Unit - V
Backtracking
Branch & Bound

Contents

- **Backtracking:** Backtracking Concept, N–Queens Problem, Four–Queens Problem, Eight–Queen Problem,
- Hamiltonian Cycle,
- Sum of Subsets Problem, Graph Coloring Problem.
- **Branch and Bound:** Introduction, Traveling Salesperson Problem,
- 15-Puzzle Problem,
- Comparisons between Backtracking and Branch and Bound.

Backtracking

- Backtracking is a technique based on algorithm to solve problem. It uses **recursive calling** to find the solution by building a solution **step by step increasing values with time**.
- It **removes the solutions that doesn't give rise to the solution** of the problem **based on the constraints** given to solve the problem.
- Backtracking algorithm is **applied to some specific types** of problems,

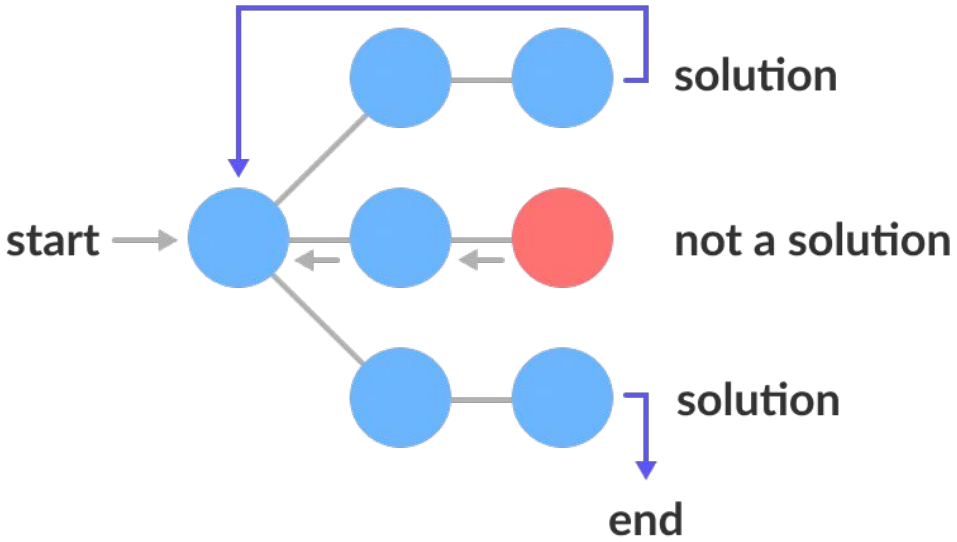
1. **Decision** problem used to **find a feasible solution** of the problem.
2. **Optimisation problem used to find the best solution** that can be applied.
3. **Enumeration** problem used to find the set of **all feasible solutions** of the problem.

Backtracking

- A backtracking algorithm is a problem-solving algorithm that uses a **brute force approach** for finding the **desired** output.
- The Brute force approach **tries out all the possible solutions and chooses the desired/best solutions.**
- The term backtracking suggests that **if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.**
- This approach is used to solve problems that have multiple solutions. If **you want an optimal solution, you must go for dynamic programming**.

State Space Tree

A space state tree is a tree **representing all the possible states (solution or no solution)** of the problem from **the root as an initial state to the leaf as a terminal state.**



Backtracking Algorithm

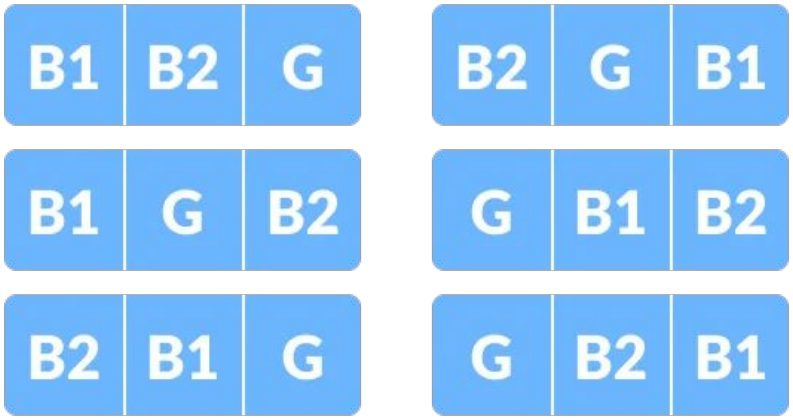
```
Backtrack(x)
    if x is not a solution
        return false
    if x is a new solution
        add to list of solutions
    backtrack(expand x)
```

Example Backtracking Approach

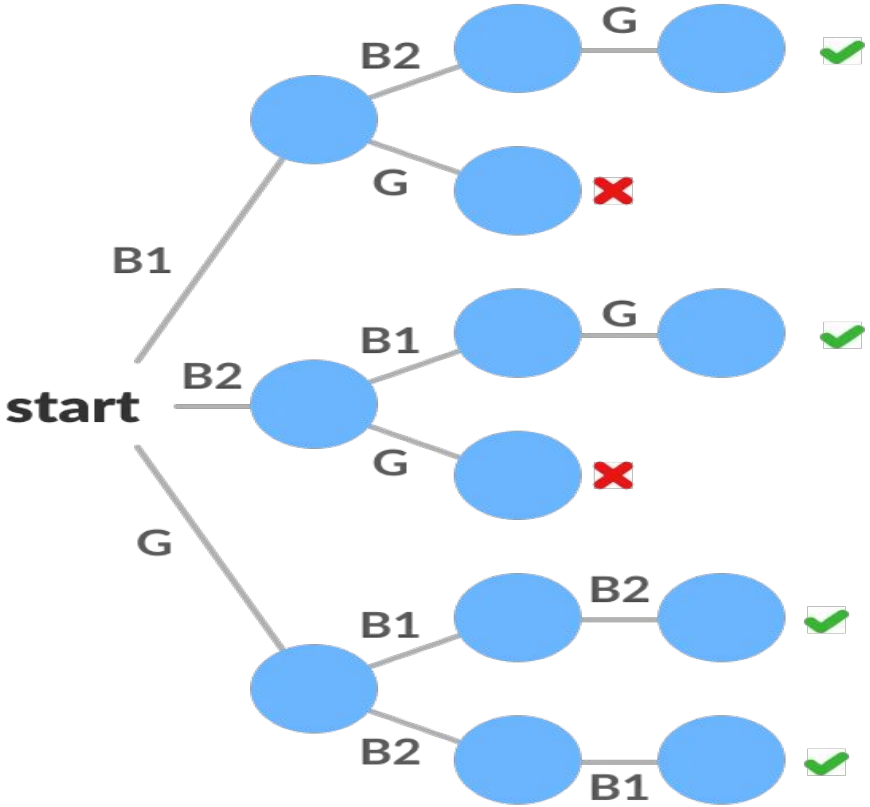
Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches.

