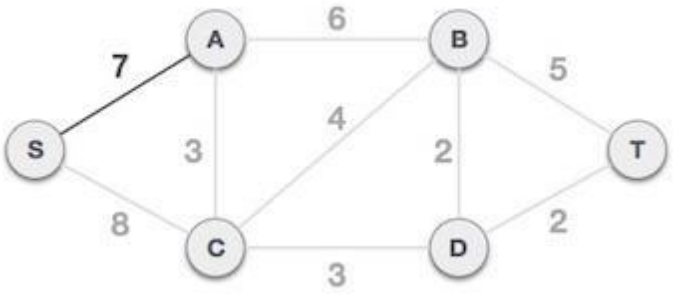


Step 2 - Choose any arbitrary node as root node

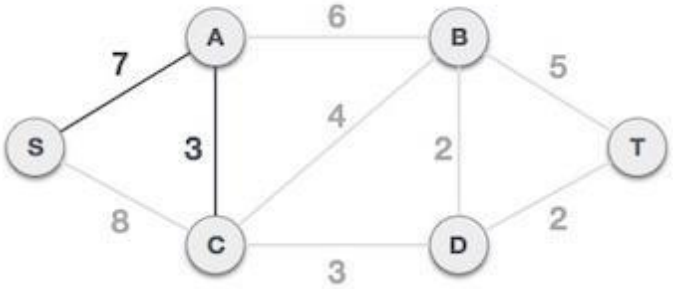
In this case, we choose S node as the root node of Prim's spanning tree. This node is arbitrarily chosen, so any node can be the root node. One may wonder why any video can be a root node. So the answer is, in the spanning tree all the nodes of a graph are included and because it is connected then there must be at least one edge, which will join it to the rest of the tree.

Step 3 - Check outgoing edges and select the one with less cost

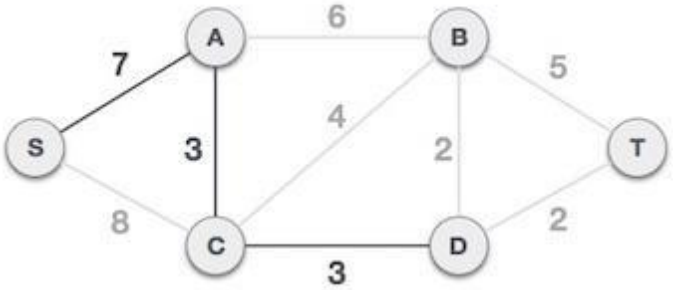
After choosing the root node S, we see that S, A and S, C are two edges with weight 7 and 8, respectively. We choose the edge S,A as it is lesser than the other.



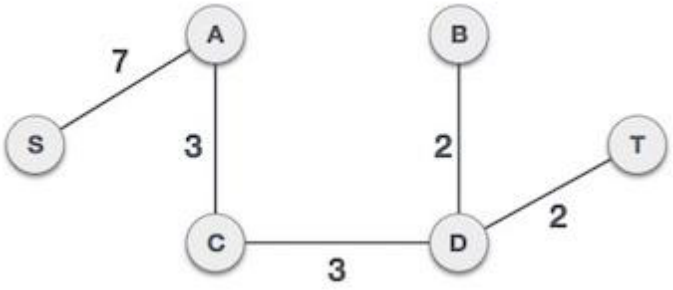
Now, the tree S-7-A is treated as one node and we check for all edges going out from it. We select the one which has the lowest cost and include it in the tree.



After this step, S-7-A-3-C tree is formed. Now we'll again treat it as a node and will check all the edges again. However, we will choose only the least cost edge. In this case, C-3-D is the new edge, which is less than other edges' cost 8, 6, 4, etc.



After adding node **D** to the spanning tree, we now have two edges going out of it having the same cost, i.e. D-2-T and D-2-B. Thus, we can add either one. But the next step will again yield edge 2 as the least cost. Hence, we are showing a spanning tree with both edges included.



