UNIT

5

Backtracking & Branch and Bound

Backtracking

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

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.

When to use a Backtracking algorithm?

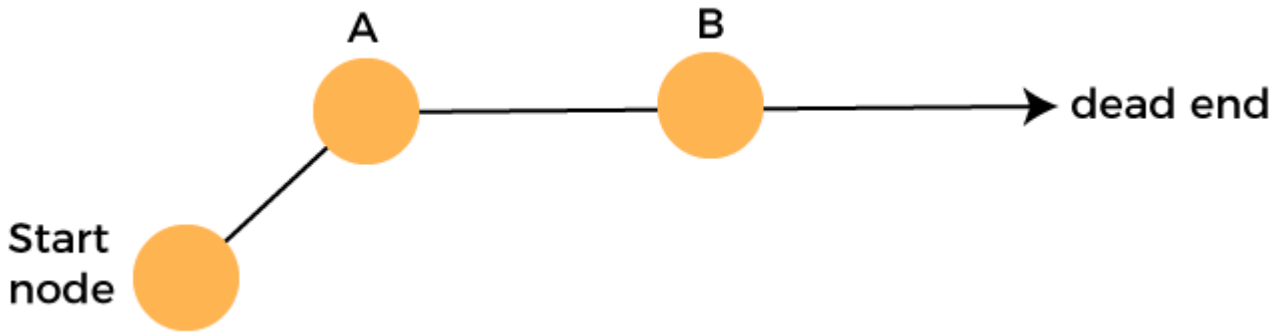
When we have multiple choices, then we make the decisions from the available choices. In the following cases, we need to use the backtracking algorithm:

- 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.

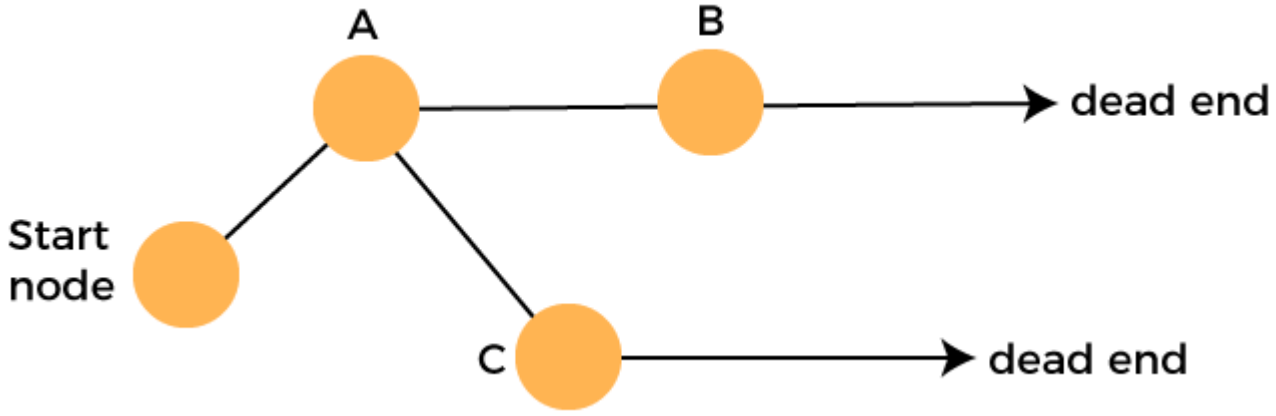
How does Backtracking work?

Backtracking is a systematic method of trying out various sequences of decisions until you find out that works. Let's understand through an example.

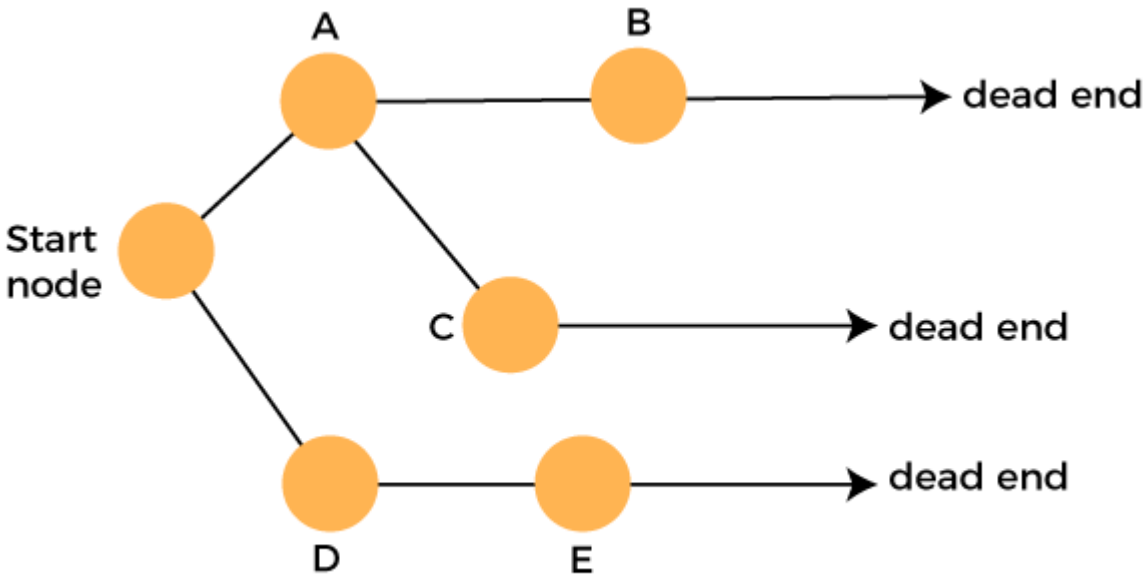
We start with a start node. First, we move to node A. Since it is not a feasible solution so we move to the next node, i.e., B. B is also not a feasible solution, and it is a dead-end so we backtrack from node B to node A.



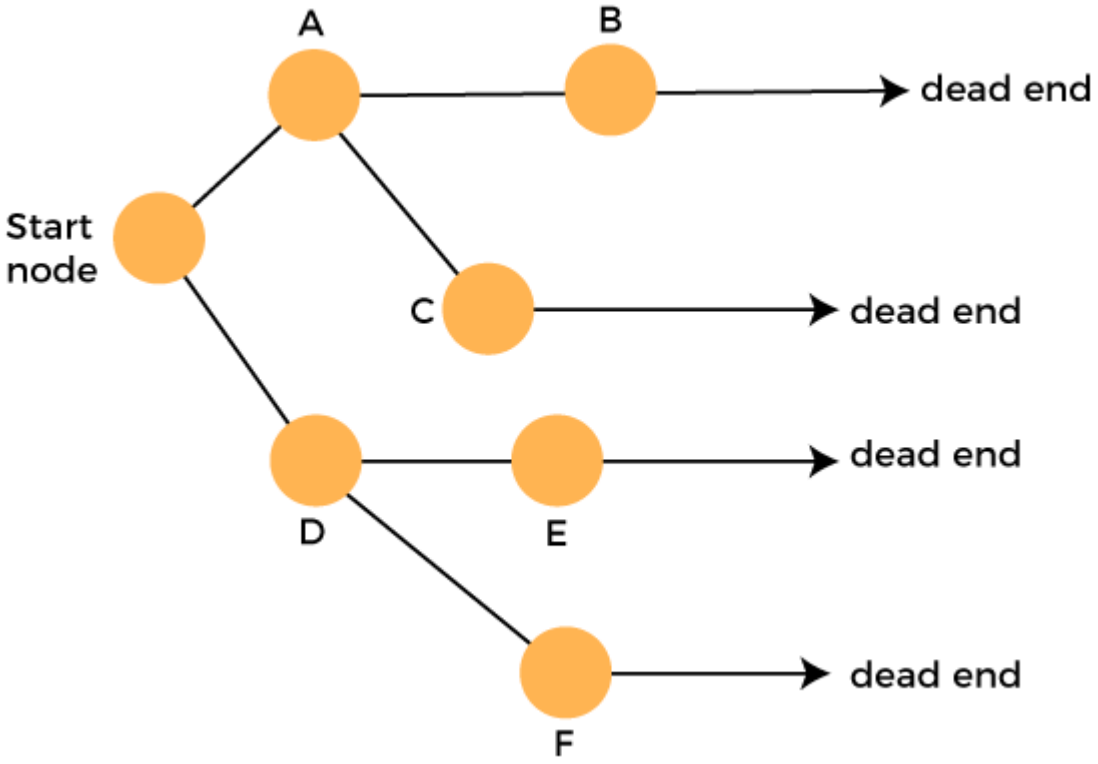
Suppose another path exists from node A to node C. So, we move from node A to node C. It is also a dead-end, so again backtrack from node C to node A. We move from node A to the starting node.



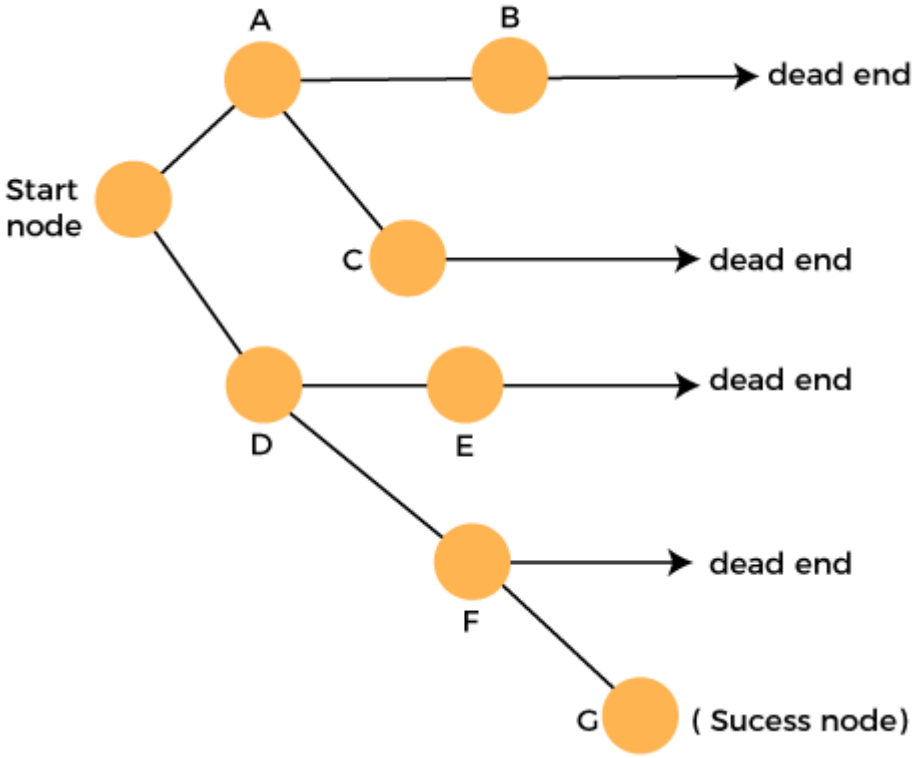
Now we will check any other path exists from the starting node. So, we move from start node to the node D. Since it is not a feasible solution so we move from node D to node E. The node E is also not a feasible solution. It is a dead end so we backtrack from node E to node D.



Suppose another path exists from node D to node F. So, we move from node D to node F. Since it is not a feasible solution and it's a dead-end, we check for another path from node F.



Suppose there is another path exists from the node F to node G so move from node F to node G. The node G is a success node.



The terms related to the backtracking are:

- **Live node:** The nodes that can be further generated are known as live nodes.
- **E node:** The nodes whose children are being generated and become a success node.
- **Success node:** The node is said to be a success node if it provides a feasible solution.
- **Dead node:** The node which cannot be further generated and also does not provide a feasible solution is known as a dead node.

Many problems can be solved by backtracking strategy, and that problems satisfy complex set of constraints, and these constraints are of two types:

- **Implicit constraint:** It is a rule in which how each element in a tuple is related.
- **Explicit constraint:** The rules that restrict each element to be chosen from the given set.

Applications of Backtracking

- N-queen problem
- Sum of subset problem
- Graph coloring
- Hamiliton cycle

Difference between the Backtracking and Recursion

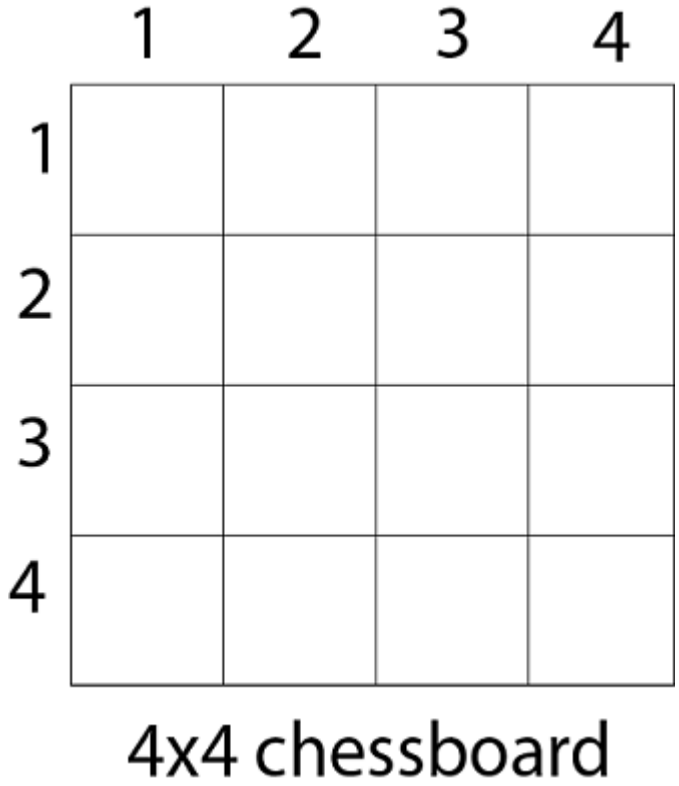
Recursion is a technique that calls the same function again and again until you reach the base case. Backtracking is an algorithm that finds all the possible solutions and selects the desired solution from the given set of solutions.

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.

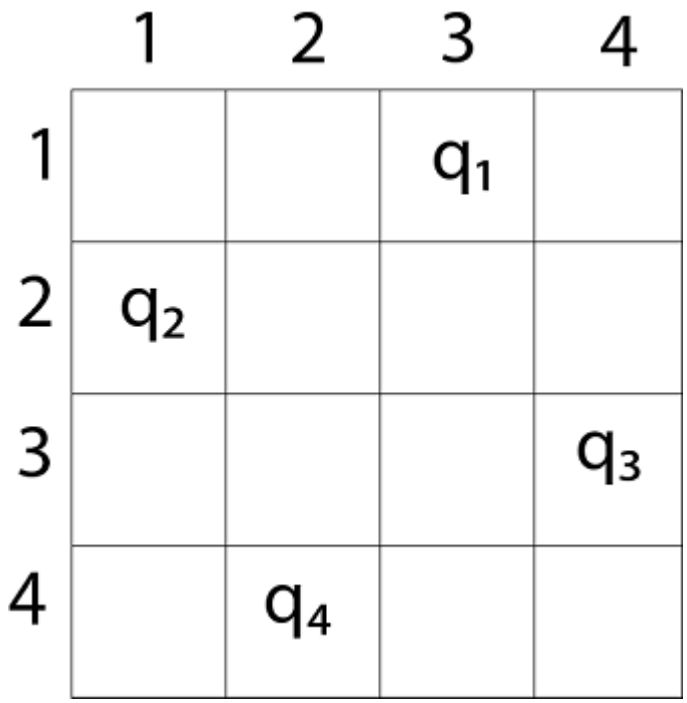
Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



4x4 chessboard

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.