Constraint/Bounding Function/Condition: Girl should not be on the middle bench.

All the possibilities are:



The following **state space tree** shows the possible solutions.

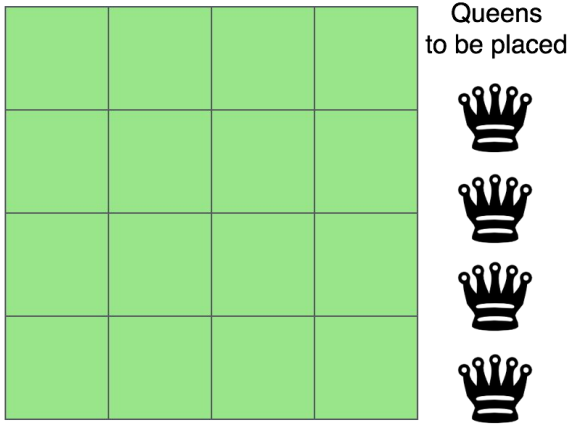


Backtracking Algorithm Applications

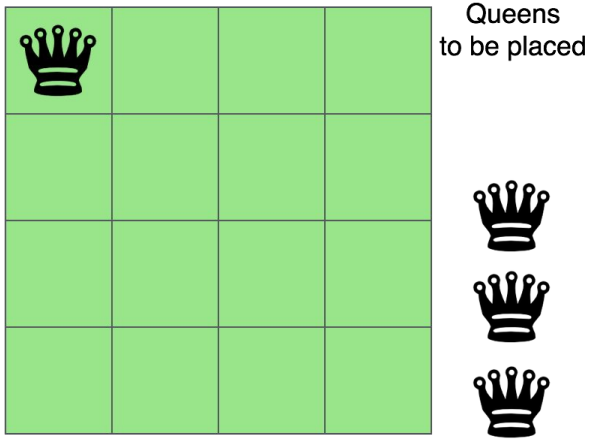
1. To find all Hamiltonian Paths present in a graph.
2. To solve the N Queen problem.
3. Maze solving problem.
4. The Knight's tour problem.

N Queens / 4 Queens problem on NxN Chessboard

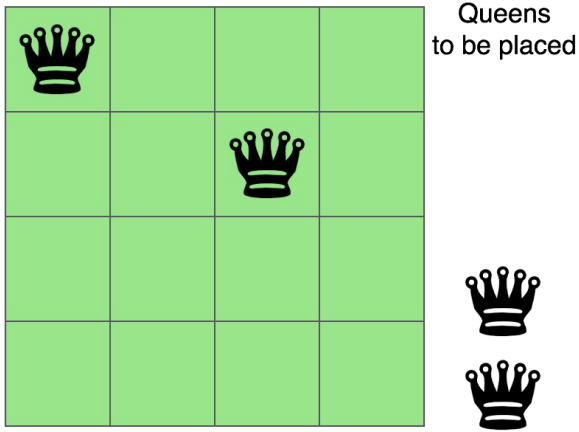
- One of the most common examples of the backtracking is to arrange **N queens on an NxN chessboard** such that no queen can **strike down any other queen**.
- A queen **can attack horizontally, vertically, or diagonally.**
- The solution to this problem is also attempted in a similar way.
- We first place the **first queen anywhere arbitrarily** and then place the **next queen in any of the safe places.**
- We continue this process **until the number of unplaced queens becomes zero** (a solution is found) or no safe place is left.
- If **no safe place is left, then we change the position of the previously placed queen.**



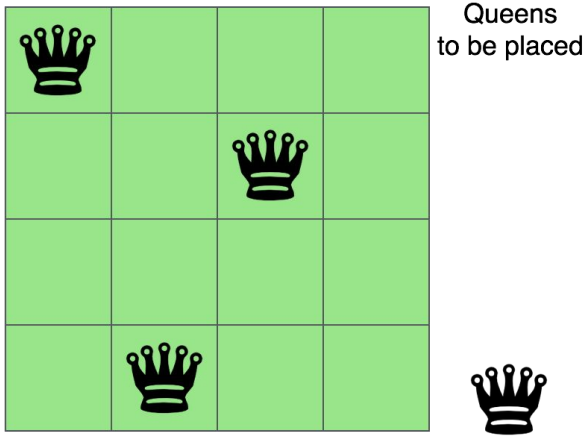
The above picture shows an NxN chessboard and we have to **place N queens on it.** So, we will start by placing the first queen.



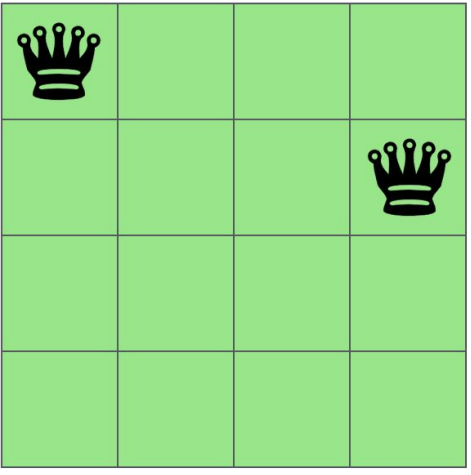
Now, the **second step is to place the second queen in a safe position** and then the third queen.