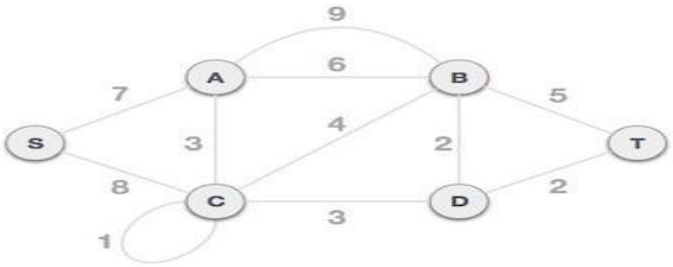
Algorithm:-

```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3:     Select an edge e connecting the tree vertex and
            fringe vertex that has minimum weight
Step 4:     Add the selected edge and the vertex to the
            minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

Kruskal's Algorithm

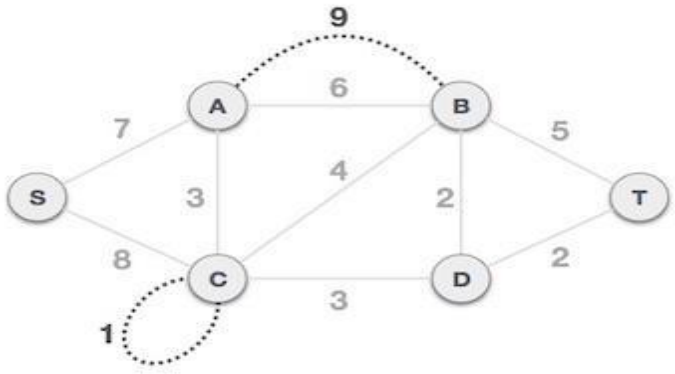
Kruskal's algorithm is used to find the minimum spanning tree for a connected weighted graph. The algorithm aims to find a subset of the edges that forms a tree that includes every vertex. The total weight of all the edges in the tree is minimized. However, if the graph is not connected, then it finds a *minimum spanning forest*. Note that a forest is a collection of trees. Similarly, a minimum spanning forest is a collection of minimum spanning trees. Kruskal's algorithm is an example of a greedy algorithm, as it makes the locally optimal choice at each stage with the hope of finding the global optimum.

To understand Kruskal's algorithm let us consider the following example –

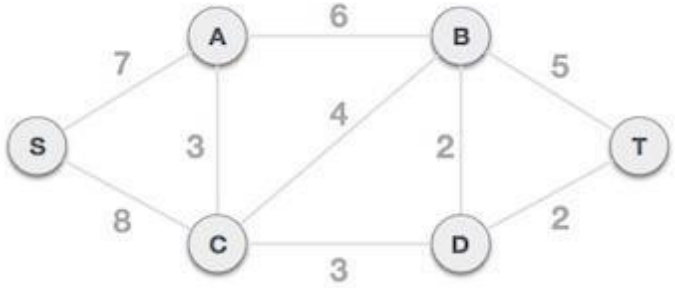


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.



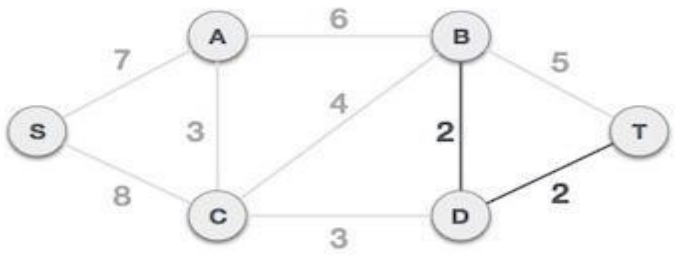
Step 2 - Arrange all edges in their increasing order of weight

The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

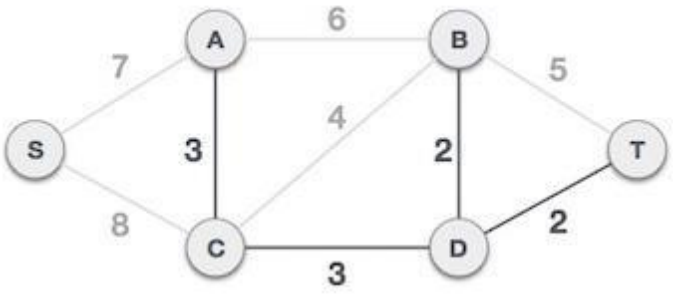
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weight age

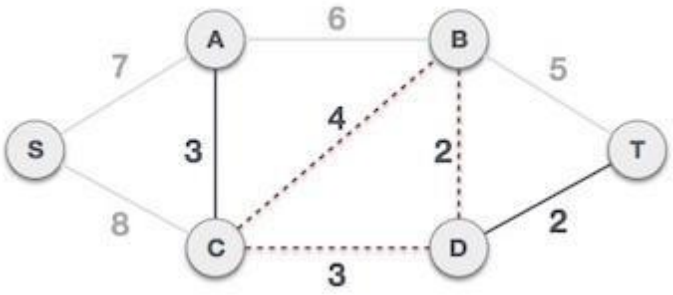
Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not to include the edge in the graph.