The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:

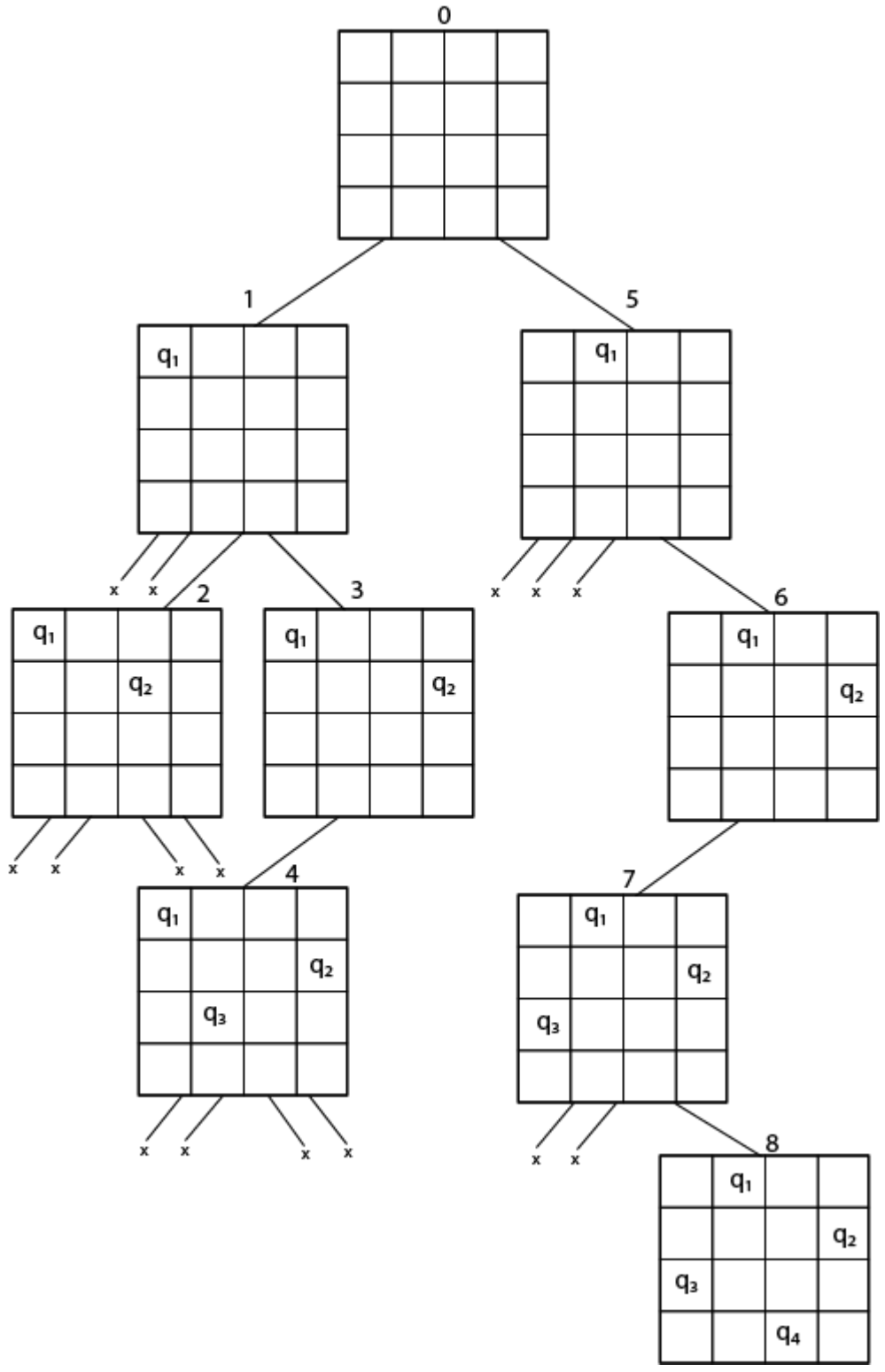
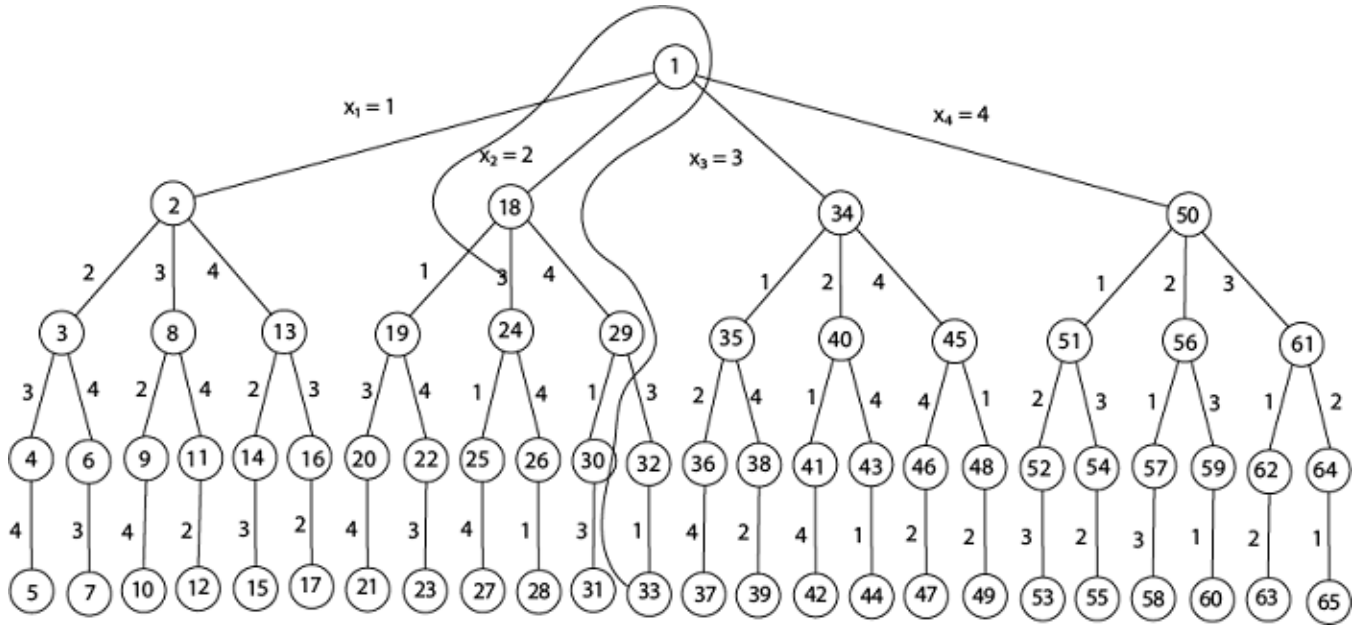


Fig shows the complete state space for 4 - queens problem. But we can use backtracking method to generate the necessary node and stop if the next node violates the rule, i.e., if two queens are attacking.



4 - Queens solution space with nodes numbered in DFS

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x_1, x_2, x_3, x_4) where x_i represents the column on which queen "q_i" is placed.

One possible solution for 8 queens problem is shown in fig:

	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.

5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x_1, x_2,....x_{k-1} and whether there is no other queen on the same diagonal.

Using place, we give a precise solution to then n- queens problem.

```
1.  Place (k, i)
2.    {
3.      For j ← 1 to k - 1
4.      do if (x [j] = i)
5.        or (Abs x [j]) - i) = (Abs (j - k))
6.     then return false;
7.      return true;
8.  }
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
1.  N - Queens (k, n)
2.  {
3.    For i ← 1 to n
4.        do if Place (k, i) then
5.    {
6.       x [k] ← i;
7.       if (k ==n) then
8.        write (x [1....n));
9.       else
10.      N - Queens (k + 1, n);
11.   }
12. }
```

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, 9],

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. This is an NP-complete problem again. It is the special case of knapsack

Let's understand this problem through an example.

problem.

We have a set of 5 integers given below:

N = 4, -2, 2, 3, 1

We want to find out the subset whose sum is equal to 5. There are many solutions to this problem.

The naïve approach, i.e., brute-force search generates all the possible subsets of the original array, i.e., there are 2n possible states. Here the running time complexity would be exponential. Then, we consider all these subsets in O(N) linear running time and checks whether the sum of the items is M or not.

The dynamic programming has pseudo-polynomial running time.

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.

1. Example 1:
2. Input: set[] = {4, 16, 5, 23, 12}, sum = 9
3. Output = true
4. Subset {4, 5} has the sum equal to 9.

5.
6. Example 2:
7. Input: set[] = {2, 3, 5, 6, 8, 10}, sum = 10
8. Output = true
9. There are three possible subsets that have the sum equal to 10.
10. Subset1: {5, 2, 3}
11. Subset2: {2, 8}
12. Subset3: {10}

There are two ways of solving the subset problem:

- Recursion
- Dynamic programming

Method 1: Recursion

Before knowing about the recursive approach, we should know about two things in a subset which are given below:

- **Include:** Here include means that we are selecting the element from the array.
- **Exclude:** Here, exclude means that we are rejecting the element from the array.

To implement the recursive approach, we consider the following two cases:

- Now we consider the first element and now the required sum is equal to the difference between the target sum and value of first element. The number of elements is equal to the difference between the total elements and 1.
- Leave the 'first' element and now the required sum = target sum. The number of elements is equal to the difference between the total elements and 1.

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

arr = [3, 4, 5, 2]

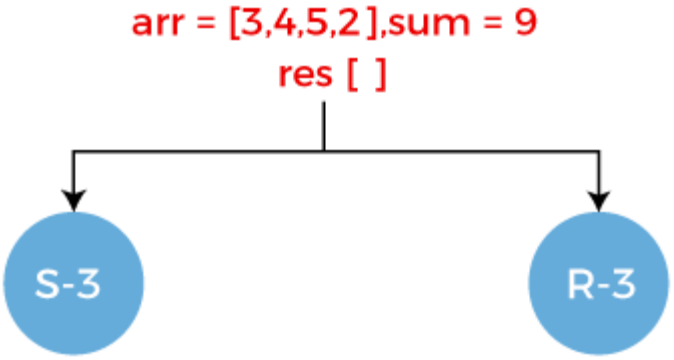
sum = 9

result = []

In the above example, we have taken an array, and the empty array named result that stores all the values whose resultant sum is equal to 9.

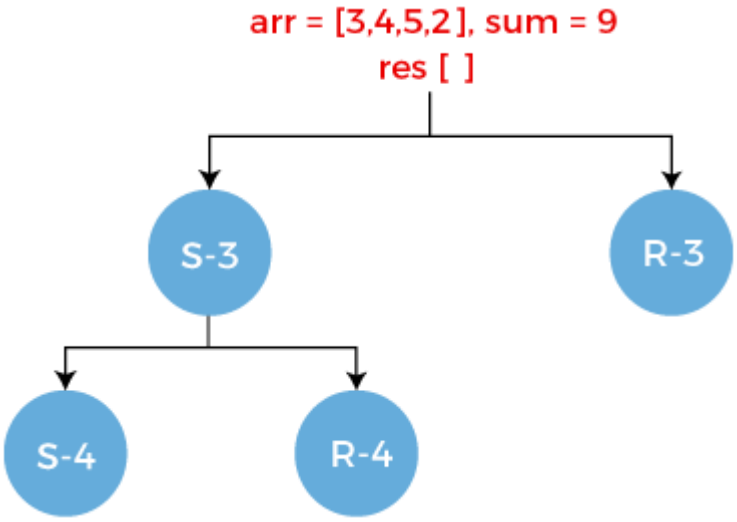
First element in an array is 3. There are two scenarios:

- First scenario is select. The sum is equal to the **target sum - value of first element, i.e.,** 9 - 3 = 6 and the first element, i.e., 3 gets stored in the result array, i.e., **result[]**.
- Second scenario is reject. The array arr contains the elements 4, 5, 2, i.e., arr = [4, 5, 2] and sum would be same as 9 as we are rejecting the element 3. The **result[]** array would remain empty.



Now we perform the same select and reject operation on element 4 as it is the first element of the array now.

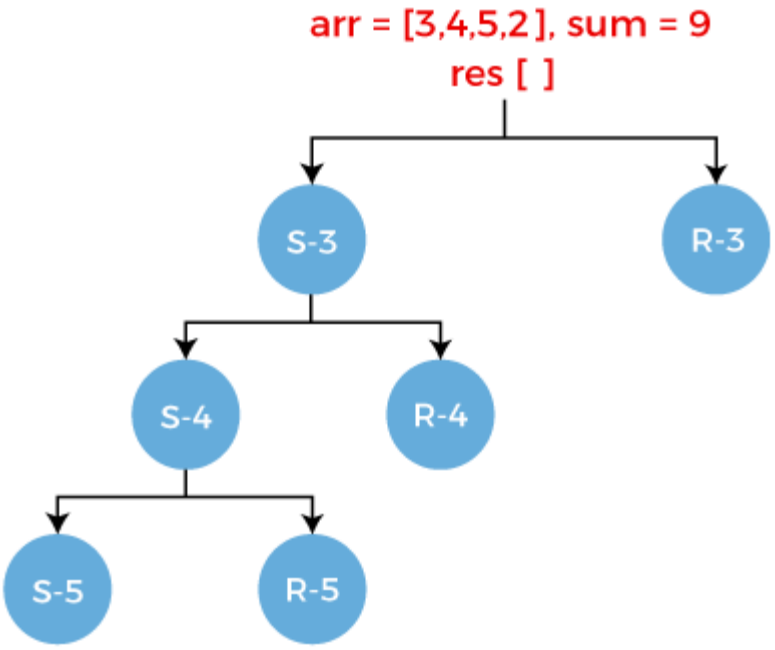
- Select the element 4 from the array. Since we are selecting 4 from the array so array arr would contain the elements 5, 2, i.e., arr = [5, 2]. The sum is equal to the 6-4 = 2 and the element 4 gets stored in the result arr. The result[] = {3, 4}.
- Reject the element 4 from the array. Since we are rejecting the 4 from the array so array arr would contain the elements 5, 2, i.e., arr = [5, 2]. The sum would remain same as 6 and the result array would be same as previous, i.e., {3}.



Now we perform the select and reject operation on element 5.

- Select the element 5 from the array. Since we are selecting 5 from the array so array arr would contain the elements 2, i.e., arr = [2]. The sum is equal to the 2 - 5 equals to -3 and the element 5 gets stored in the result arr. The result[] = {3, 4, 5}.
- Reject the element 5 from the array. Since we are rejecting 5 from the array so array arr would contain the element 2, i.e., arr = [2]. The sum would remain same as previous, i.e., 6 and the result array would be same

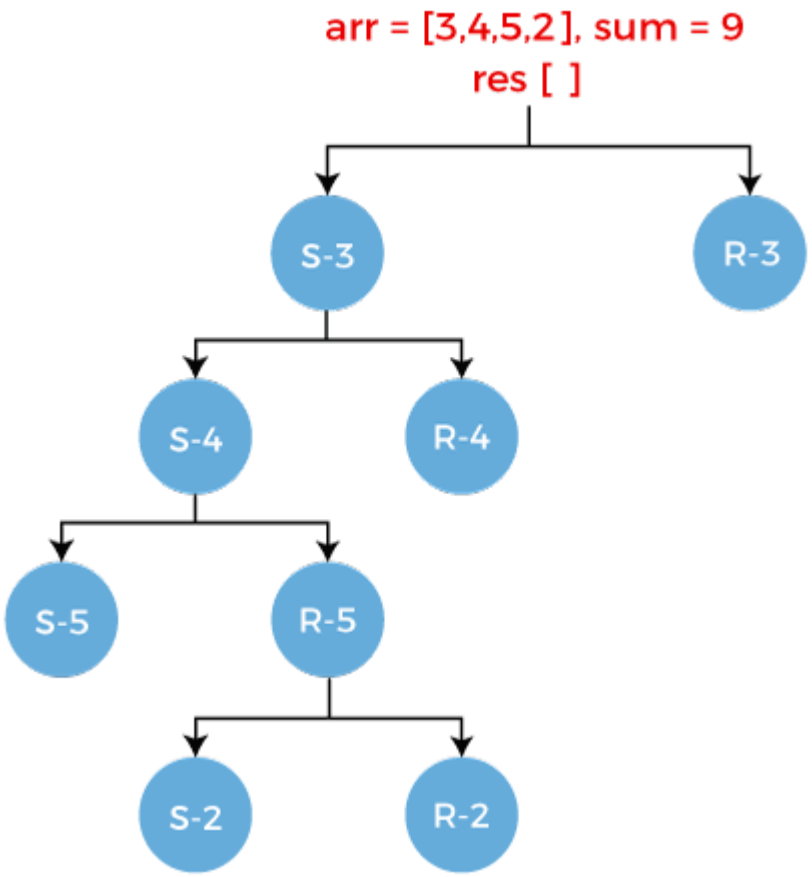
as previous, i.e., {3, 4}.



If we observe S-5, we can see that the sum is negative that returns false. It means that there is no further subset available in the set.

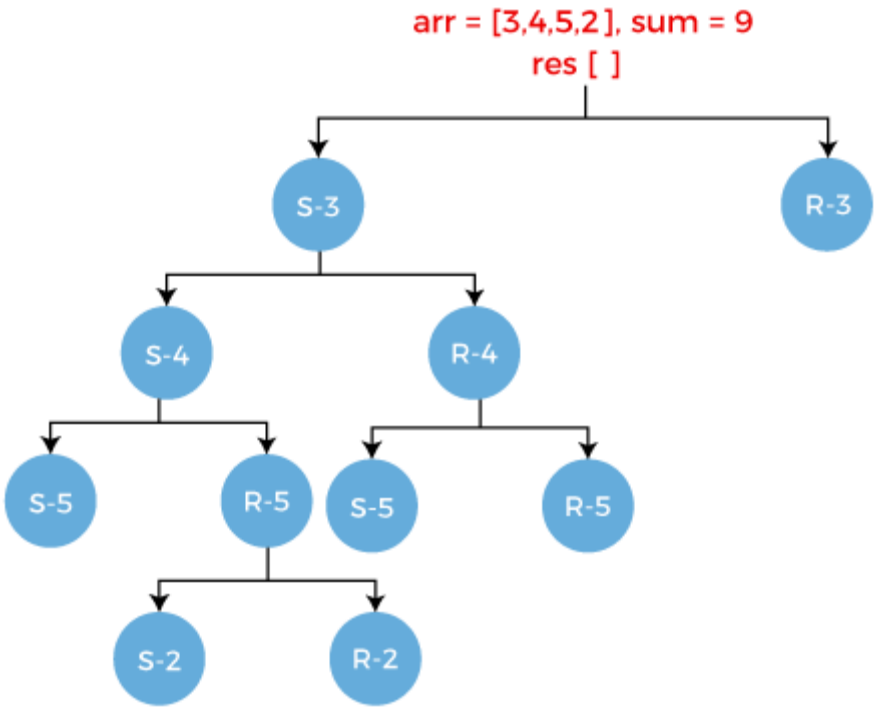
Consider R-5. It also has two scenarios:

- Select the element 2 from the array. Once the element 2 gets selected, the array becomes empty, i.e., arr[] = " ". The sum would be 2-2 equals to 0 and the element 2 gets stored in the result array. The result[] = [3, 4, 2].
- Reject the element 2 from the array. Once the element 2 gets rejected, the array becomes empty, i.e., arr[] = " ". The sum would be same as previous, i.e., 2 and the result array would also be same as previous, i.e., [3, 4].