Now, you can see that **there is no safe place where we can put the last queen.** So, we will just **change the position of the previous queen.** And this is **backtracking.**



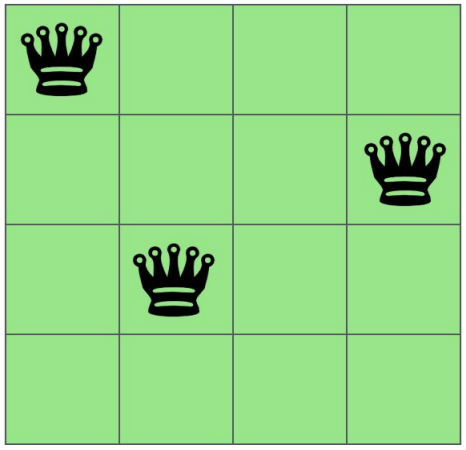
Also, there is **no other position where we can place the third queen** so we will go back one more step and change the position of the second queen.



Queens to be placed

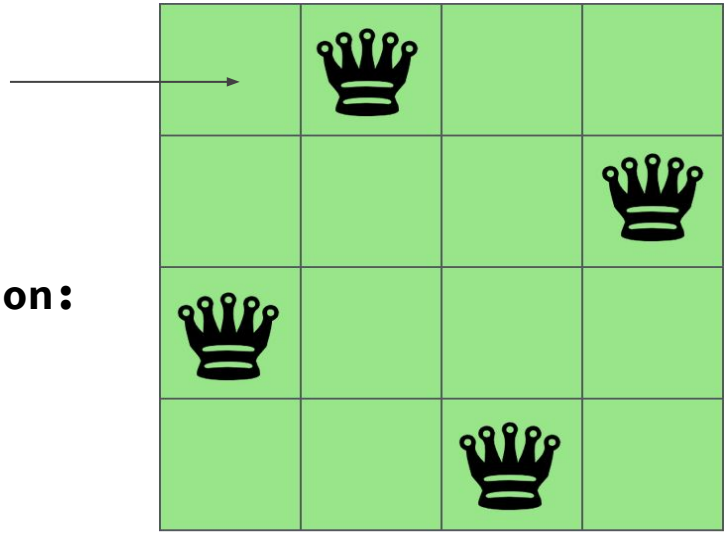
And now we will place the third queen again in a safe position until we find a solution.

We will continue this process and finally, we will get the solution as shown below.



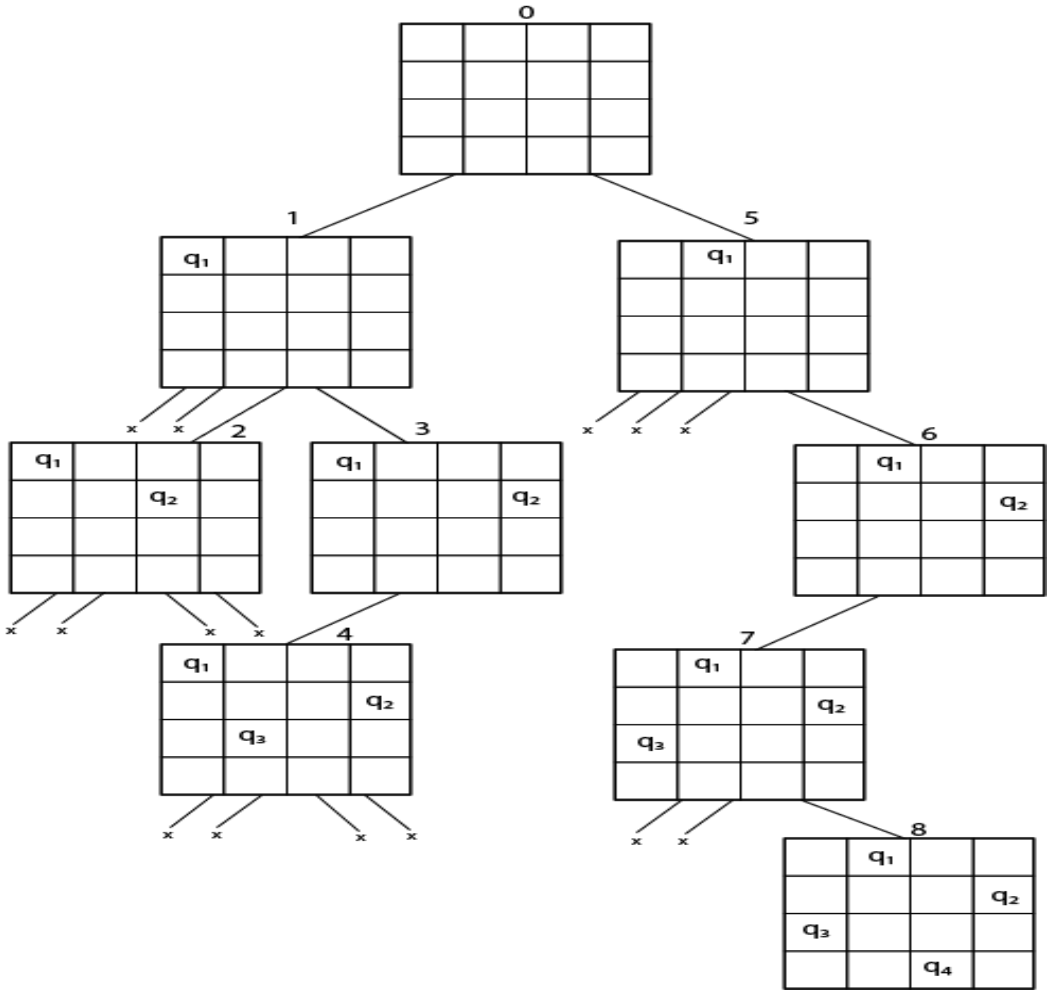
Queens to be placed

Solution:



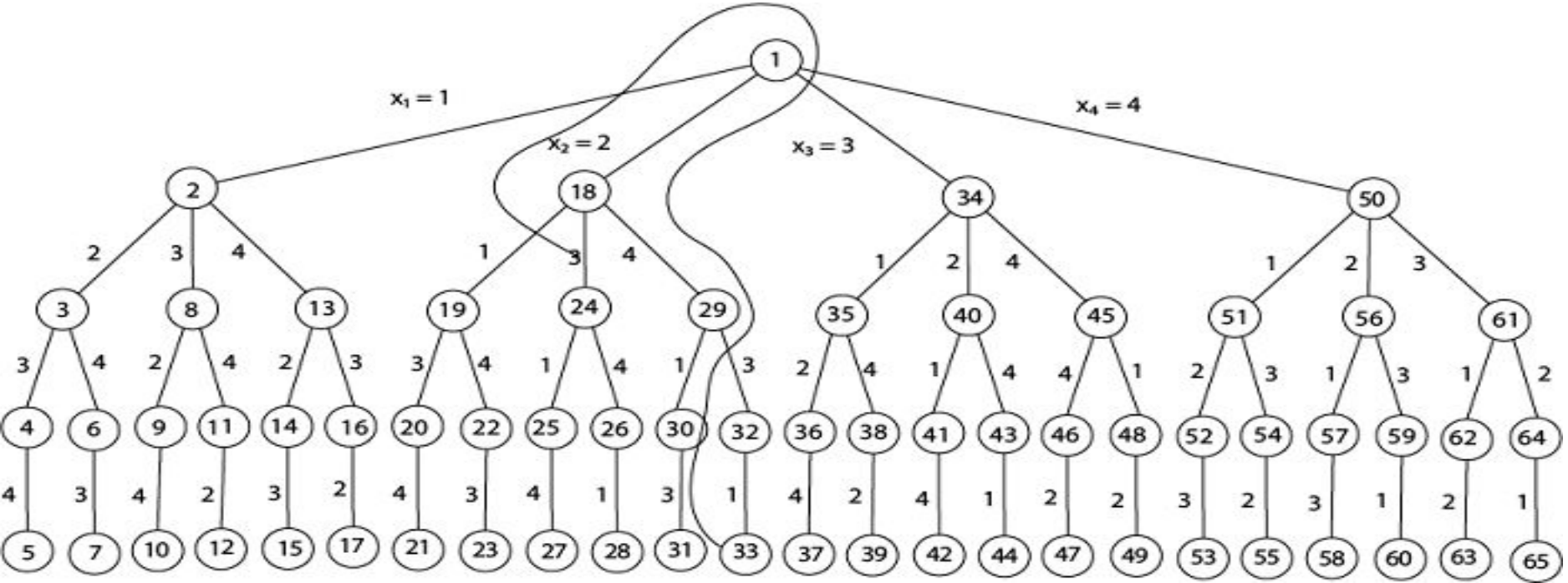
The implicit tree for N - queens problem for a solution (2, 4, 1, 3) is as follows:

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		



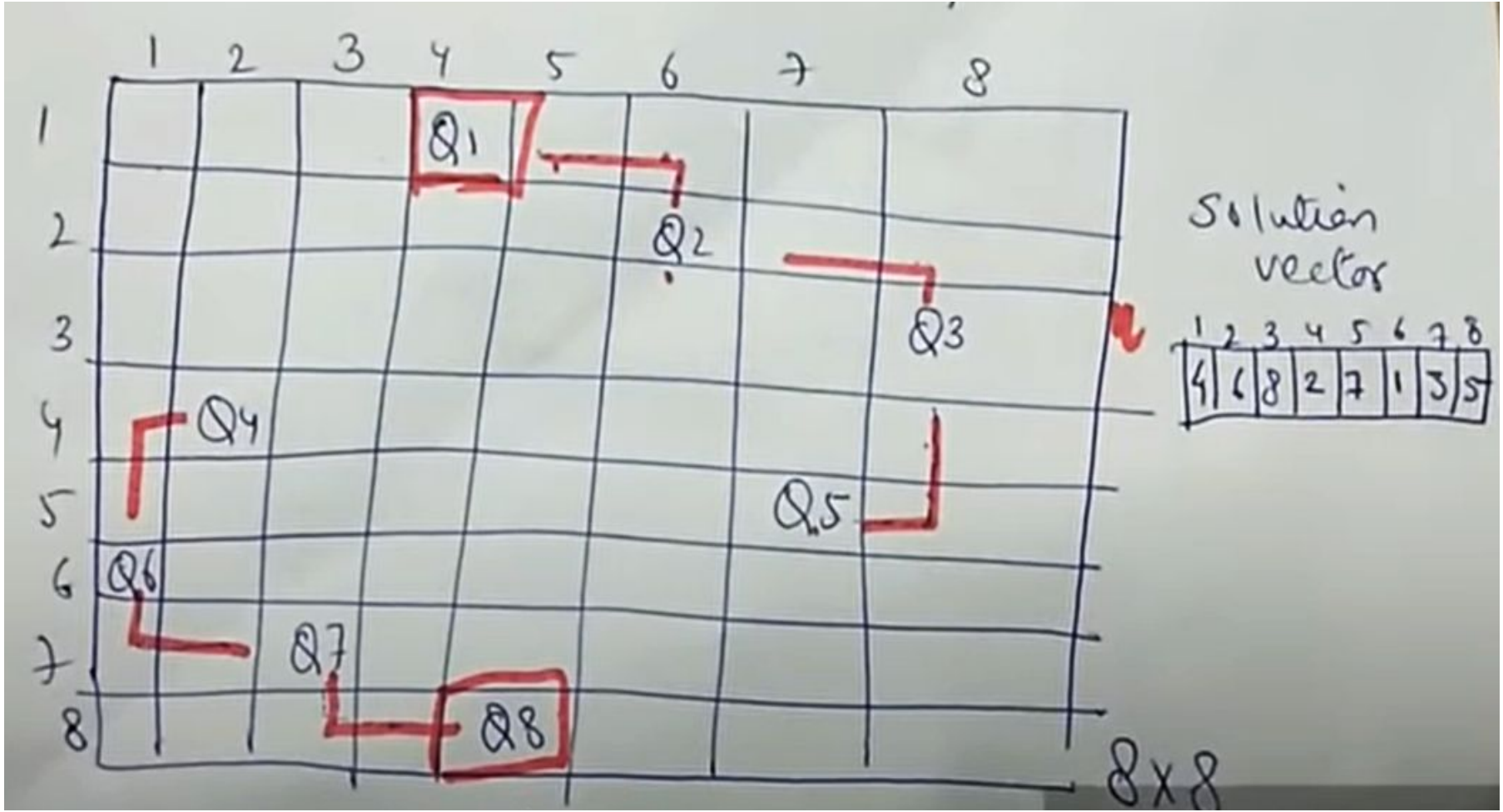
4 - Queens solution State Space Tree with nodes numbered in DFS