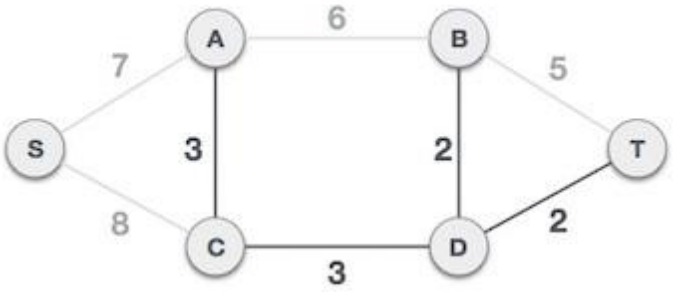
The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection.
Next cost is 3, and associated edges are A,C and C,D. We add them again –



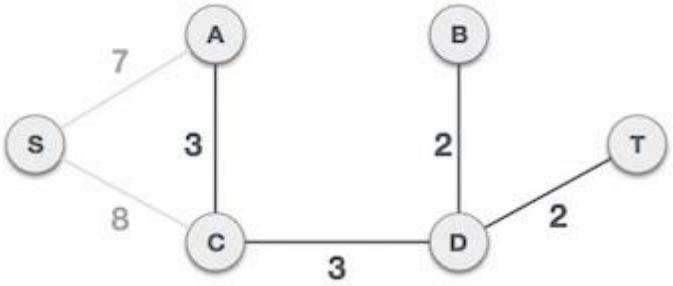
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph. –



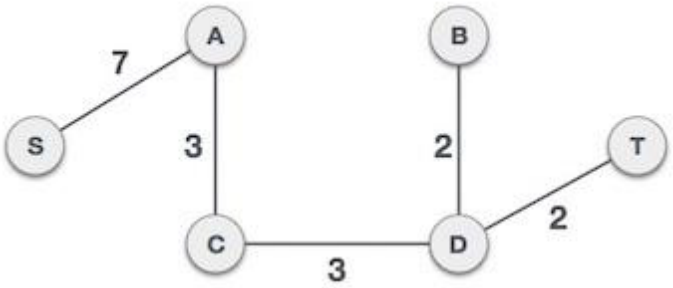
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

Algorithm

Step 1: Create a forest F in such a way that every vertex of the graph is a separate tree.
Step 2: Create a set E that contains all the edges of the graph.
Step 3: Repeat Steps 4 and 5 **while** E is NOT EMPTY and F is not spanning
Step 4: Remove an edge from E with minimum weight
Step 5: IF the edge obtained in Step 4 connects two different trees, then add it to the forest F (**for** combining two trees into one tree).
ELSE
Discard the edge
Step 6: END

Dijkstra's Algorithm

Dijkstra's Algorithm is a **greedy algorithm** that is used to find the shortest path between a **single source** and all other nodes in a graph. It is widely used in routing and navigation systems because of its efficiency in finding the shortest path in a graph with non-negative edge weights.

Key Concepts:

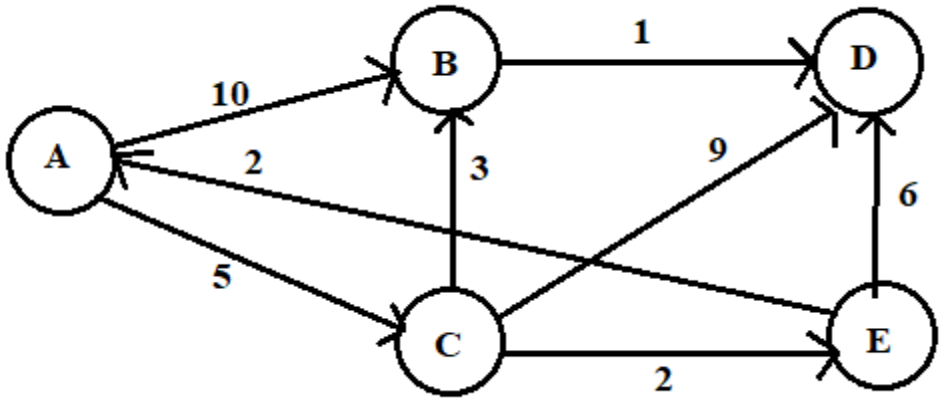
- **Single-source shortest path**: The algorithm finds the shortest path from one source node to all other nodes in the graph.