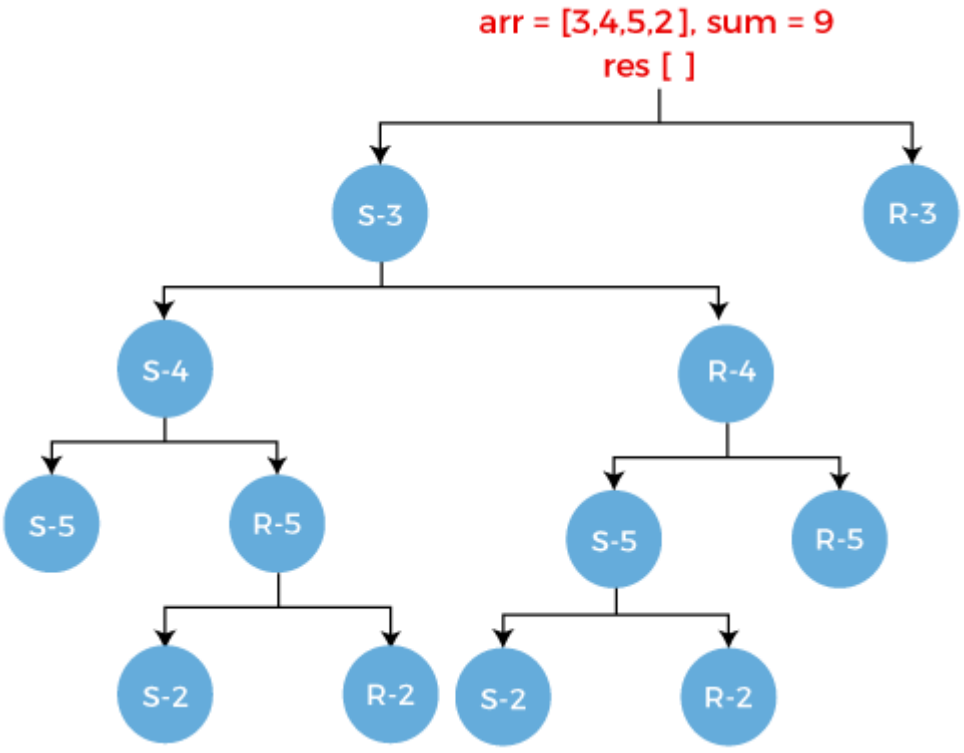
Consider R-4. It has two scenarios:

- Select the element 5 from the array. Since we are selecting 5 from the array so array arr would contain the elements 2, i.e., arr = [2]. The sum would be 6-5 equals to 1 and the element 5 gets stored in the result array. The result[] = [3, 5].
- Reject the element 5 from the array. Since we are rejecting 5 from the array so array arr would contain the element 2, i.e., arr = [2]. The sum would remain same as previous, i.e., 6 and the result array would be same as previous, i.e., {3}.



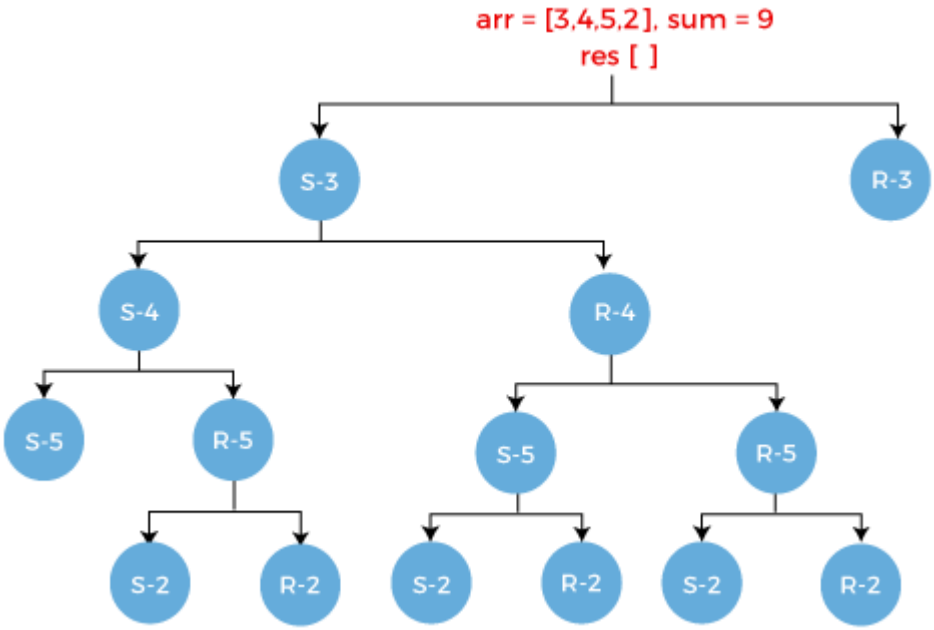
Consider S-5. It has two scenarios:

- Select the element 2 from the array. Since we are selecting 2 from the array so array arr would be empty, i.e., arr = " ". The sum would be 1-2 equals to -1 and the element 2 gets stored in the result array. The result[] = [3, 5, 2].
- Reject the element 2 from the array. Since we are rejecting 2 from the array so array arr would become empty. The sum would remain same as previous, i.e., 1 and the result array would be same as previous, i.e., {3, 5}.

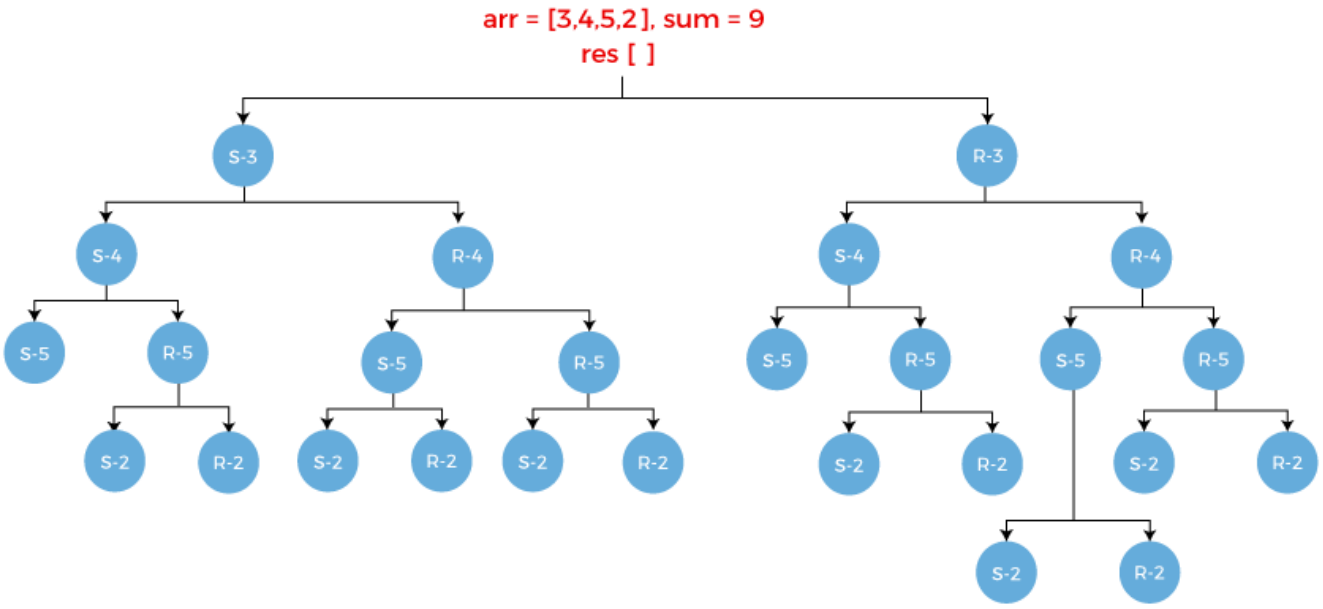


Consider R-5. It has two scenarios:

- Select the element 2 from the array. Since we are selecting 2 from the array so array arr would be empty, i.e., arr = " ". The sum would be 6-2 equals to 4 and the element 2 gets stored in the result array. The result[] = [3, 2].
- Reject the element 2 from the array. Since we are rejecting 2 from the array so array arr would become empty. The sum would remain same as previous, i.e., 6 and the result array would be same as previous, i.e., {3}.



Similarly, we get the reject case, i.e., R-3 as shown as below:



Following are the base conditions:

```
1. if sum == 0
2. return true
3. if sum < 0
4. return false
5. if (arr[] && sum!= 0)
6. return false
```

When we apply the base conditions on the above tree, then we will find two subsets given below:

S1 = {3, 4, 2}

S2 = {4, 5}

Implementation

```
1. def subset_sum(arr, res, sum)
2.  if sum ==0
3.  return true
4.  if sum < 0
5. return false
6. if len(arr) == 0 and sum!= 0
7. return false
8. arr.pop(0);
9. if len(arr) > 0
10.  res.append(arr[0])
11. select = subset_sum(arr, sum-arr[0], res)
12. reject = subset_sum(arr, res, sum)
13. return reject or sum
```

Method 2: Dynamic Programming

Let A be an array or set which contains 'n' non-negative integers. Find a subset 'x' of set 'A' such that the sum of all the elements of x is equal to w where x is another input (sum).

For example:

A = {1, 2, 5, 9, 4}

Sum(w) = 18

Now we have to find out the subset from the given set whose sum is equal to 18. Here we will use the dynamic programming approach to solve the subset sum problem.

Example:

A = [2, 3, 5, 7, 10]

Sum(w) = 14

First, we create a table. The column contains the values from 0 to 14 while row contains the elements of the given set shown as below:

In the below table:

i: It represents the rows. Rows represents the elements.

j: It represents the columns. Columns represent the sum.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2															
3															
5															
7															
10															

Consider the element 2. We will use 1 as a true value and 0 as a false value. The value 1 comes under 0 and 2 columns shown as below:

Here i=1, a[i] =2

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3															
5															
7															
10															

> ***Note: The rule for filling each column is given below:***
> ***Required sum = j - element***
> ***A[i][j] = A[i-1][required sum]***

When j= 1

Required sum = 1 - 2 = -1; Since the sum is negative so put 0 under the column 1 as shown in the above table.

When j= 2

Required sum = 2 - 2 = 0; Since the value of sum is zero so we put 1 under the column 2 as shown in the above table.

We put 0 under the columns whose sum is greater than 2 as we cannot make sum more than 2 from the element 2.

Consider the element 3.

Here i = 2, a[i] = 3

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0
5															
7															
10															

The columns whose sum is less than 3 will have the same values as the previous columns.

When j = 3, sum[j] = 3

Required sum = 3 -3 = 0; since the sum is zero so we put 1 under the column 3 as shown in the above table.

When j = 4; sum[j] = 4

Required sum = 4 - 3 = 1; Since the sum is 1 so we move to the previous row, i.e., i=1 and j=1. The value at a[1][1] is 0 so we put 0 at a[2][4].

When j = 5, sum[j] = 5

Required sum = 5 -3 = 2; The value of sum is 2 so the value at a[1][2] equals to 1. Therefore, the value at a[2][5] would be 1.

When j = 6, sum[j] = 6

Required sum = 6 -3 = 3; The value of sum is 3 so the value at a[1][3] equals to 0. Therefore, the value at a[2][6] would be 0.

When j = 7, sum[7] = 7

Required sum = 7 - 3 = 4; The value of sum is 4 so the value at a[1][4] equals to 0. Therefore, the value at a[2][7] would be 0.

In this way, we get the value 0 from the columns 8 to 14.

Consider the element 5.

Here i=3, a[i] = 5

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0
5	1	0	1	1	0	1	0	1	1	0	1	0	0	0	0
7															
10															

The columns whose sum is less than 5 will have the same values as the previous columns.

When j =5, sum[j] = 5

Required sum = 5-5 = 0; Since the value of sum is 0; therefore, the value at a[2][5] equals to 1.

When j = 6, sum[j] = 6

Required sum = 6-5 = 1; the value of sum is 1 so the value at a[2][1] equals to 0; therefore, the value at a[3][6] equals to 0.

When j=7, sum[j] = 7

Required sum = 7-5 = 2; the value of sum is 2 so the value at a[2][2] equals to 1; therefore, the value at a[3][7] equals to 1.

When j=8, sum[j] = 8

Required sum = 8-5 = 3; the value of sum is 3 so the value at a[2][3] equals to 1; therefore, the value at a[3][8] equals to 1.

When j=9, sum[j] =9

Required sum= 9-5 = 4; the value of sum is 4 so the value at a[2][4] equals to 0; therefore the value at a[3][9] equals to 0.

In this way, we get the values from the columns 10 to 14.

Consider the element 7.

Here i=4, a[i] =7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0
5	1	0	1	1	0	1	0	1	1	0	1	0	0	0	0
7	1	0	1	1	0	1	0	1	1	1	1	0	1	0	1
10															

The columns whose sum is less than 7 will have the same values as the previous columns.

When j=9, sum[j] = 9

Required sum = 9 - 7 = 2; the value of sum is 2 so the value at a[3][2] equals to 1; therefore the value at a[4][9] equals to 1.

When j=10, sum[j] = 10

Required sum = 10 - 7= 3; the value of sum is 3 so the value at a[3][3] equals to 1; therefore, the value at a[4][10] equals to 1.

When j=11, sum[j] =11

Required sum = 11-7 = 4; the value of sum is 4 so the value at a[3][4] equals to 0; therefore, the value at a[4][11] equals to 0.

When j=12, sum[j] = 12

Required sum = 12-7 = 5; the value of sum is 5 so the value at a[3][5] equals to 1; therefore, the value at a[4][12] equals to 1.

When j=13, sum[j] =13

Required sum= 13 - 7 = 6; the value of sum is 6 so the value at a[3][6] equals to 0; therefore, the value at a[4][13] equals to 0.

When j=14, sum[j] = 14

Required sum = 14 - 7 = 7; the value of sum is 7 so the value at a[3][7] equals to 1; therefore, the value at a[4][14] equals to 1.

Consider the element 10

Here i=5, a[i] = 10

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0
5	1	0	1	1	0	1	0	1	1	0	0	0	0	0	0
7	1	0	1	1	0	1	0	1	0	1	1	0	1	0	1
10	1	0	1	1	0	1	0	1	0	1	1	0	1	1	1

The columns whose sum is less than 10 will have the same values as the previous columns.

When j = 10, sum[j] = 10

Required sum = 10 - 10 = 0; the value of sum is 0 so the value at a[4][0] equals to 1; therefore, the value at a[5][10] equals to 1.

When j = 11, sum[j] = 11

Required sum = 11 - 10 = 1; the value of sum is 1 so the value at a[4][1] equals to 0; therefore, the value at a[5][11] equals to 0.

When j=12, sum[j] = 12

Required sum = 12-10 = 2; the value of sum is 2 so the value at a[4][2] equals to 1; therefore, the value at a[5][12] equals to 1.

When j=13, sum[j] = 13

Required sum = 13 - 10 = 3; the value of sum is 3 so the value at a[4][3] equals to 1; therefore, the value at a[5][13] equals to 1.

To determine whether the above given problem contains the subset or not, we need to check the last row and the last column. If the value is 1 which means that there would exist atleast one subset.

1. We have basically followed three conditions where we write 1 in the cell of the table:
2. • A[i] = j
3. • A[i-1][j] = 1
4. • A[i-1][j-A[i]] = 1

Graph Coloring Problem

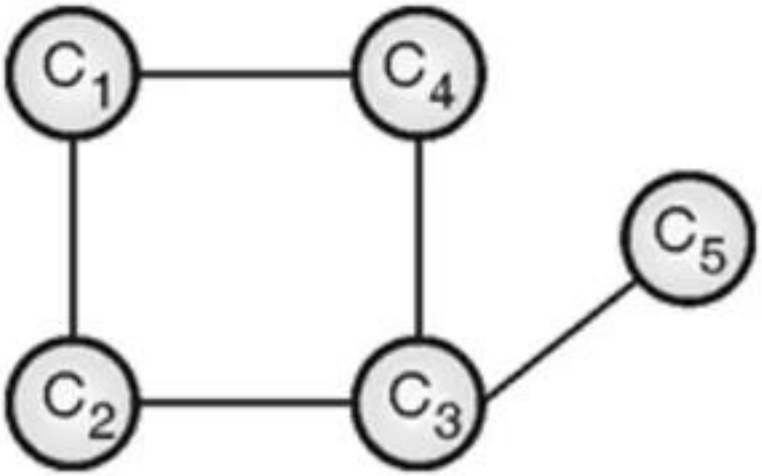
Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the **vertex coloring** problem.

- If coloring is done using at most k colors, it is called **k-coloring**.
- The smallest number of colors required for coloring graph is called its **chromatic number**.
- The chromatic number is denoted by X(G). Finding the chromatic number for the graph is NP-complete problem.
- Graph coloring problem is both, decision problem as well as an optimization problem. A decision problem is stated as, “With given M colors and graph G, whether such color scheme is possible or not?”.
- The optimization problem is stated as, “Given M colors and graph G, find the minimum number of colors required for graph coloring.”
- Graph coloring problem is a very interesting problem of graph theory and it has many diverse applications. Few of them are listed below.