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		



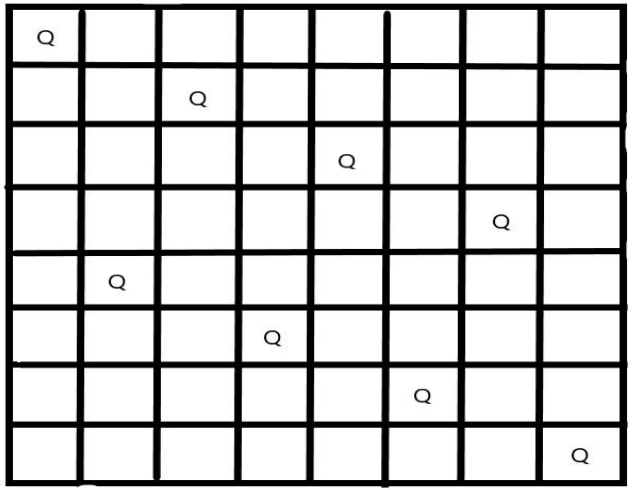
8 Queens Problem

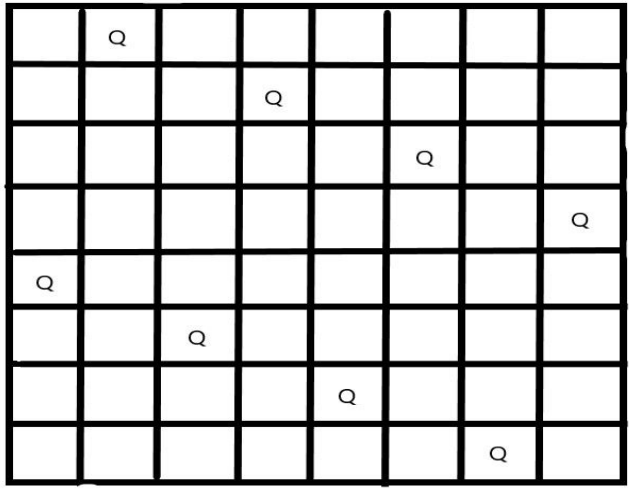
- The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal).
- More generally, the n queens problem places n queens on an n×n chessboard.


1 2 3 4 5 6 7 8
1
2
3
4
5
6
7
8
Q1
Q2
Q3
Q4
Q5
Q6
Q7
Q8
8x8
Solution vector
1 2 3 4 5 6 7 8
4 6 8 2 7 1 3 5

Possible Solutions

	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			



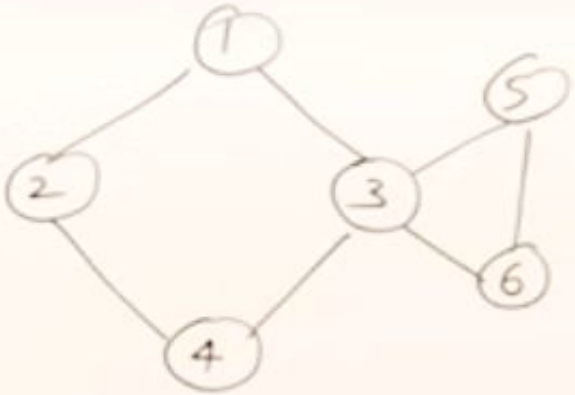


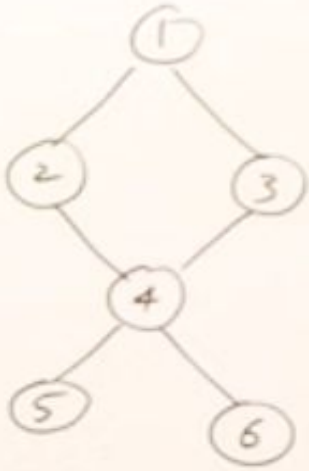
Backtracking Algorithm

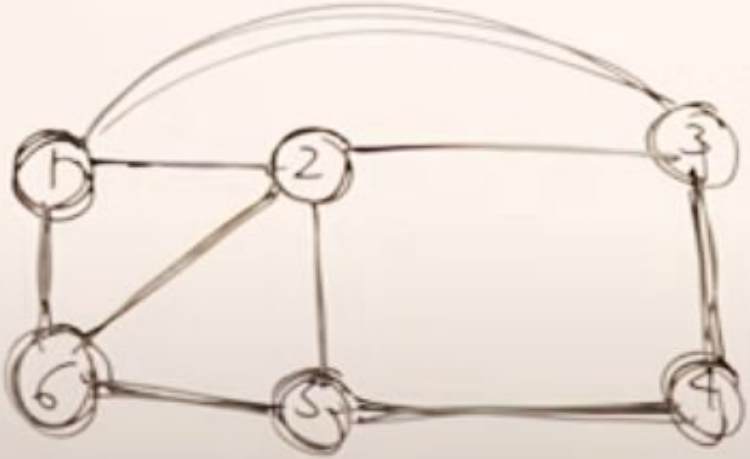
1) Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column.
   Do following for every tried row.
    a) If the queen can be placed safely in this row
       then mark this [row, column] as part of the
       solution and recursively check if placing
       queen here leads to a solution.
    b) If placing the queen in [row, column] leads to
       a solution then return true.
    c) If placing queen doesn't lead to a solution then
       unmark this [row, column] (Backtrack) and go to
       step (a) to try other rows.
3) If all rows have been tried and nothing worked,
   return false to trigger backtracking.