- **Greedy method**: Dijkstra's algorithm builds the shortest path step by step, selecting the closest unvisited vertex at each step.
- **Non-negative weights**: The algorithm assumes that the weights (or distances) between vertices are non-negative.

Algorithm Steps:

1. **Initialization**:
    - Set the distance to the source node as 0 and all other nodes as infinity.
    - Create a set of unvisited nodes (initially, all nodes are unvisited).
2. **Visit the nearest unvisited node**:
    - Among all the unvisited nodes, select the node with the smallest known distance (initially, the source node).
    - Mark this node as visited (it will not be revisited).

3. **Update distances**:
    - For each neighboring node of the currently visited node, calculate the distance from the source through the current node.
    - If this new distance is smaller than the previously known distance, update it.
4. **Repeat**:
    - Repeat the process for the next nearest unvisited node.
    - Continue until all nodes have been visited or the shortest distance to all nodes has been determined.

Example:



SELECTED VERTEX	A	B	C	D	E
A	0	α	α	α	α
C		10	5	α	α
E		8		14	7
B		8		13	
D				9	

Shortest distance from A to B is 8
Shortest distance from A to C is 5
Shortest distance from A to D is 9
Shortest distance from A to E is 7

Path from A to D is A – C – B – D = 9

BACKTRACKING

Backtracking is a problem-solving strategy that involves recursively exploring all possible solutions to a problem. It is a form of brute force search that involves systematically exploring all possible solutions, and when a dead end is reached, the algorithm backtracks to a previous state and explores another branch. This process continues until a solution is found or all possible solutions have been exhausted.

Backtracking is often used to solve problems that have multiple solutions or where the solution space is too large to be explored exhaustively. It is particularly useful when the problem has a recursive structure, and the solution to the problem can be broken down into smaller sub-problems.

How Backtracking Works

The backtracking algorithm works by recursively exploring all possible solutions to a problem. The algorithm starts by selecting an initial solution, and then recursively explores all possible solutions by adding one element at a time. When a dead end is reached, the algorithm backtracks to a previous state and explores another branch.

The backtracking algorithm can be broken down into three main steps:

1. Select a starting point

The algorithm starts by selecting an initial solution or a starting point. This starting point is often an empty solution or a partial solution.

2. Explore possible solutions

The algorithm recursively explores all possible solutions by adding one element at a time. This is done by generating all possible solutions and evaluating each solution to determine if it is a valid solution.

3. Backtrack and explore another branch

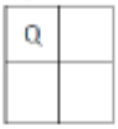
When a dead end is reached, the algorithm backtracks to a previous state and explores another branch. This process continues until a solution is found or all possible solutions have been exhausted.

8-QUEENS PROBLEM

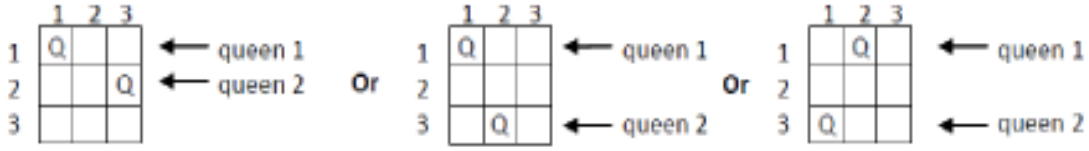
The 8-queens problem is probably the best know variation of the n-queens problem of placing n queens on an n× n chessboard, where solutions exist only for n = 1 or n ≥4. The question of the problem is how to place eight queens on a chess board in a way that they are not able to attack each other. There cannot be any 2 queens in the same horizontal, vertical or diagonal line. The goal is not just to find one possible solution, but all of them.

For n = 1, the problem has **a trivial solution**.

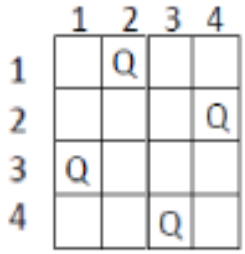
For n = 2, it is easy to see that there is **no solution** to place 2 queens in 2 × 2 chessboard.