Applications of Graph Coloring Problem

- Design a timetable.
- Sudoku
- Register allocation in the compiler
- Map coloring
- Mobile radio frequency assignment:

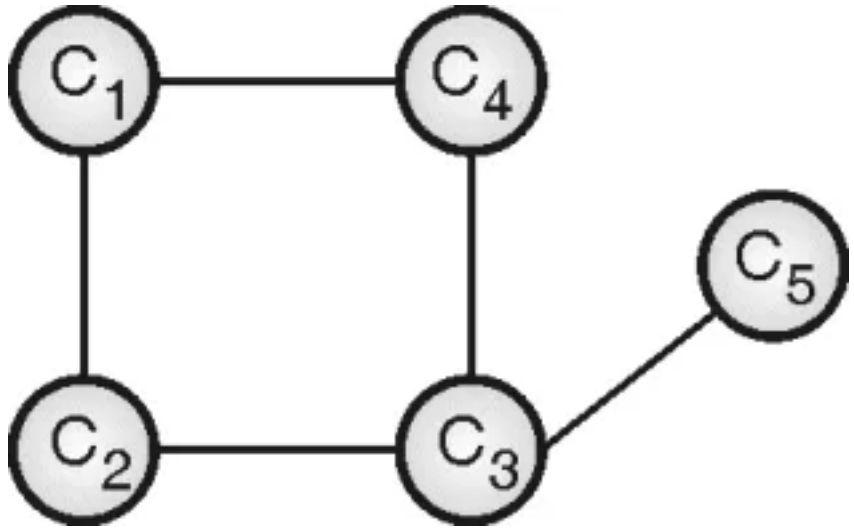
The input to the graph is an adjacency matrix representation of the graph. Value M(i, j) = 1 in the matrix represents there exists an edge between vertex i and j. A graph and its adjacency matrix representation are shown in Figure (a)



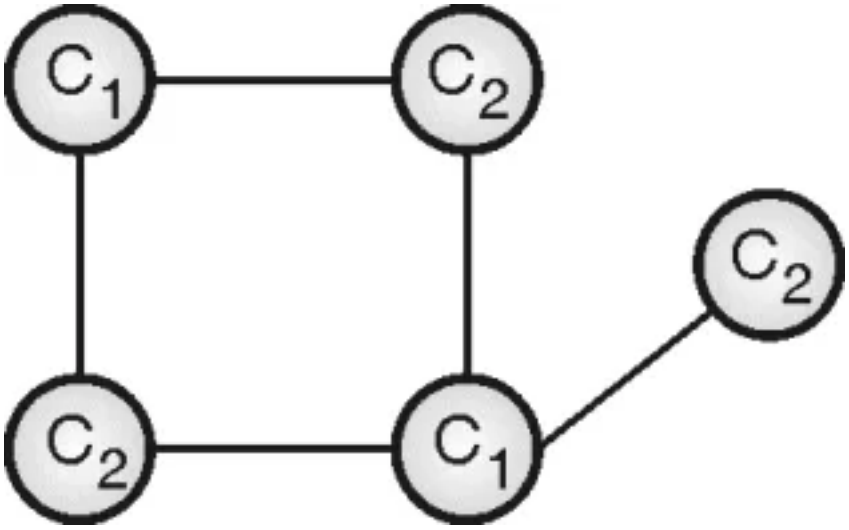
	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

Adjacency matrix for graph G

The problem can be solved simply by assigning a unique color to each vertex, but this solution is not optimal. It may be possible to color the graph with colors less than |V|. Figure (b) and figure (c) demonstrate both such instances. Let Ci denote the i^{th} color.



(b). Nonoptimal solution (uses 5 colors)



(c). Optimal solution (uses 2 colors)

- This problem can be solved using backtracking algorithms as follows:
  - List down all the vertices and colors in two lists
  - Assign color 1 to vertex 1
  - If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.
  - Repeat the process until all vertices are colored.
- Algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects next color i + 1 and test is repeated.
  Consider the graph shown in
  Figure (d)

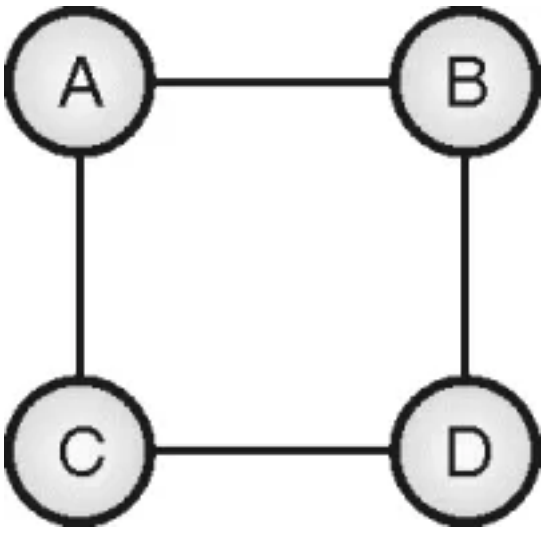


Figure (d)

If we assign color 1 to vertex A, the same color cannot be assigned to vertex B or C. In the next step, B is assigned some different colors 2. Vertex A is already colored and vertex D is a neighbor of B, so D cannot be assigned color 2. The process goes on. State-space tree is shown in Figure (e)

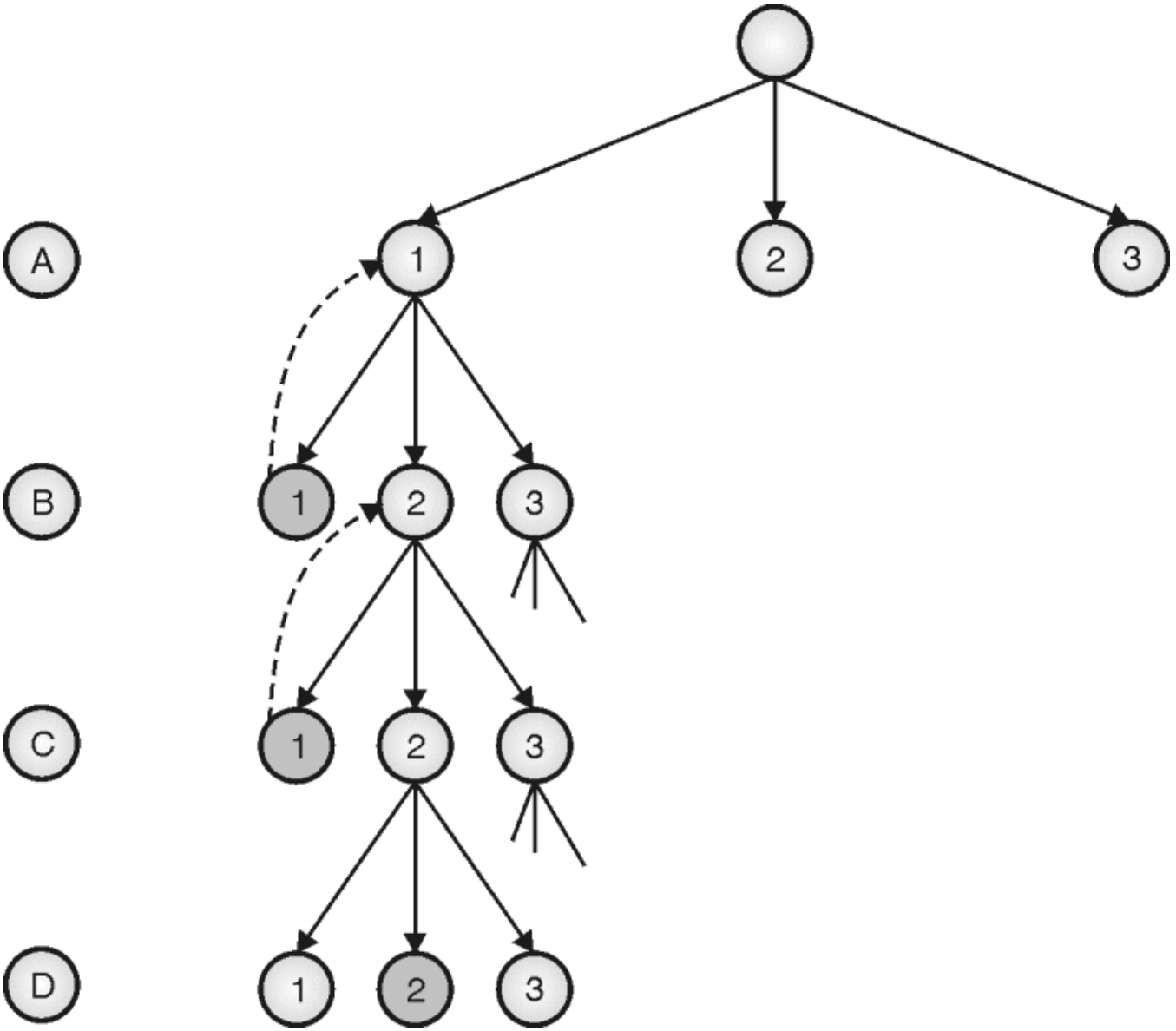


Figure (e): State-space tree of the graph of Figure (d)

Thus, vertices A and C will be colored with color 1, and vertices B and D will be colored with color 2.

Algorithm

```
 GRAPH_COLORING(G, COLOR, i)

// Description :  Solve the graph coloring problem
using backtracking

// Input : Graph G with n vertices, list of colors,
initial

vertex i
```

```
// COLOR[1…n] is the array of n different colors



// Output : Colored graph with minimum color


```

if

```
 CHECK_VERTEX(i) == 1
```

then

```


```

if

```
i == N
```

then

```


```

print

```
COLOR[1…n]


```

else

```


    j ← 1


```

while

```
(j ≤ M)
```

do

```


      COLOR(i + 1) ← j

      j ← j + 1


```

end

```
end
```

```
end
```

Function

```
CHECK_VERTEX(i)
```

for

```
j ← 1 to i – 1
```

do

if

```
Adjacent(i, j)
```

then

if

```
COLOR(i) == COLOR(j)
```

then

return

```
0
```

end

```
end
```

```
end
```

```
return
```

```
1
```

Complexity Analysis

The number of anode increases exponentially at every level in state space tree. With M colors and n vertices, total number of nodes in state space tree would be

$1 + M + M^2 + M^3 + ... + M^n$

Hence, $T(n) = 1 + M + M^2 + M^3 + ... + M^n$

$=fracM^{n+1}-1M-1$

So, $T(n) = O(M^n)$.

Thus, the graph coloring algorithm runs in exponential time.

Branch and bound

What is Branch and bound?

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.

Let's understand through an example.

Jobs = {j1, j2, j3, j4}

P = {10, 5, 8, 3}

d = {1, 2, 1, 2}

The above are jobs, problems and problems given. We can write the solutions in two ways which are given below:

Suppose we want to perform the jobs j1 and j2 then the solution can be represented in two ways:

The first way of representing the solutions is the subsets of jobs.

S1 = {j1, j4}

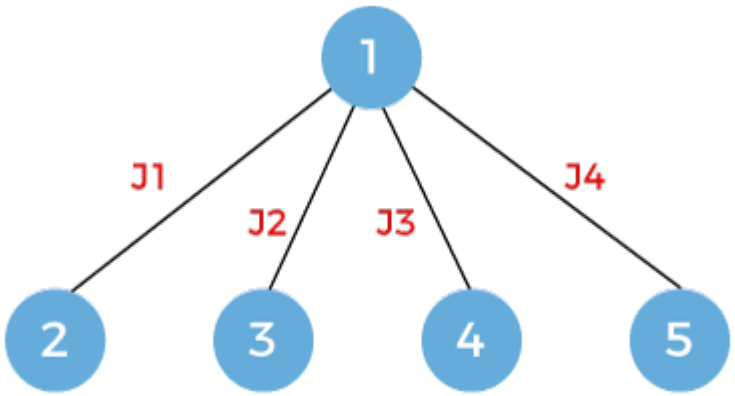
The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.

S2 = {1, 0, 0, 1}

The solution s1 is the variable-size solution while the solution s2 is the fixed-size solution.

First, we will see the subset method where we will see the variable size.

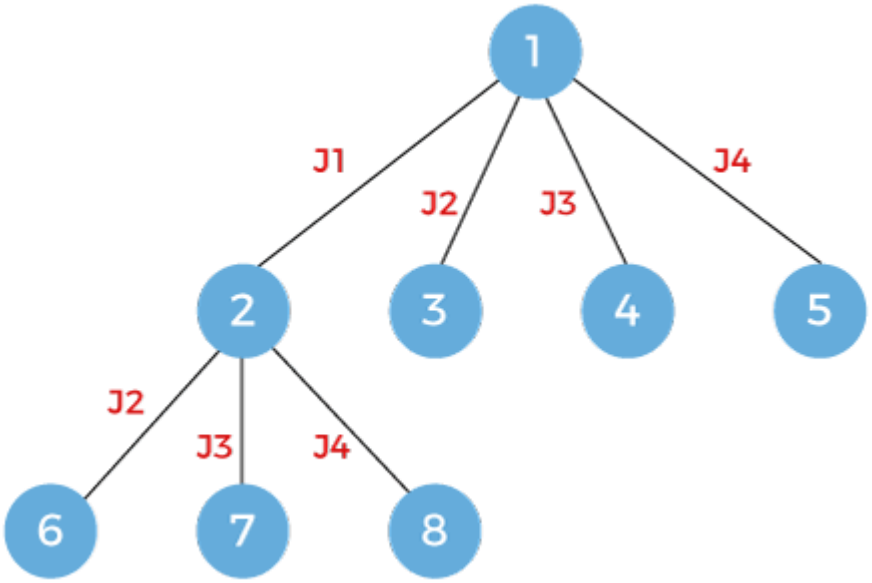
First method:



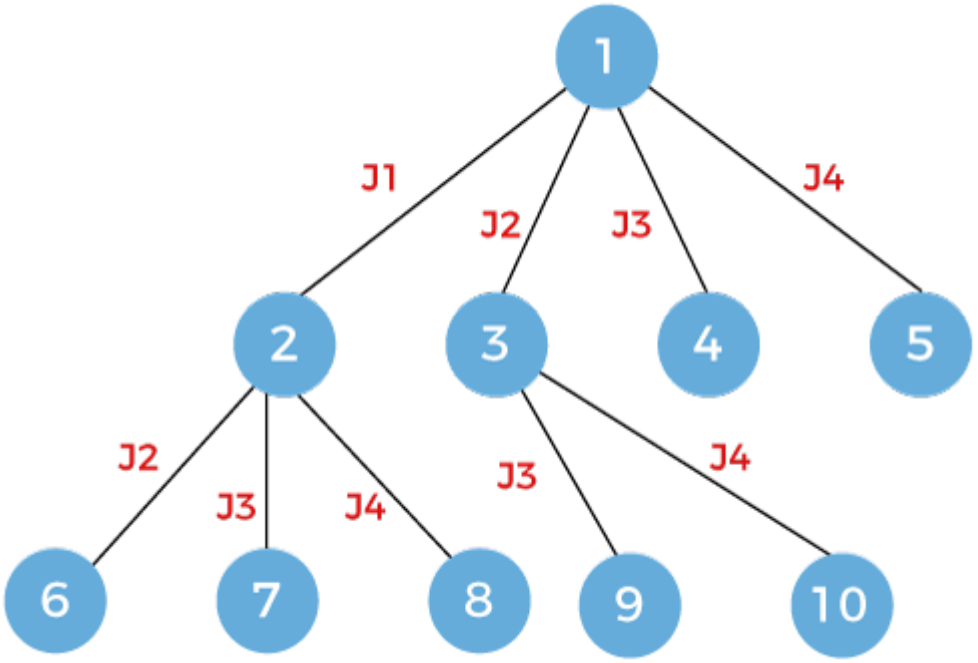
In this case, we first consider the first job, then second job, then third job and finally we consider the last job.

As we can observe in the above figure that the breadth first search is performed but not the depth first search. Here we move breadth wise for exploring the solutions. In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.

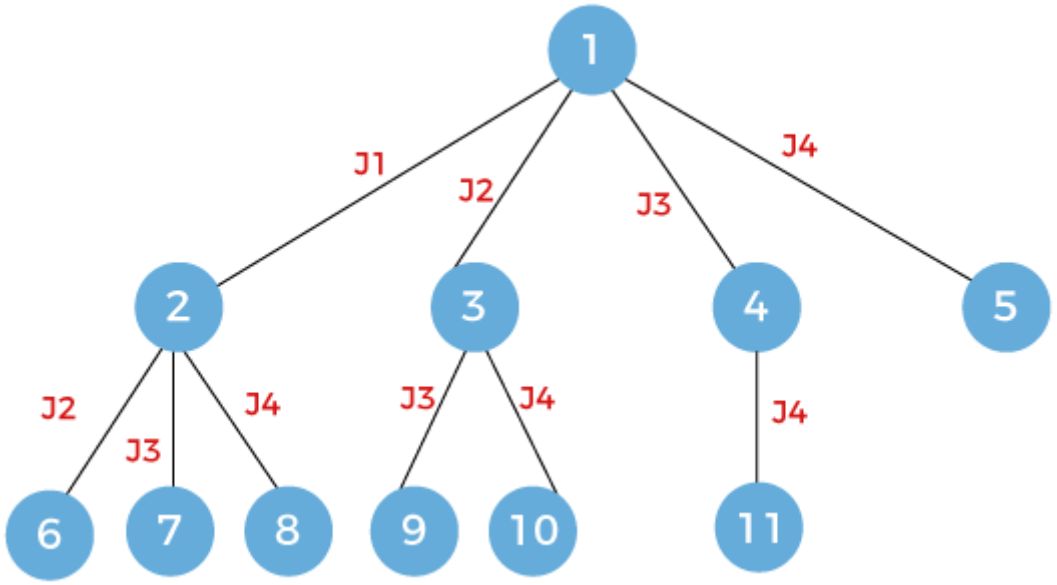
Now one level is completed. Once I take first job, then we can consider either j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.