Hamiltonian Cycle

- Hamiltonian Path is a path in a directed or undirected graph that visits each vertex exactly once.
- The problem to check whether a graph (directed or undirected) contains a Hamiltonian Path is NP-complete, so is the problem of finding all the Hamiltonian Paths in a graph.
- Following images explains the idea behind Hamiltonian Path more clearly.







1, 2, 3, 4, 5, 6, 1

1, 2, 6, 5, 4, 3, 1

1, 6, 2, 5, 4, 3, 1

2, 3, 4, 5, 6, 1, 2

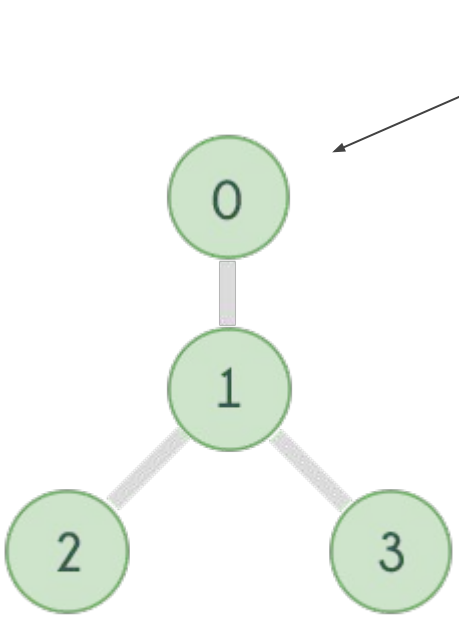


Fig. 1

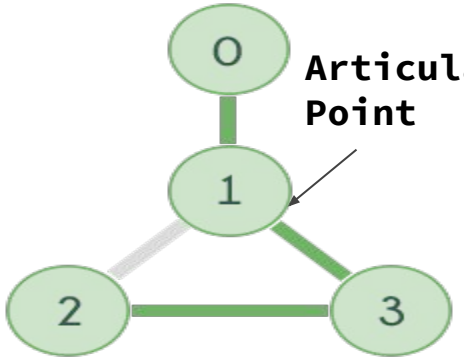


Fig. 4

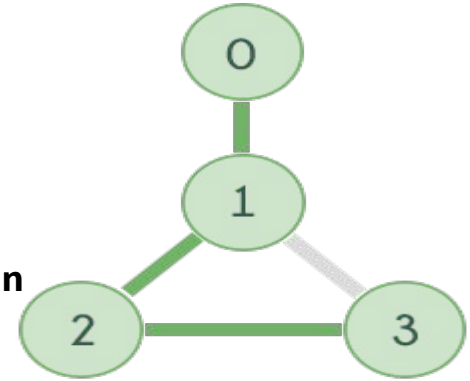


Fig. 3

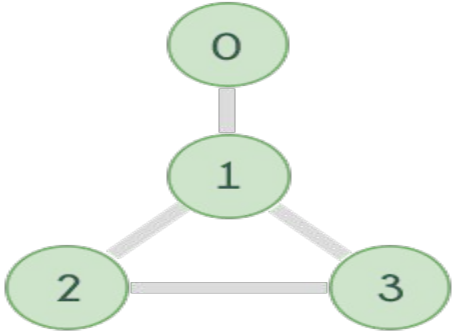
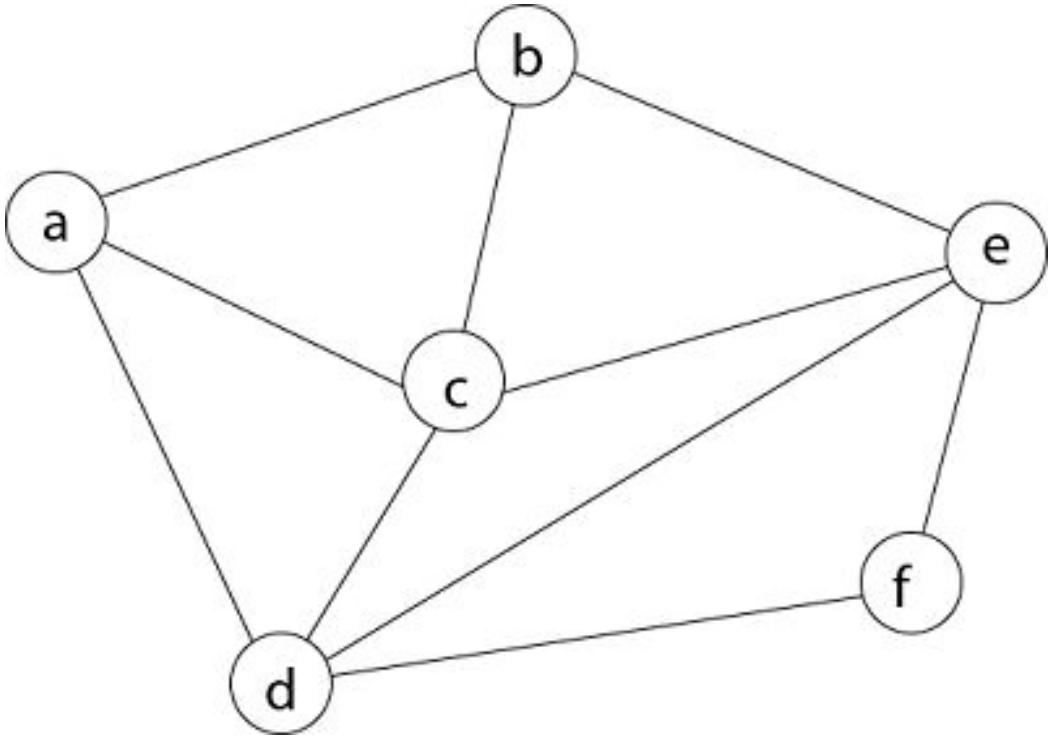