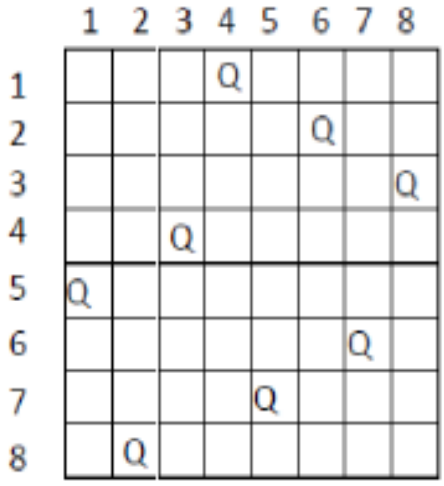
For n = 3, it is easy to see that there is **no solution** to place 3 queens in 3 × 3 chessboard.



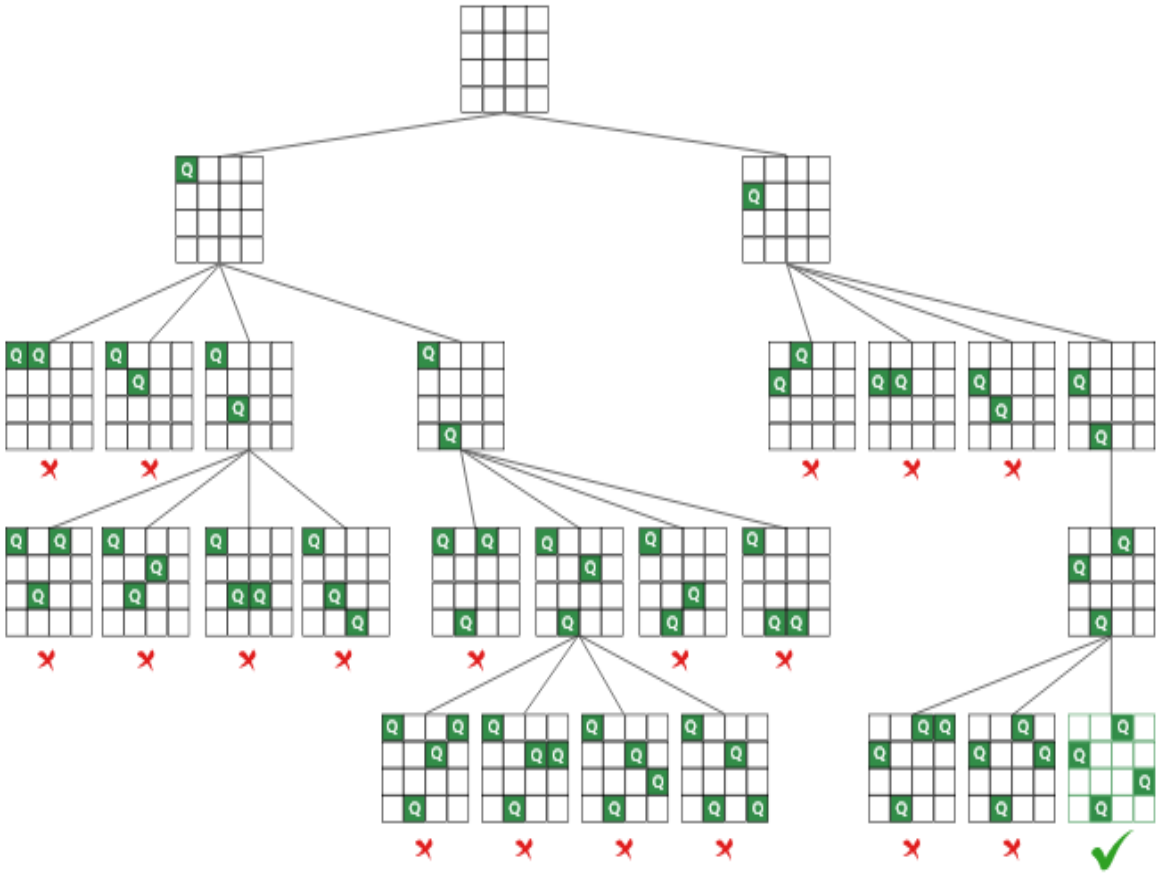
For n = 4, There is **solution** to place 4 queens in 4 × 4 chessboard. the four-queens problem solved by the backtracking technique.



For n = 8, There is **solution** to place 8 queens in 8 × 8 chessboard.