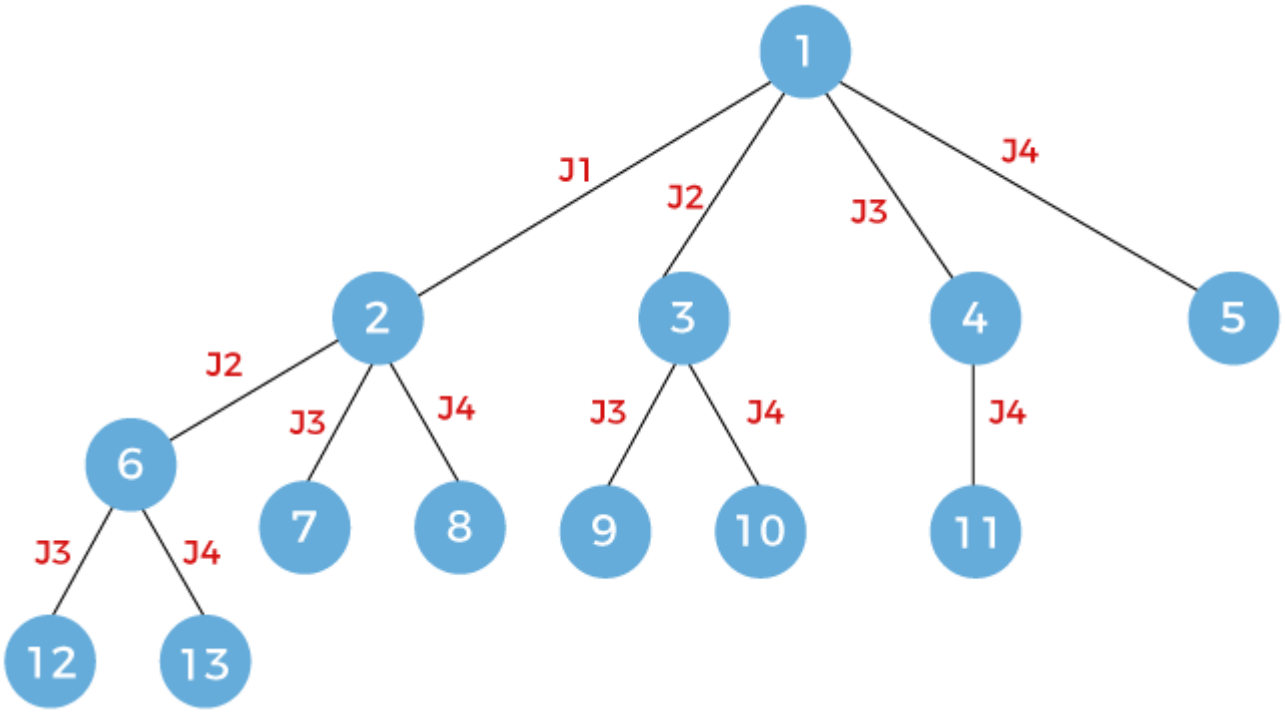
Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.



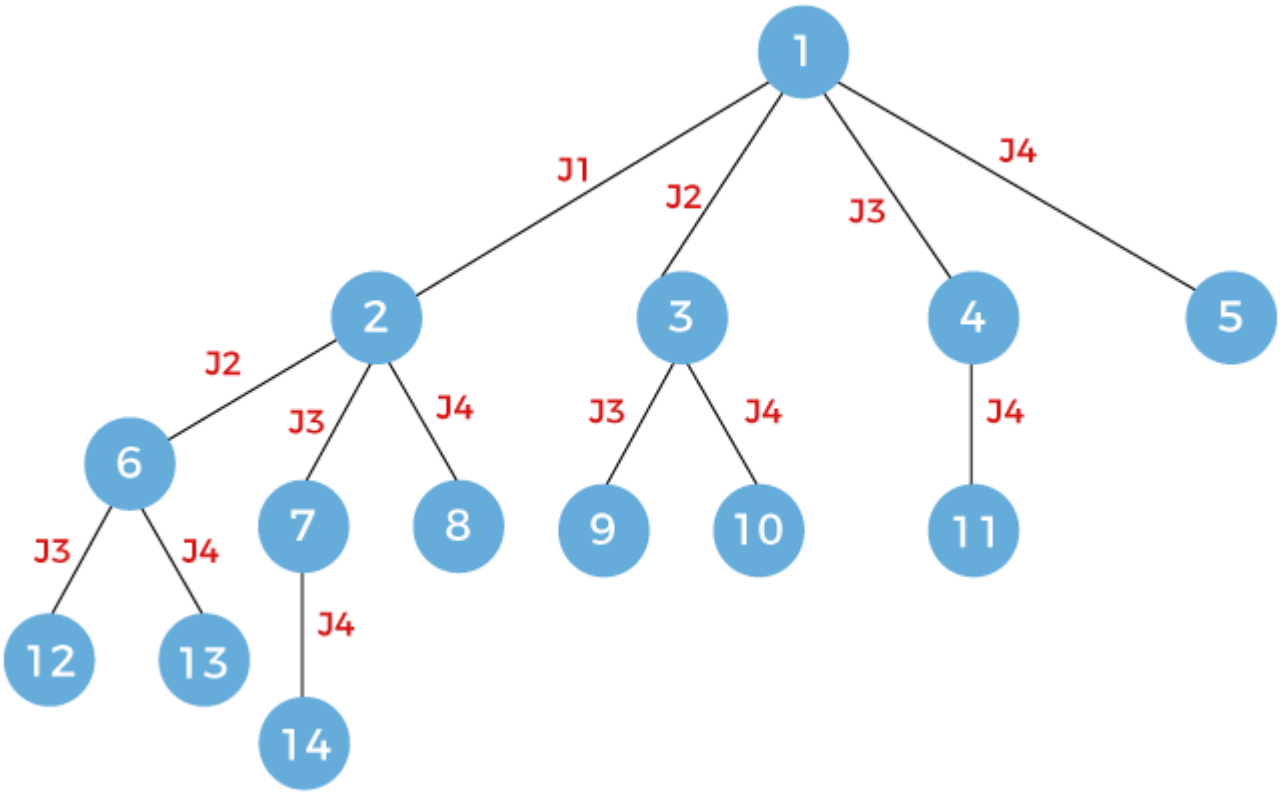
Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.



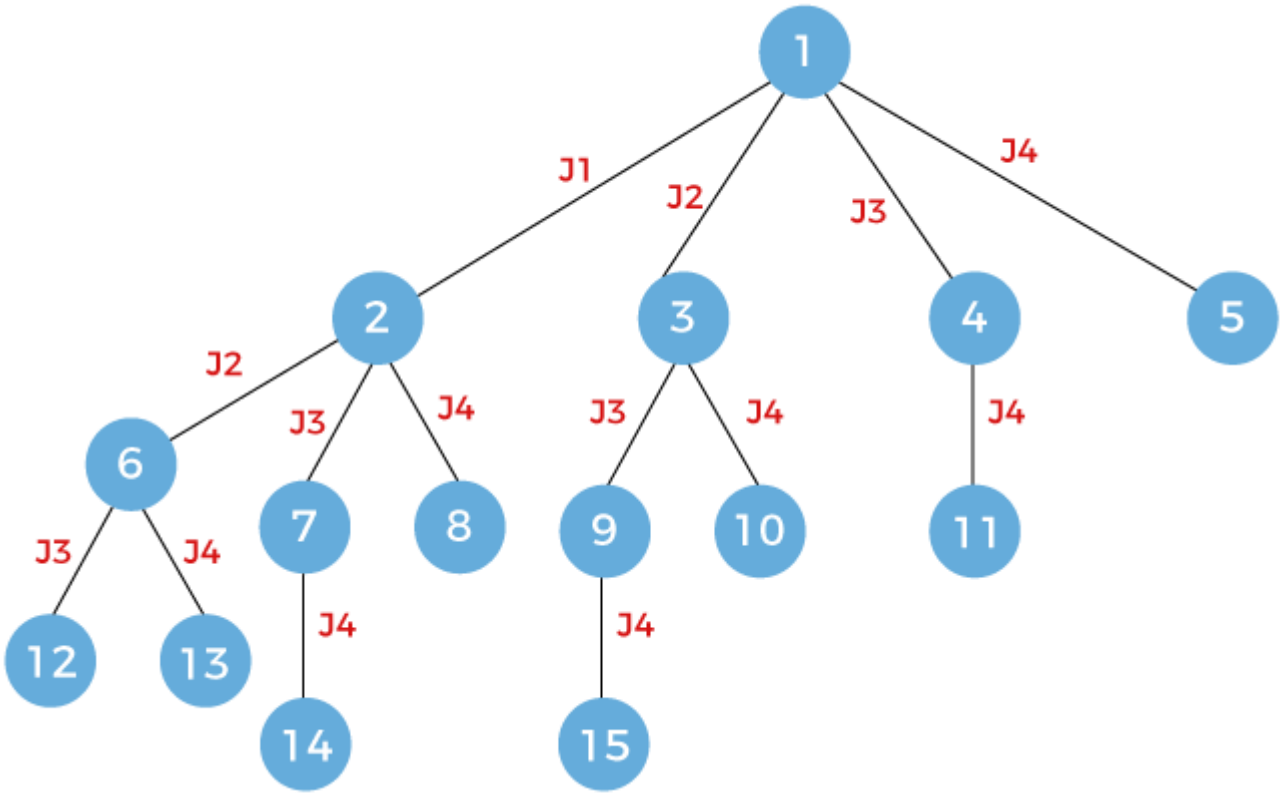
Now we will expand node 6, and here we will consider the jobs j3 and j4.



Now we will expand node 7 and here we will consider job j4.



Now we will expand node 9, and here we will consider job j4.



The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.

The above is the state space tree for the solution s1 = {j1, j4}

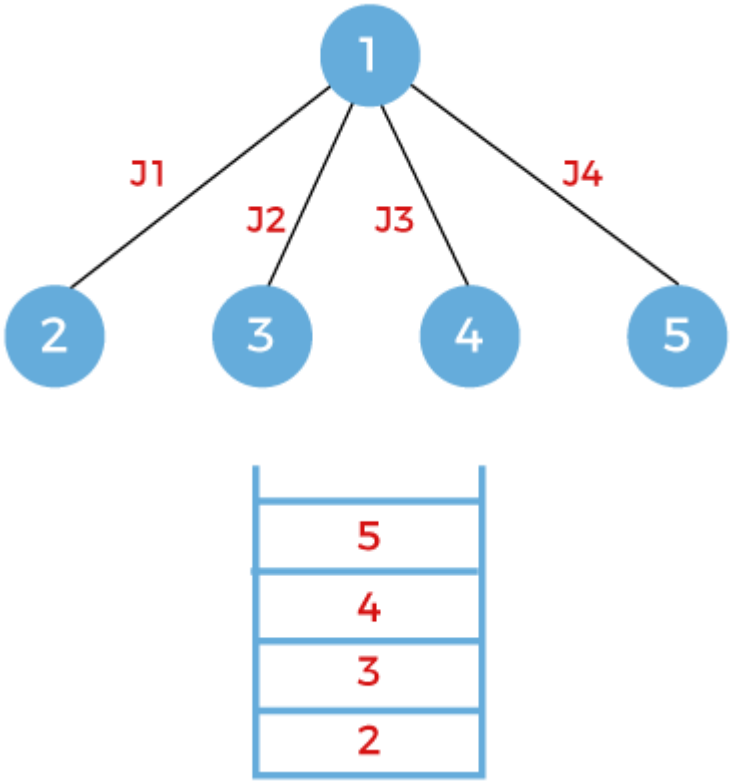
Second method:

We will see another way to solve the problem to achieve the solution s1.

First, we consider the node 1 shown as below:

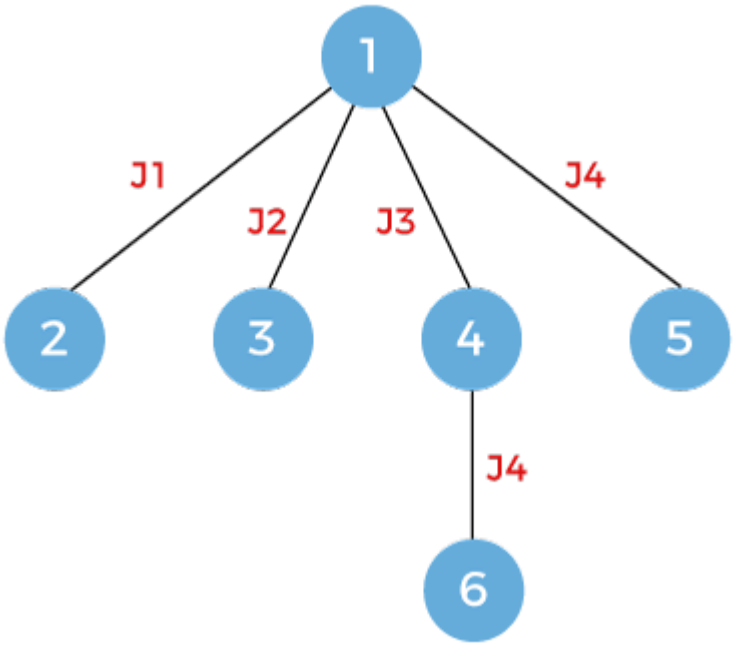
Now, we will expand the node 1. After expansion, the state space tree would be appeared as:

On each expansion, the node will be pushed into the stack shown as below:

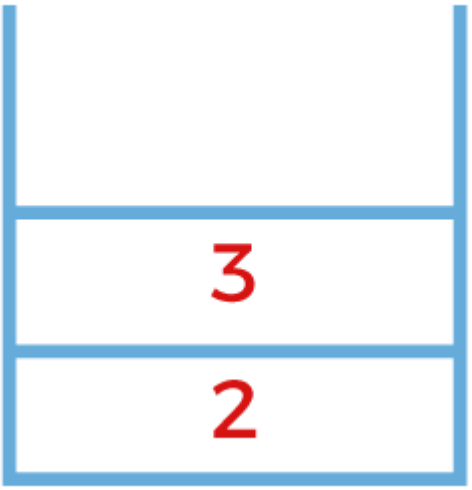


Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.

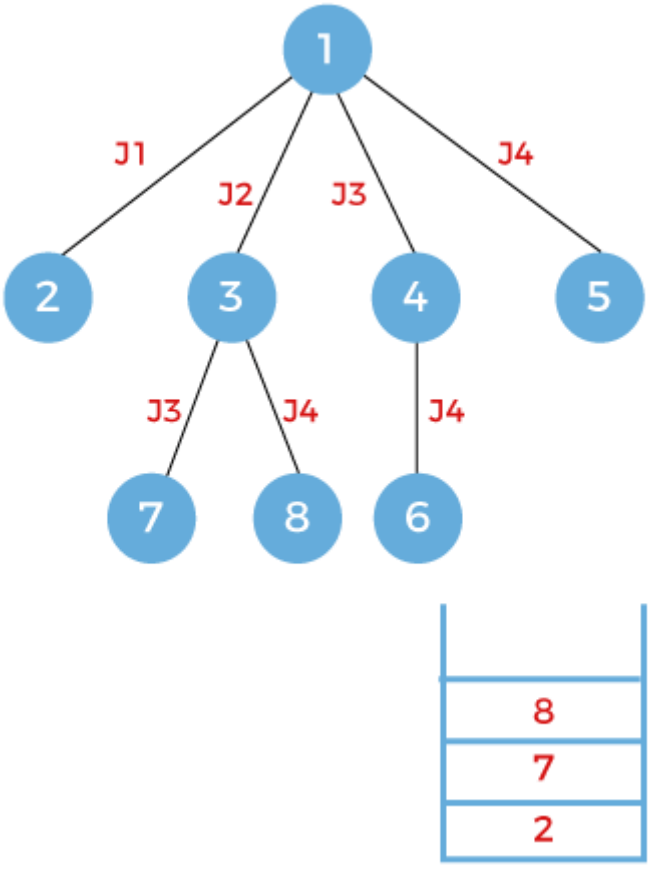
The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:



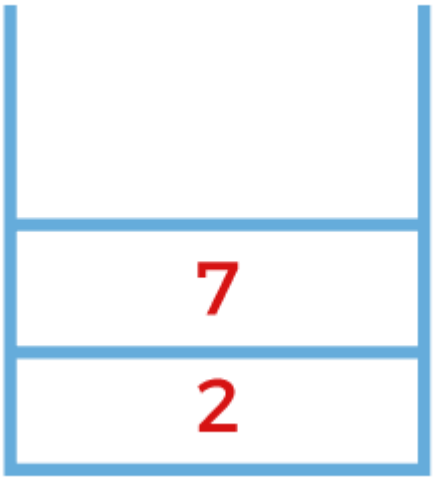
The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j4 so there is no further scope of expansion.