Fig. 2

Hamiltonian Circuit/Cycle Problem

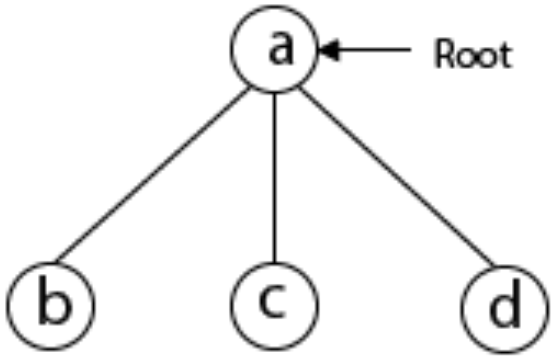
- Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach.
- We start our search from any **arbitrary vertex** say **'a.'** This vertex **'a' becomes the root of our implicit tree.**
- The first element of our partial solution is the first intermediate vertex of the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

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



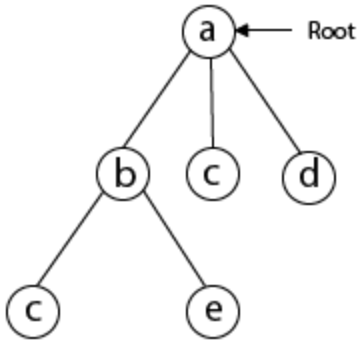
Solution: Firstly, we start our search with vertex 'a.' this vertex 'a' becomes the root of our implicit tree.

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

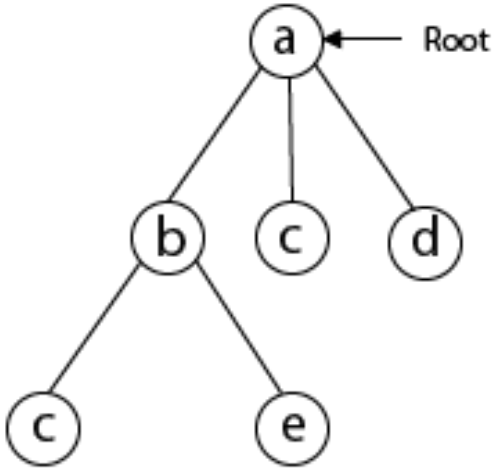


Next, we select 'c' adjacent to 'b.'

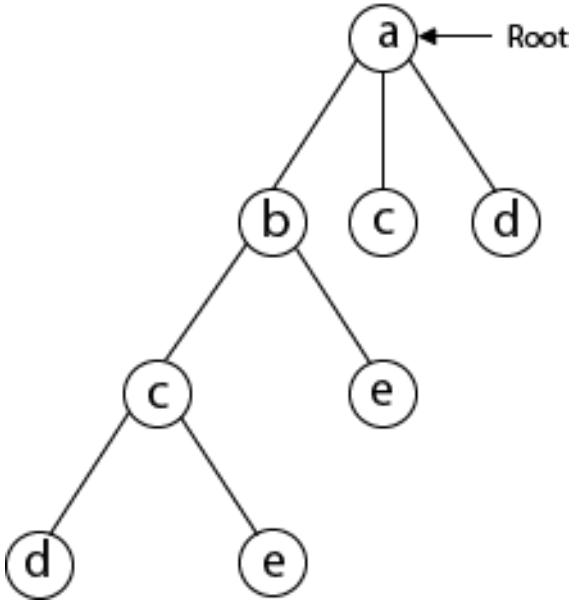
Next, we select 'd' adjacent to 'c.'



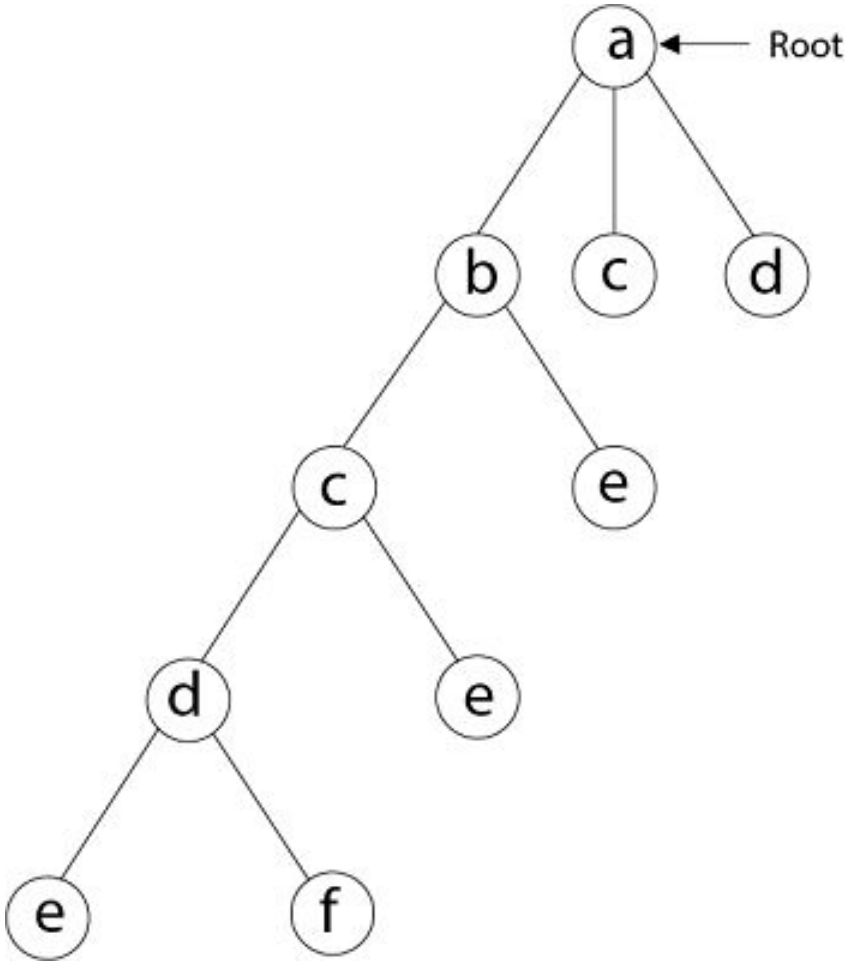
Next, we select 'd' adjacent to 'c.'