State space tree for 4- queens problem is



SUM OF SUBSET PROBLEM

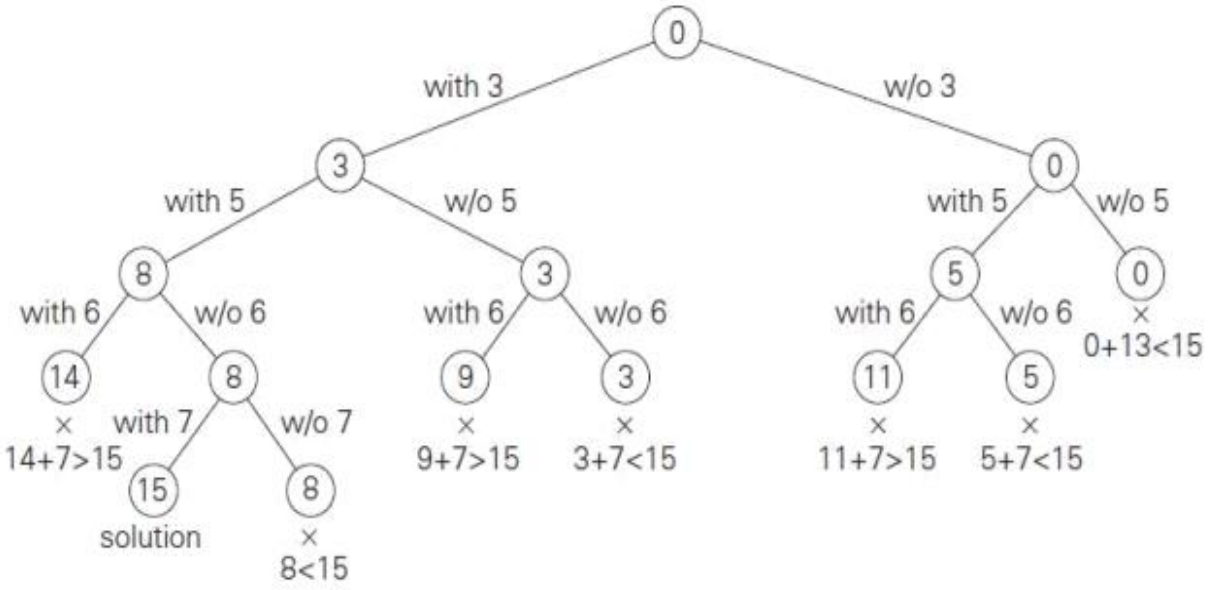
The Sum of Subset problem involves finding a subset of numbers from a given set that sums up to a specified target.

Steps:

1. Initialization: Start with an empty subset and a sum of zero.
2. Recursive Function:
    - Base Case: If the current sum equals the target, return success; if it exceeds the target or all elements have been considered, return failure.
    - For each element, decide to either:
        - Include the element in the current subset and recursively check for the new sum.
        - Exclude the element and recursively check the next elements.
3. Backtracking: If including an element doesn't lead to a solution, backtrack by removing it from the current subset and exploring other possibilities.
4. Output: If a valid subset is found, return it; otherwise, indicate that no such subset exists.

State the subset-sum problem and Complete state-space tree of the backtracking algorithm applied to the instance A={3, 5, 6, 7} and d=15 of the subset-sum problem.

The subset-sum problem finds a subset of a given set A = {a1, . . . , an} of n positive integers whose sum is equal to a given positive integer d. For example, for A = {1, 2, 5, 6, 8} and d = 9, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions. It is convenient to sort the set‘s elements in increasing order. So, we will assume that a1< a2 < . . . < an. For Example, A = {3, 5, 6, 7} and d = 15 of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.