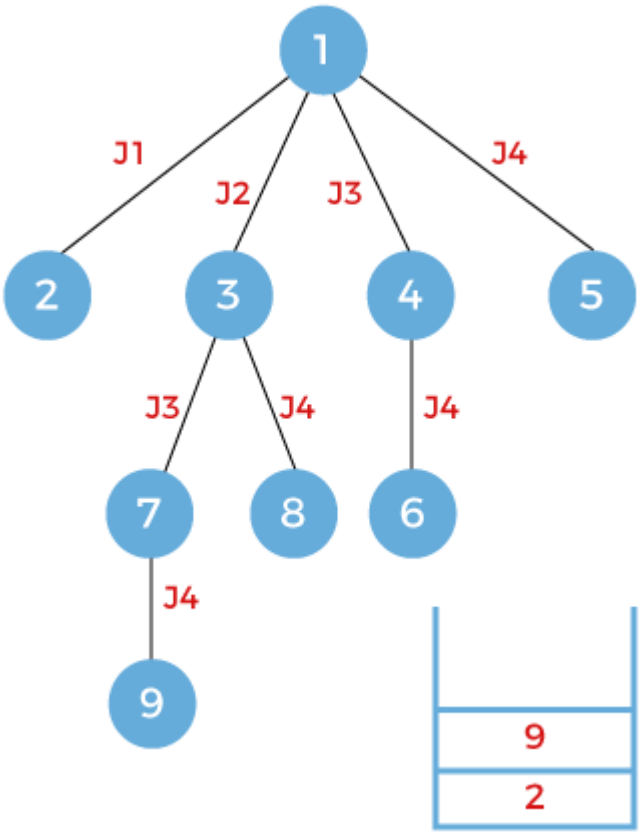
The next node to be expanded is node 3. Since the node 3 works on the job j2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:



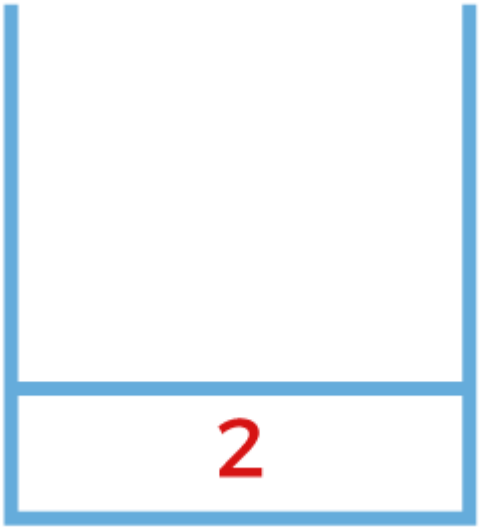
The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j4 so there is no further scope for the expansion.



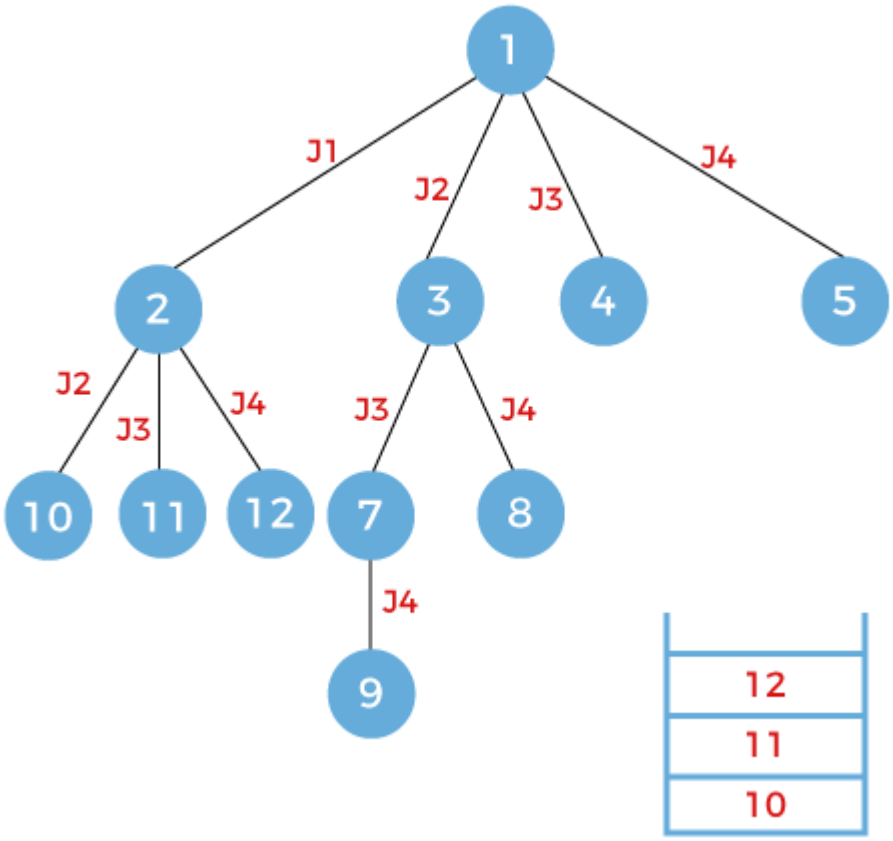
The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j3 so node 7 will be further expanded to node 9 that works on the job j4 as shown as below and the node 9 will be pushed into the stack.



The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j2, j3, and j4 respectively. There newly nodes will be pushed into the stack shown as below:



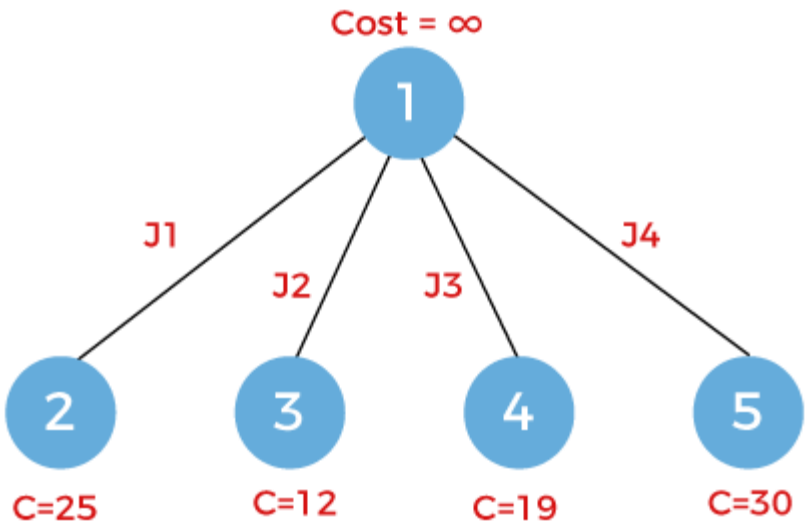
In the above method, we explored all the nodes using the stack that follows the LIFO principle.

Third method

There is one more method that can be used to find the solution and that method is Least cost branch and bound. In this technique, nodes are explored based on the cost of the node. The cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.

Let's first consider the node 1 having cost infinity shown as below:

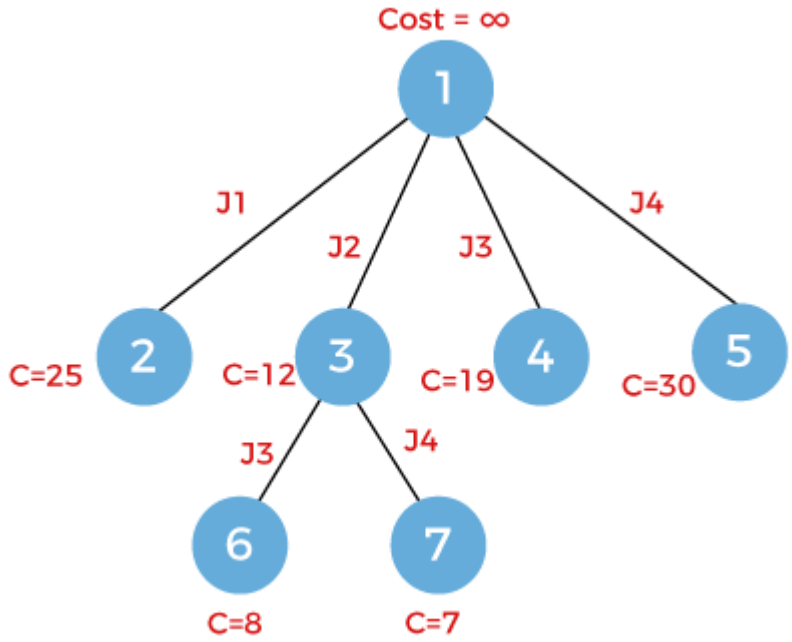
Now we will expand the node 1. The node 1 will be expanded into four nodes named as 2, 3, 4 and 5 shown as below:



Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.

Since it is the least cost branch n bound, so we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 3. So, we will explore the node 3 having cost 12.

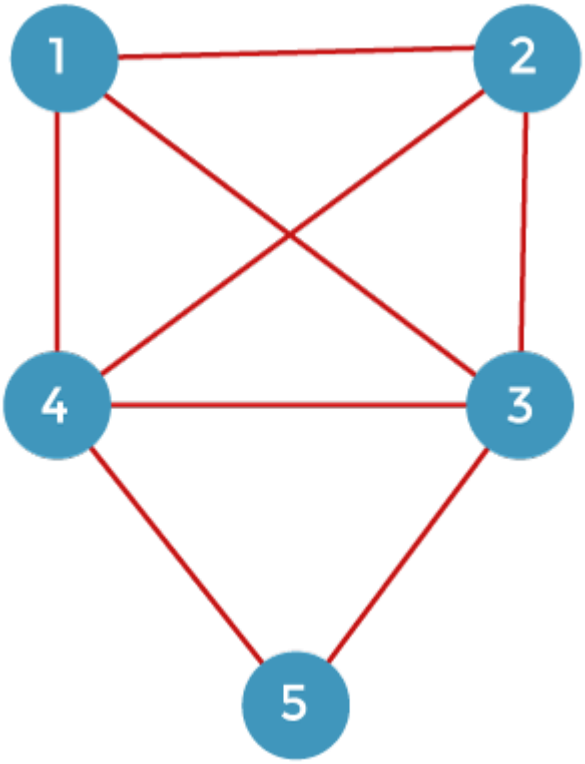
Since the node 3 works on the job j2 so it will be expanded into two nodes named as 6 and 7 shown as below:



The node 6 works on job j3 while the node 7 works on job j4. The cost of the node 6 is 8 and the cost of the node 7 is 7. Now we have to select the node which is having the minimum cost. The node 7 has the minimum cost so we will explore the node 7. Since the node 7 already works on the job j4 so there is no further scope for the expansion.

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. Consider the below graph:



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.

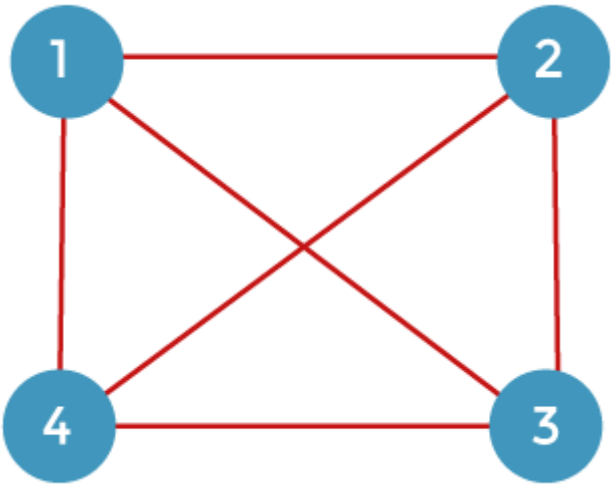
The adjacent matrix of the problem is given below:

	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	∞

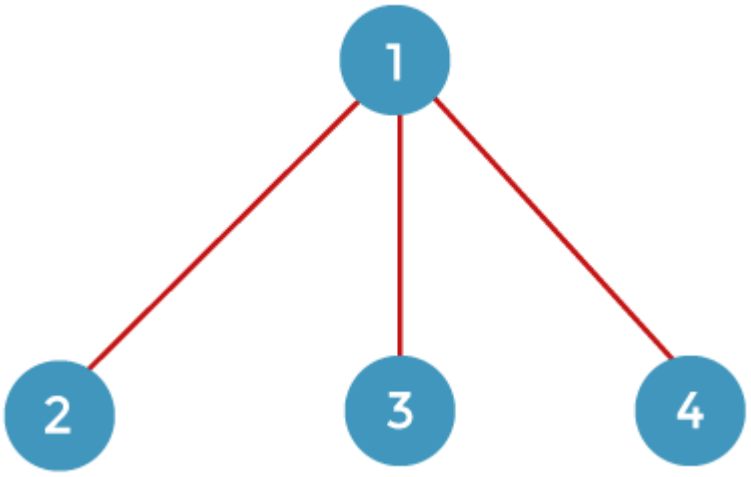
Now we look at how this problem can be solved using the branch n bound.

Let's first understand the approach then we solve the above problem.

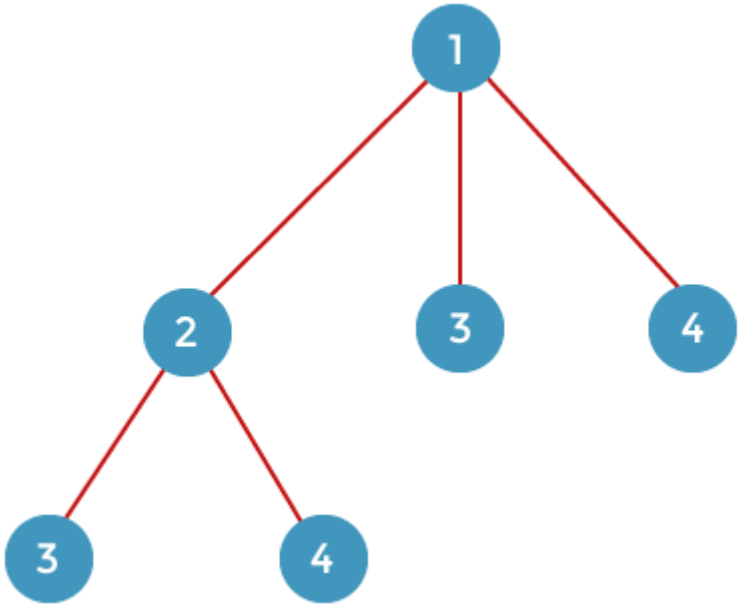
The graph is given below, which has four vertices:



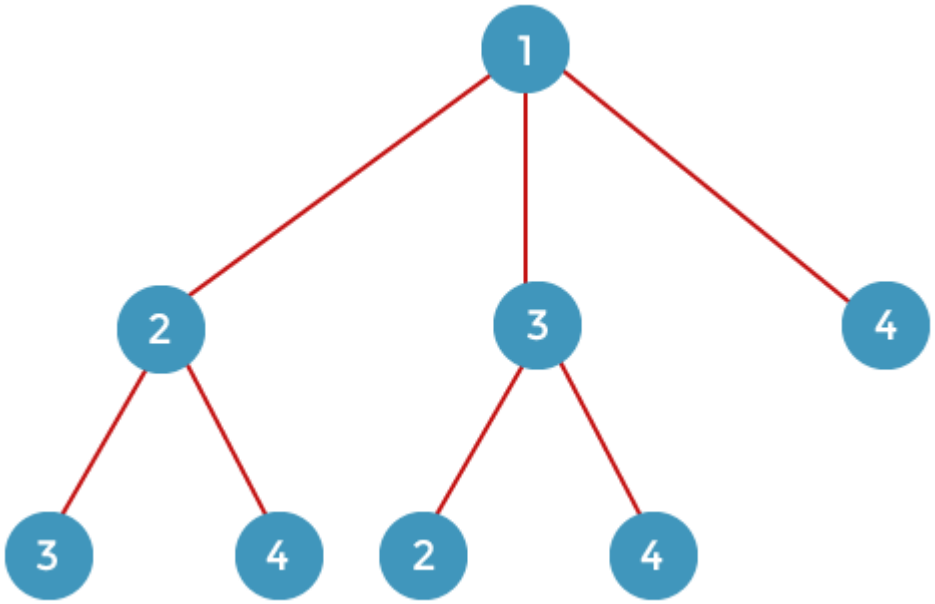
Suppose we start travelling from vertex 1 and return back to vertex 1. There are various ways to travel through all the vertices and returns to vertex 1. We require some tools that can be used to minimize the overall cost. To solve this problem, we make a state space tree. From the starting vertex 1,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



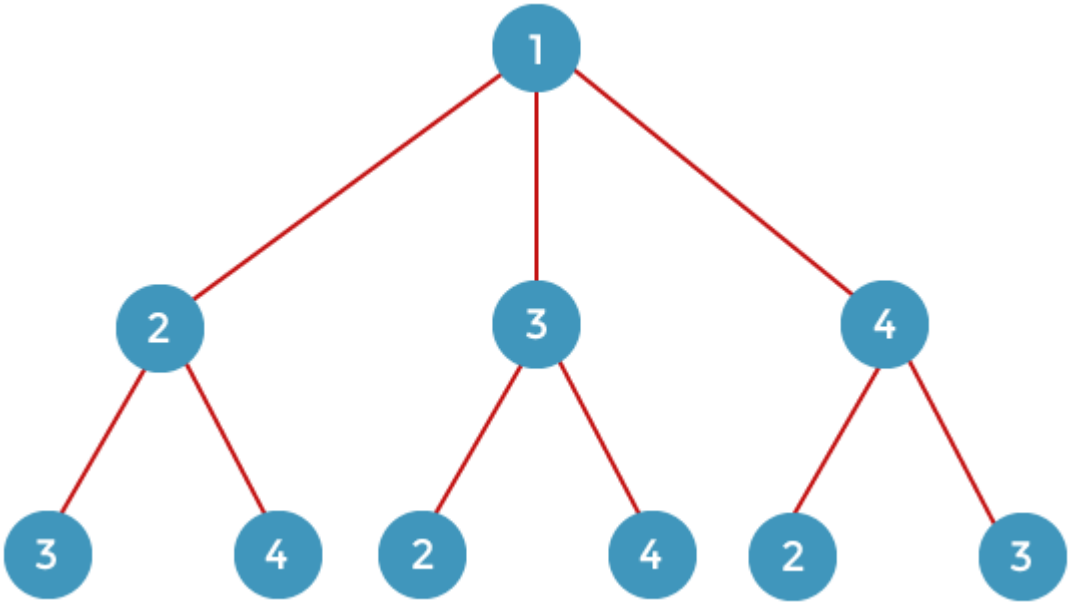
From vertex 2, we can go either to vertex 3 or 4. If we consider vertex 3, we move to the remaining vertex, i.e., 4. If we consider the vertex 4 shown in the below diagram:



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:

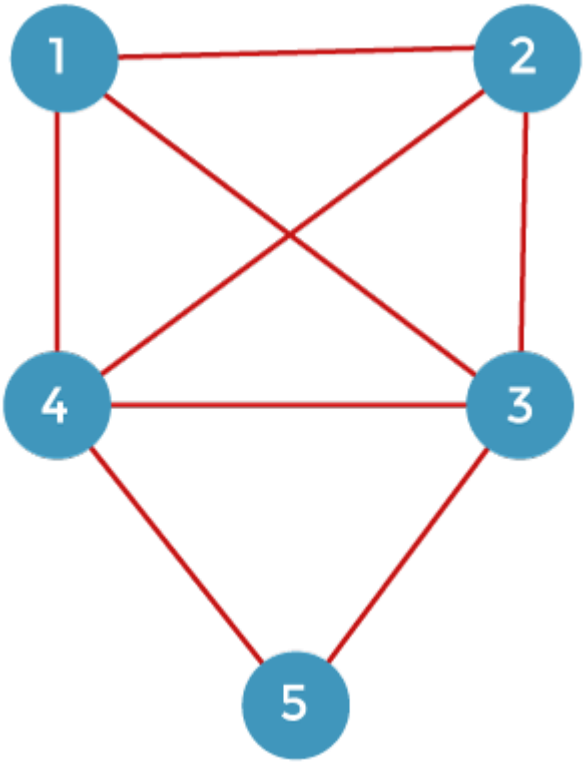


From vertex 4, we can go to the remaining vertices, i.e., 2 or 3. If we consider vertex 2, then we move to the remaining vertex, i.e., 3, and if we consider the vertex 3, then we move to the remaining vertex, i.e., 2 shown in the below diagram:



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. In order to solve the problem using branch n bound, we use a level order. First, we will observe in which order, the nodes are generated. While creating the node, we will calculate the cost of the node simultaneously. If we find the cost of any node greater than the upper bound, we will remove that node. So, in this case, we will generate only useful nodes but not all the nodes.