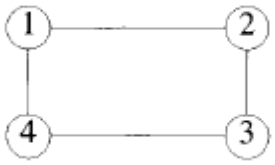
GRAPH COLORING

Graph coloring using backtracking involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The process typically follows these steps:

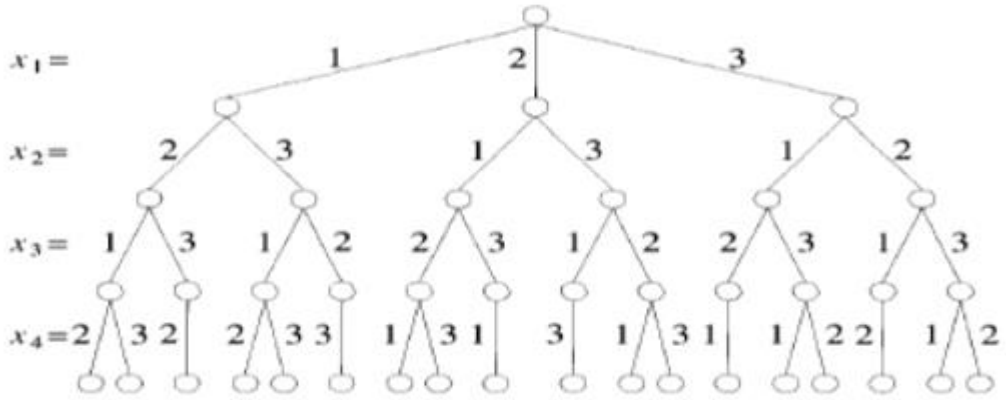
1. **Initialization**: Start with an empty coloring array.
2. **Recursive Function**:
    - If all vertices are colored, return success.
    - For each vertex, attempt to color it with available colors.
    - Check if the color is valid (i.e., no adjacent vertices have the same color).
    - If valid, recursively try to color the next vertex.
    - If no valid color leads to a solution, backtrack (reset the color).
3. **Output**: If a valid coloring is found, return the coloring; otherwise, indicate failure.

Example:-

Given a graph contain four vertex and given no of colors (m) is 3



The state space tree for the given problem is



0/1 KNAPSACK PROBLEM

The 0/1 Knapsack problem is an optimization challenge where you have a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity. The objective is to maximize the total value of the items packed into the knapsack without exceeding its weight limit.

Backtracking Approach

1. **Recursive Exploration**: The backtracking method explores all possible combinations of items. For each item, you decide whether to include it in the knapsack or exclude it.
2. **Base Case:** If all items have been considered or the weight exceeds the capacity, you return the accumulated value.
3. **Recursive Case:** For each item, you:
    - Include the item (if it fits) and call the function recursively.
    - Exclude the item and call the function recursively.
    - Compare the values from both options and choose the maximum.

Complexity

- **Time Complexity**: O(2^n) due to the exploration of all combinations.
- **Space Complexity**: O(n) for the recursion stack.

items
A B C D
profit 1 6 10 16
weight 1 2 3 5
Capacity: 7
values: [A, B, C, D]
selected: []
profit: 0, weight:0
select A
skip A
values: [B, C, D]
selected: [A]
profit: 1, weight:1
values: [B, C, D]
selected: []
profit: 0, weight:0
select B
skip B
values: [C, D]
selected: [A, B]
profit: 7, weight:3
values: [C, D]
selected: [A]
profit: 1, weight:1
values: [C, D]
selected: [B]
profit: 6, weight:2
values: [C, D]
selected: []
profit: 0, weight:0
select C
skip C
values: [D]
selected: [A, B, C]
profit: 17
weight: 6
values: [D]
selected: [A, B]
profit: 7
weight:3
values: [D]
selected: [A, C]
profit: 11
weight:4
values: [D]
selected: [A]
profit: 1
weight:1
values: [D]
selected: [B, C]
profit: 16
weight:5
values: [D]
selected: [B]
profit: 6
weight:2
values: [D]
selected: [C]
profit: 10
weight:3
values: [D]
select: []
profit: 0
weight:0
select D
skip D
values: []
select: [A, B, C, D]
profit: 33
weight: 11
values: []
select: [A, B, C]
profit: 17
weight: 6
values: []
select: [A, B, D]
profit: 23
weight: 8
values: []
select: [A, B]
profit: 7
weight: 3
values: []
select: [A, C, D]
profit: 27
weight: 9
values: []
select: [A, C]
profit: 11
weight: 4
values: []
select: [A, D]
profit: 16
weight: 6
values: []
select: [A]
profit: 1
weight:1
values: []
select: [B, C, D]
profit: 32
weight:10
values: []
select: [B, C]
profit: 16
weight: 5
values: []
select: [B, D]
profit: 22
weight: 7
values: []
select: [B]
profit: 6
weight: 2
values: []
select: [C, D]
profit: 26
weight: 8
values: []
select: [C]
profit: 10
weight: 3
values: []
select: [D]
profit: 16
weight: 5
values: []
select: []
profit: 0
weight:0