Let's consider the above problem.



	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	∞

As we can observe in the above adjacent matrix that 10 is the minimum value in the first row, 2 is the minimum value in the second row, 2 is the minimum value in the third row, 3 is the minimum value in the third row, 3 is the minimum value in the fourth row, and 4 is the minimum value in the fifth row.

Now, we will reduce the matrix. We will subtract the minimum value with all the elements of a row. First, we evaluate the first row. Let's assume two variables, i.e., i and j, where 'i' represents the rows, and 'j' represents the columns.

When i = 0, j =0

M[0][0] = ∞-10= ∞

When i = 0, j = 1

M[0][1] = 20 - 10 = 10

When i = 0, j = 2

M[0][2] = 30 - 10 = 20

When i = 0, j = 3

M[0][3] = 10 - 10 = 0

When i = 0, j = 4

M[0][4] = 11 - 10 = 1

The matrix is shown below after the evaluation of the first row:

	0	1	2	3	4
0	∞	10	20	0	1
1	13	∞	14	2	0
2	3	5	∞	2	4
3	19	6	18	∞	3
4	16	4	7	16	∞

Consider the second row.

When i = 1, j =0

M[1][0] = 15-2= 13

When i = 1, j = 1

M[1][1] = ∞ - 2= ∞

When i = 1, j = 2

M[1][2] = 16 - 2 = 14

When i = 1, j = 3

M[1][3] = 4 - 2 = 2

When i = 1, j = 4

M[1][4] = 2 - 2 = 0

The matrix is shown below after the evaluation of the second row:

	0	1	2	3	4
0	∞	10	20	0	1
1	13	∞	14	2	0
2	1	3	∞	0	2
3	19	6	18	∞	3
4	16	4	7	16	∞

Consider the third row:

When i = 2, j =0

M[2][0] = 3-2= 1

When i = 2, j = 1

M[2][1] = 5 - 2= 3

When i = 2, j = 2

M[2][2] = ∞ - 2 = ∞

When i = 2, j = 3

M[2][3] = 2 - 2 = 0

When i = 2, j = 4

M[2][4] = 4 - 2 = 2

The matrix is shown below after the evaluation of the third row:

Consider the fourth row:

When i = 3, j =0

M[3][0] = 19-3= 16

When i = 3, j = 1

M[3][1] = 6 - 3= 3

When i = 3, j = 2

M[3][2] = 18 - 3 = 15

When i = 3, j = 3

M[3][3] = ∞ - 3 = ∞

When i = 3, j = 4

M[3][4] = 3 - 3 = 0

The matrix is shown below after the evaluation of the fourth row:

	0	1	2	3	4
0	∞	10	20	0	1
1	13	∞	14	2	0
2	1	3	∞	0	2
3	16	3	15	∞	0
4	16	4	7	16	∞

Consider the fifth row:

When i = 4, j =0

M[4][0] = 16-4= 12

When i = 4, j = 1

M[4][1] = 4 - 4= 0

When i = 4, j = 2

M[4][2] = 7 - 4 = 3

When i = 4, j = 3

M[4][3] = 16 - 4 = 12

When i = 4, j = 4

M[4][4] = ∞ - 4 = ∞

The matrix is shown below after the evaluation of the fifth row:

	0	1	2	3	4
0	∞	10	20	0	1
1	13	∞	14	2	0
2	1	3	∞	0	2
3	16	3	15	∞	0
4	12	0	3	12	∞

The above matrix is the reduced matrix with respect to the rows.

Now we reduce the matrix with respect to the columns. Before reducing the matrix, we first find the minimum value of all the columns. The minimum value of first column is 1, the minimum value of the second column is 0, the minimum value of the third column is 3, the minimum value of the fourth column is 0, and the minimum value of the fifth column is 0, as shown in the below matrix:

Now we reduce the matrix.

Consider the first column.

When i = 0, j =0

M[0][0] = ∞-1= ∞

When i = 1, j = 0

M[1][0] = 13 - 1= 12

When i = 2, j = 0

M[2][0] = 1 - 1 = 0

When i = 3, j = 0

M[3][0] = 16 - 1 = 15

When i = 4, j = 0

M[4][0] = 12 - 1 = 11

The matrix is shown below after the evaluation of the first column:

	0	1	2	3	4
0	∞	10	20	0	1
1	12	∞	14	2	0
2	0	3	∞	0	2
3	15	3	15	∞	0
4	11	0	3	12	∞

Since the minimum value of the first and the third columns is non-zero, we will evaluate only first and third columns. We have evaluated the first column. Now we will evaluate the third column.

Consider the third column.

When i = 0, j =2

M[0][2] = 20-3= 17

When i = 1, j = 2

M[1][2] = 13 - 1= 12

When i = 2, j = 2

M[2][2] = 1 - 1 = 0

When i = 3, j = 2

M[3][2] = 16 - 1 = 15

When i = 4, j = 2

M[4][2] = 12 - 1 = 11

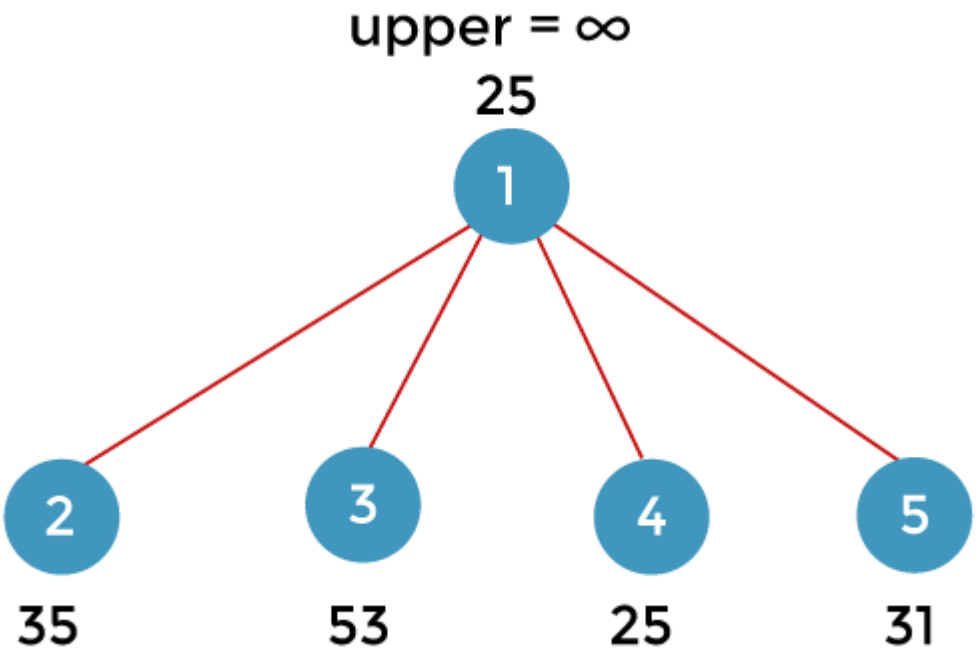
The matrix is shown below after the evaluation of the third column:

	0	1	2	3	4
0	∞	10	17	0	1
1	12	∞	12	2	0
2	0	3	0	0	2
3	15	3	15	∞	0
4	11	0	0	12	∞

The above is the reduced matrix. The minimum value of rows is 21, and the columns is 4. Therefore, the total minimum value is (21 + 4) equals to 25.

Let's understand that how to solve this problem using branch and bound with the help of a state-space tree.

To make a state-space tree, first, we consider node 1. From node 1, we can go either to nodes 2, 3, 4, or 5 as shown in the below image. The cost of node 1 would be the cost which we achieved in the above-reduced matrix, i.e.., 25. Here, we will also maintain the upper bound. Initially, the upper bound would-be infinity.



Now, consider node 2. It means that we are moving from node 1 to node 2. Make the first row and second column as infinity shown in the below matrix:

Once we move from node 1 to node 2, we cannot move back to node 1. Therefore, we have to make 2 to 1 as infinity shown in the below matrix:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	12	2	0
3	0	∞	0	0	2
4	15	∞	15	∞	0
5	11	∞	0	12	∞

Since each row and column contains atleast one zero value; therefore, we can say that above matrix has been reduced. The cost of reduction of node 2 is c(1, 2) + r + r` = 10 + 25 + 0 = 35.

Now we will find the minimum value of each column of the new reduced matrix. The minimum value of the first column is 11 and the minimum value of other three columns is 0.

Now, consider the node 3. It means that we are moving from the node 1 to node 3. Make the first row and third column as infinity shown in the below matrix:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	∞	2	0
3	0	3	∞	0	2
4	15	3	∞	∞	0
5	11	0	∞	12	∞

Once we move from the node 1 to node 3, we cannot move back to the node 1. Therefore, we have to make 3 to 1 as infinity shown in the below matrix:

Since each row and column contains atleast one zero value; therefore, we can say that above matrix has been reduced. The cost of reduction of node 3 is c(1, 3) + r + r` = 17 + 25 + 11= 53.

Now, consider the node 4. It means that we are moving from the node 1 to node 4. Make the first row and forth column as infinity shown in the below matrix:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	12	∞	0
3	0	∞	0	∞	2
4	∞	∞	∞	∞	∞
5	11	∞	0	∞	∞

Once we move from the node 1 to node 4, we cannot move back to the node 1. Therefore, we have to make 4 to 1 as infinity shown in the below matrix:

Since each row and column contains atleast one zero value; therefore, we can say that above matrix has been reduced. The cost of reduction of node 4 is c(1, 4) + r + r` = 0 + 25 + 0 = 25.

Now, consider the node 5. It means that we are moving from the node 1 to node 5. Make the first row and fifth column as infinity shown in the below matrix:

Once we move from the node 1 to node 5, we cannot move back to the node 1. Therefore, we have to make 5 to 1 as infinity shown in the below matrix:

Since each row and column contains atleast one zero value; therefore, we can say that above matrix has been reduced. In this case, second and third rows are non-zero. Therefore, we have to first find the minimum values of both the rows. The minimum value of second row is 2; therefore, we subtract 2 from all the elements of the second row. The elements of second row would be:

A[1][0] = 12-2 = 10

A[1][1] = ∞

A[1][2] = 11 - 2 = 9

A[1][3] = 2 - 2 = 0

A[1][4] = ∞ - 2 = ∞

As we can observe now that the second row contains one zero value.

The cost of reduction of node 5 is c(1, 5) + r + r` = 1 + 25 + 5 = 31

Since the node 4 has the minimum cost, i.e., 25. So we will explore the node 4 first. From the vertex 4, we can go either to the vertex 2, 3 or 5 as shown in the below image:

Now we have to calculate the cost of the path from the vertex 4 to 2, vertex 4 to 3, and vertex 4 to 5. Here, we will use the matrix of node 4 to find the cost of all the nodes.

First, we consider the path from the vertex 4 to the vertex 2. We make fourth row as ∞ and second column as ∞. Since we cannot move back from 2 to 1, so 1 to 2 is also infinity as shown in the below matrix:

Since all the rows and columns have atleast one zero value. Therefore, we can say that this matrix is already reduced. So, there would be no reduction cost. The cost of reduction of node 2 is c(4, 2) + r + r` = 3 + 25 + 0 = 28

Now we have to calculate the cost of the path from the vertex 4 to the vertex 3. We make fourth row and third column as infinity as shown in the below matrix. Since we cannot move from the vertex 3 to 1, so we make 3 to 1 as infinity shown in the below matrix:

Now we will check whether each row and column contain atleast one zero value or not. First, we observe all the rows. Since the third row does not have a zero value, so we first find the minimum value of the third row. The minimum value of the third row is 2, so we subtract 2 from all the elements of the third row. The elements of third row would be:

A[2][0] = ∞ - 2 = ∞

A[2][1] = 3 - 2 = 1

A[2][2] = ∞ - 2 = ∞

A[2][3] = ∞ - 2 = ∞

A[2][4] = 2 - 2 = 0

As we can observe now that the third row contains one zero value.

The first column does not contain the zero value. The minimum value of the first column is 11. We subtract 11 from all the elements of the first column. The elements of first column would be:

A[0][0] = ∞ - 11 = ∞

A[1][0] = 12 - 11 = 1

A[2][0] = ∞ - 11= ∞

A[3][0] = ∞ - 11= ∞

A[4][0] = 11 - 11 = 0

As we can observe now that the first column contains one zero value. The total minimum cost is 11 +2 equals to 13. The cost of reduction of node 3 is c(4, 3) + r + r` = 12 + 25 + 13 = 50.

Now we will calculate the cost of the path from the vertex 4 to 5. We make fourth row and fifth column as infinity. Since we cannot move back from the node 5 to 1, so we also make 1 to 5 as infinity shown in the below matrix:

Now we will check whether each row and column contain atleast one zero value or not. First, we observe all the rows. The second row does not contain the zero value, so we find the minimum value of the second row. The minimum value is 11 so we subtract 11 from all the elements of the second row. The elements of second row would be:

A[1][0] = 12 - 11 = 1

A[1][1] = ∞ - 11 = ∞

A[1][2] = 11 - 11 = 0

A[1][3] = ∞ - 11 = ∞

A[1][4] = ∞ - 11 = ∞

As we can observe now that the second row contains one zero value. The cost of reduction of node 5 is c(4, 5) + r + r` = 0 + 25 + 11 = 36.

Now we will compare the cost of all the leaf nodes. The node with a cost 28 is minimum so we will explore this node. The node with a cost 28 can be further expanded to the nodes 3 and 5 as shown in the below figure:

Now we have to calculate the cost of both the nodes, i.e., 3 and 5. First we consider the path from node 2 to node 3. Consider the matrix of node 2 which is shown below:

We make second row and third column as infinity. Also, we cannot move back from the node 3 to node 1 so we make 3 to 1 as infinity as shown in the below matrix:

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	∞	∞	∞
3	∞	∞	∞	0	2
4	15	∞	∞	∞	∞
5	11	∞	∞	∞	∞

Now we will check whether any row contains zero value or not. Since third row does not contain any zero value so we will find the minimum value of the third row. The minimum value of the third row is 2 so we subtract 2 from all the elements of the third row. The elements of third row would be:

A[2][0] = ∞ - 2 = ∞

A[2][1] = ∞ - 2 = ∞

A[2][2] = ∞ - 2 = ∞

A[2][3] = ∞ - 2 = ∞

A[2][4] = 2 - 2 = 0

Since fifth row does not contain any zero value so we will find the minimum value of the fifth row. The minimum value of the fifth row is 11 so we subtract 11 from all the elements of the fifth row.

A[4][0] = 11 - 11 = 0

A[4][1] = ∞ - 11 = ∞

A[4][2] = ∞ - 11 = ∞

A[4][3] = ∞ - 11 = ∞

A[4][4] = ∞ - 11 = ∞

The total minimum cost is (11 + 2) equals to 13. The cost of reduction of node 3 is c(2, 3) + r + r` = 11 + 28 + 13 = 52.

Consider the path from node 2 to node 5. Make the fourth row and third column as infinity. Since we cannot move back from the node 5 to node 1 so make 1 to 5 also as infinity as shown in the below matrix:

Now we will check whether any row contains zero value or not. Since every row and column contains a zero value; therefore, the above matrix is the reduced matrix.

The cost of reduction of node 5 is c(2, 5) + r + r` = 0 + 28 + 0 = 28

Now we will find out the leaf node with a minimum cost. The node 5 with a cost 28 is minimum so we select node 5 for the further exploration. The node 5 can be further expanded to the node 3 as shown in the below figure:

Here, we will use the matrix of node 5 having cost 28 as shown below:

Consider the path from node 5 to node 3. Make the fifth row and third column as infinity. Since we cannot move back from the node 3 to node 1 so make 1 to 5 also as infinity as shown in the below matrix:

Now we will check whether any row contains zero value or not. Since every row and column contains a zero value; therefore, the above matrix is the reduced matrix.

The cost of reduction of node 3 is c(5, 3) + r + r` = 0 + 28 + 0 = 28.

Finally, we traverse all the nodes. The upper value is updated from infinity to 28. We will check whether any leaf node has a value less than 28. Since no leaf node contains the value less than 28 so we discard all the leaf nodes from the tree as shown below:

The path of the tour would be 1->4->2->5->3.

P-class Problems

- P problems are a set of problems that can be solved in polynomial time by deterministic algorithms.
- P is also known as PTIME or DTIME complexity class.
- P problems are a set of all decision problems which can be solved in polynomial time using the deterministic Turing machine.
- They are simple to solve, easy to verify and take computationally acceptable time for solving any instance of the problem. Such problems are also known as *"tractable"*.
- In the worst case, searching an element from the list of size n takes n comparisons. The number of comparisons increases linearly with respect to the input size. So linear search is P problem.
- In practice, most of the problems are P problems. Searching an element in the array (O(n)), inserting an element at the end of a linked list (O(n)), sorting data using selection sort($O(n^2)$), finding the height of the tree ($O(\log_2 n)$), sort data using merge sort($O(n\log_2 n)$), matrix multiplication $O(n^3)$ are few of the examples of P problems.
- An algorithm with $O(2^n)$ complexity takes double the time if it is tested on a problem of size (n + 1). Such problems do not belong to class P.
- It excludes all the problems which cannot be solved in polynomial time. The knapsack problem using the brute force approach cannot be solved in polynomial time. Hence, it is not a P problem.
- There exist many important problems whose solution is not found in polynomial time so far, nor it has been proved that such a

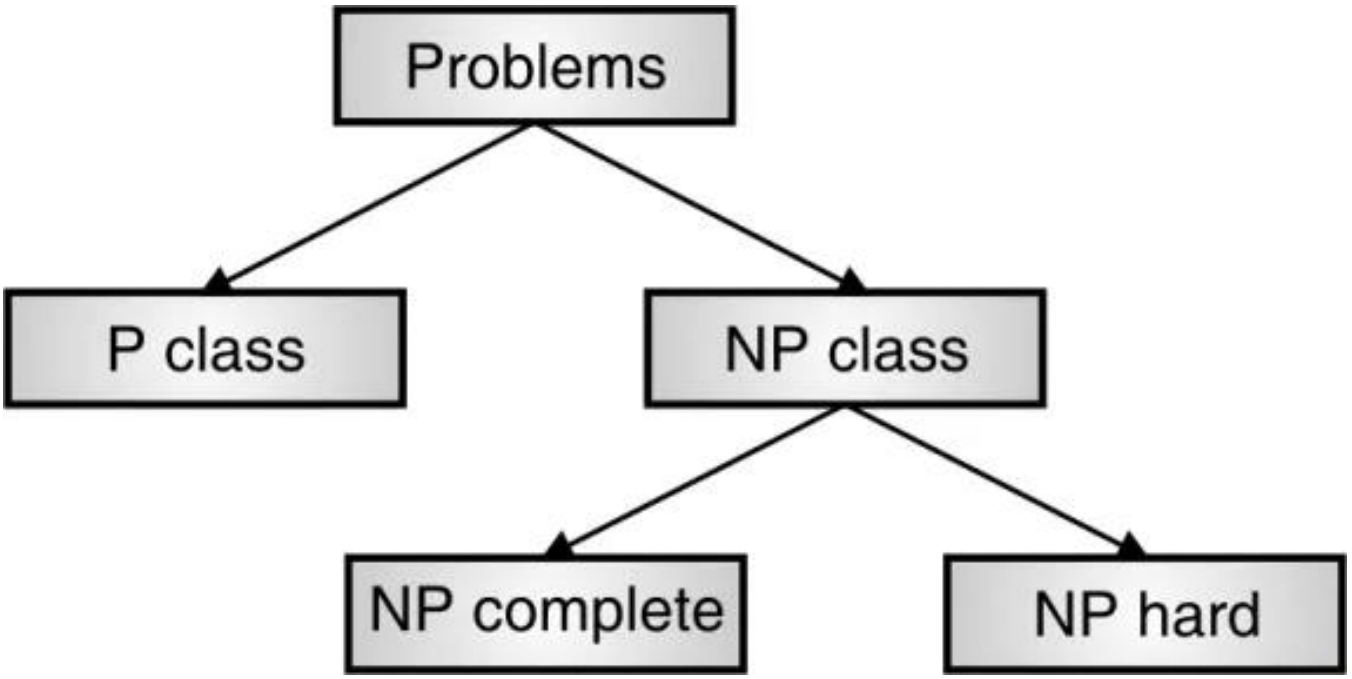
solution does not exist. TSP, Graph colouring, partition problem, knapsack etc. are examples of such classes.

Examples of P Problems:

- Insertion sort
- Merge sort
- Linear search
- Matrix multiplication
- Finding minimum and maximum elements from the array

NP-Class of Problems

- NP is a set of problems which can be solved in nondeterministic polynomial time. NP does not mean non-polynomial, it stands for Non-Deterministic Polynomial-time.
- The non-deterministic algorithm operates in two stages.
- **Nondeterministic (guessing) stage:** For input instance I, some solution string S is generated, which can be thought of as a candidate solution.
- **Deterministic (verification) stage:** I and S are given as input to the deterministic algorithm, which returns "Yes" if S is a solution for input instance I.
- The solution to NP problems cannot be obtained in polynomial time, but given the solution, it can be verified in polynomial time.
- NP includes all problems of P, i.e. $P \subseteq NP$.
- Knapsack problem ($O(2^n)$), Travelling salesman problem ($O(n!)$), Tower of Hanoi ($O(2^n - 1)$), Hamiltonian cycle ($O(n!)$) are examples of NP problems.
- NP Problems are further classified into NP-complete and NP-hard categories.
- The following shows the taxonomy of complexity classes.



- The NP-hard problems are the hardest problem. NP-complete problems are NP-hard, but the converse is not true.
- If NP-hard problems can be solved in polynomial time, then so is NP-complete.

Examples of NP problems

- Knapsack problem ($O(2^n)$)
- Travelling salesman problem ($O(n!)$)
- Tower of Hanoi ($O(2^n - 1)$)
- Hamiltonian cycle ($O(n!)$)

P and NP Problems – Differentiate

Sr. No.	P Problems	NP Problems
1.	P problems are set of problems which can be solved in polynomial time by deterministic algorithms.	NP problems are problems which can be solved in nondeterministic polynomial time.
2.	P Problems can be solved and verified in polynomial time	The solution to NP problems cannot be obtained in polynomial time, but if the solution is given, it can be verified in polynomial time.
3.	P problems are a subset of NP problems	NP Problems are a superset of P problems

Sr. No.	P Problems	NP Problems
4.	All P problems are deterministic in nature	All the NP problems are non-deterministic in nature
5.	**Example:** Selection sort, Linear search	**Example:** TSP, Knapsack problem

Tractable Problem: A problem that is solvable by a polynomial-time algorithm.
The upper bound is polynomial.
Here are **examples** of tractable problems (ones with known polynomial-time algorithms):
– Searching an unordered list
– Searching an ordered list
– Sorting a list
– Multiplication of integers (even though there's a gap)
– Finding a minimum spanning tree in a graph (even though there's a gap)

j = 4 near
(4) = 0

J = 5
Is near (5) 0
4 0 and cost (4, 5) = 4

select the min cost from the above obtained costs, which is 3 and corresponding J = 3

min cost = 1 + cost(3, 4)
= 1 + 3 = 4

T (2, 1) = 3
T (2, 2) = 4

Near [j] = 0
i.e. near (3) =0

<u>for (k = 1 to n)</u>

K = 1
is near (1) 0, yes
2 0
and cost (1,2) > cost(1, 3)
4 > 9, No

K = 2
Is near (2) 0, No

K = 3
Is near (3) 0, No

K = 4
Is near (4) 0, No

K = 5

2	0	0	0	4

1 2 3 4 5

2	0	0	0	3

T (2, 1) = 3
T (2, 2) = 4

Is near (5) 0 4 0, yes and is cost (5, 4) > cost (5, 3) 4 > 3, yes than near (5) = 3 **i =** **3** for (j = 1 to 5) J = 1 is near (1) 0 2 0 cost (1, 2) = 4 J = 2 Is near (2) 0, No		1 2 3 4 5	

J = 3
Is near (3) 0, no Near
(3) = 0

J = 4
Is near (4) 0, no Near
(4) = 0

J = 5
Is near (5) 0
Near (5) = 3 3 0, yes And
cost (5, 3) = 3

Choosing the min cost from the above obtaining costs
which is 3 and corresponding J = 5

Min cost = 4 + cost (5, 3)
= 4 + 3 = 7

T (3, 1) = 5
T (3, 2) = 3

Near (J) = 0 near (5) = 0

<u>for (k=1 to 5)</u>

k = 1
is near (1) 0, yes and
cost(1,2) > cost(1,5)
4 > , No

K = 2
Is near (2) 0 no

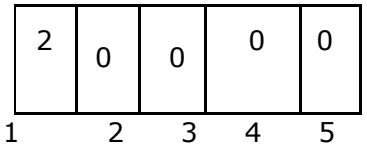
K = 3
Is near (3) 0 no

K = 4
Is near (4) 0 no

K = 5
Is near (5) 0 no

<u>i = 4</u>

<u>for J = 1 to 5</u> J
= 1



T (3, 1) = 5
T (3, 2) = 3

Is near (1) 0 2
0, yes cost (1,
2) = 4

j = 2
is near (2) 0, No

J = 3
Is near (3) 0, No Near
(3) = 0

J = 4
Is near (4) 0, No Near
(4) = 0

J = 5
Is near (5) 0, No Near
(5) = 0

Choosing min cost from the above it is only '4' and corresponding J = 1

Min cost = 7 + cost (1,2)
= 7+4 = 11

T (4, 1) = 1
T (4, 2) = 2

Near (J) = 0 Near (1) = 0 <u>for</u>

<u>(k = 1 to 5)</u>

K = 1
Is near (1) 0, No

K = 2
Is near (2) 0, No

K = 3
Is near (3) 0, No

K = 4
Is near (4) 0, No

K = 5
Is near (5) 0, No

End.

0	0	0	0	0
1	2	3	4	5

T (4, 1) = 1
T (4, 2) = 2

4.8.7. The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.

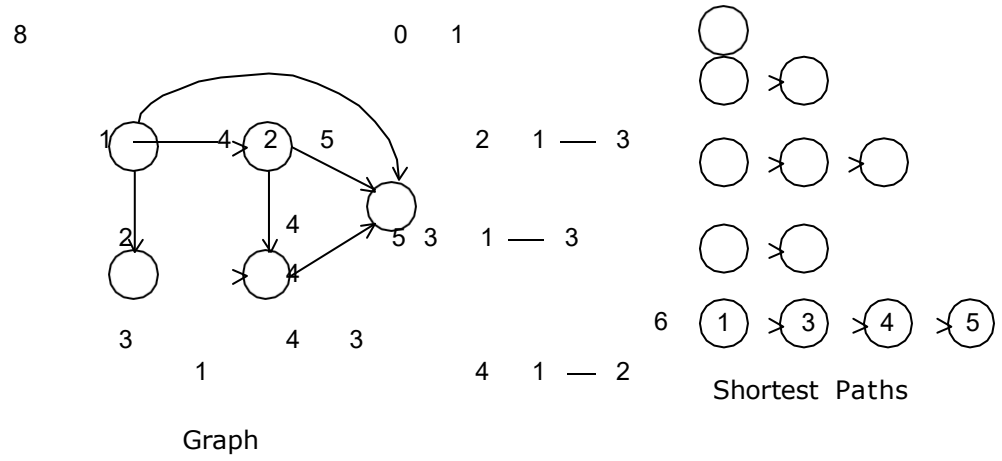
In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.

Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees. Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the

shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.

Dijkstra's algorithm does not work for negative edges at all.

The figure lists the shortest paths from vertex 1 for a five vertex weighted digraph.



Graph

Algorithm:

Algorithm Shortest-Paths (v, cost, dist, n)

```
// dist [j], 1 ≤ j ≤ n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{ for i :=1 to n do
        {
                 S [i] := false;                          // Initialize S.
                dist [i] :=cost [v, i];
        }
        S[v] := true; dist[v]  := 0.0;                    // Put v in S.
        for num := 2 to n – 1 do
        {
                Determine n - 1 paths from v.
                Choose u from among those vertices not in S such that dist[u] is minimum;
                S[u] := true;                                    // Put u is S.
                for (each w adjacent to u with S [w] = false) do
                        if (dist [w] > (dist [u] + cost [u,  w]) then      // Update distances
        dist [w] := dist [u] + cost [u, w]; }
}
```

Running time:

Depends on implementation of data structures for dist.

Build a structure with n elements A
at most m = E times decrease the value of an item mB

'n' times select the smallest value nC

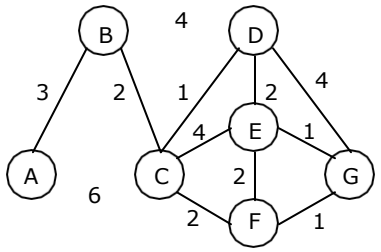
For <u>array</u> A = O (n); B = O (1); C = O (n) which gives O (n^2) total.

For heap A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n)

total.

Example 1:

Use Dijkstras algorithm to find the shortest path from A to each of the other six vertices in the graph:



Solution:

0 3 6 - - - - -

3 0 2 4 - - - -

6 2 0 1 4 2 -

The cost adjacency matrix is 4 1 0 2 - - 4

- 4 2 0 2 1

- - 2 - 2 0 1

- - - - 4 1 1 0

Here – means infinite

The problem is solved by considering the following information:

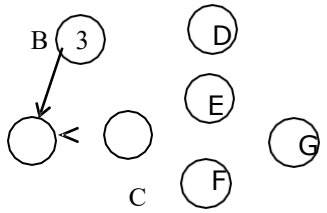
Status[v] will be either '0', meaning that the shortest path from v to v_0 has definitely been found; or '1', meaning that it hasn't.

Dist[v] will be a number, representing the length of the shortest path from v to v_0 found so far.

Next[v] will be the first vertex on the way to v_0 along the shortest path found so far from v to v_0

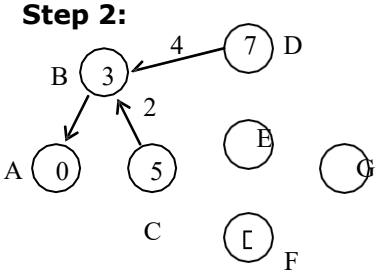
The progress of Dijkstra's algorithm on the graph shown above is as follows:

Step 1:



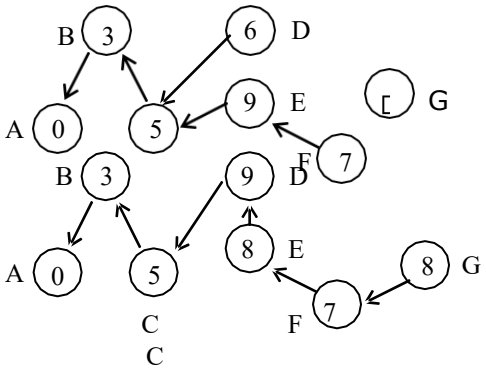
Vertex	A	B	C	D	E	F	G
Status	1	1	1	1	1	1	
Dist.	3	6	□	□	□	□	
Next	*	A	A	A	A	A	A

Step 2:

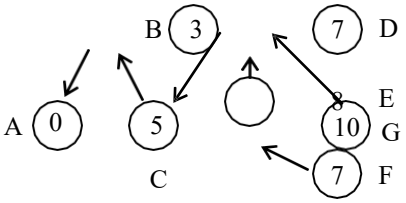


Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	□	□	□
Next	*	A	B	B	A	A	A

Step 3:

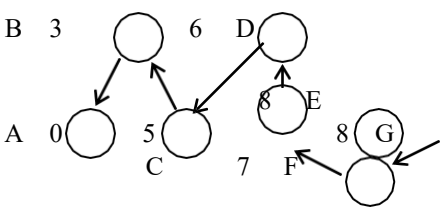


Step 4:



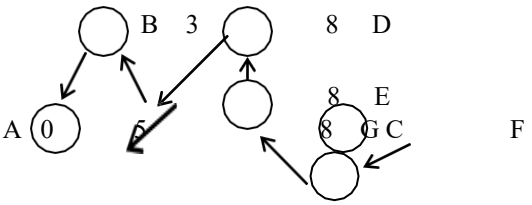
Vertex A		B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	□
Next	*	A	B	C	C	C	A

Step 5:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step 6:



Step 7:

						0	1
						7	8
						C	F

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Vertex	A
Status	0
Dist.	0
Next	*

B C D . . . . .
E F G

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F