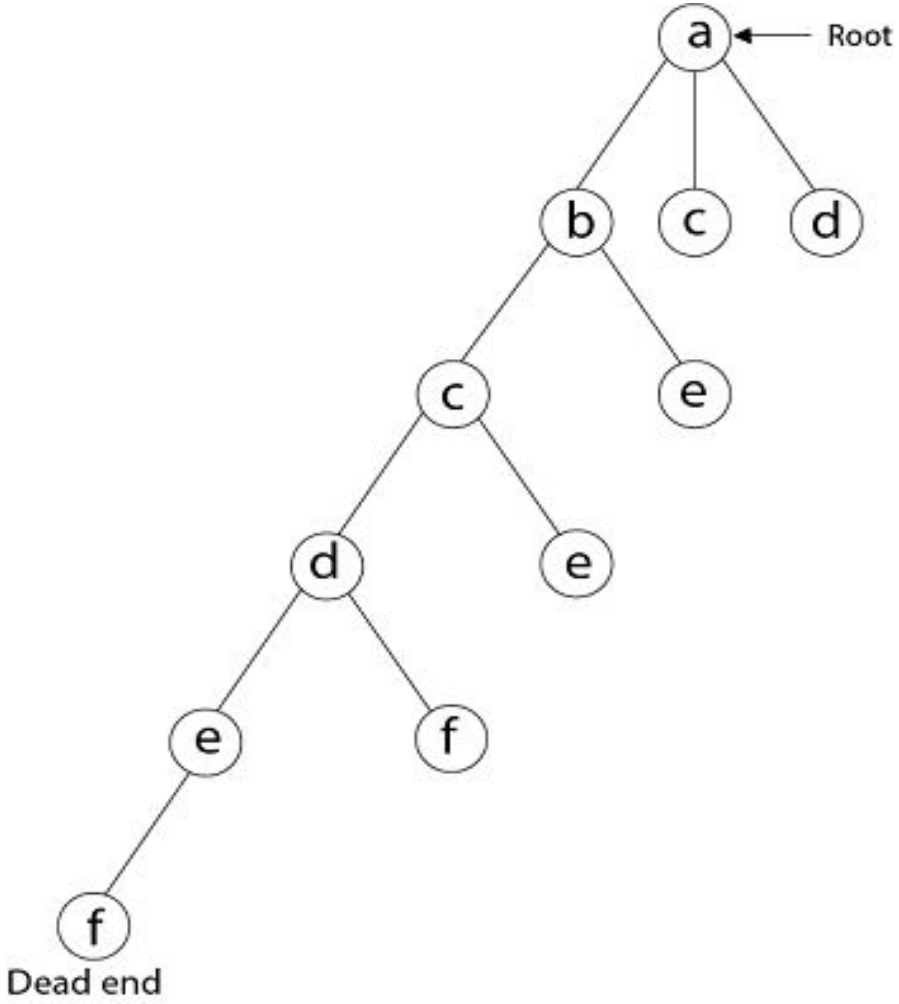
Next, we select 'e' adjacent to 'd.'



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

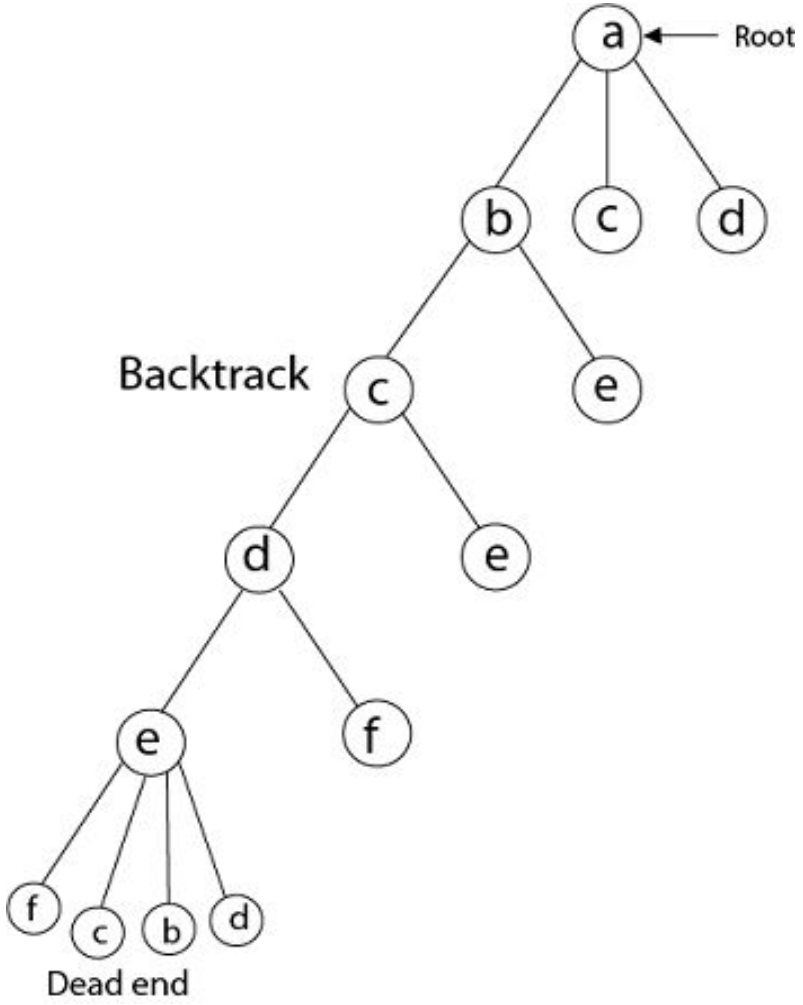


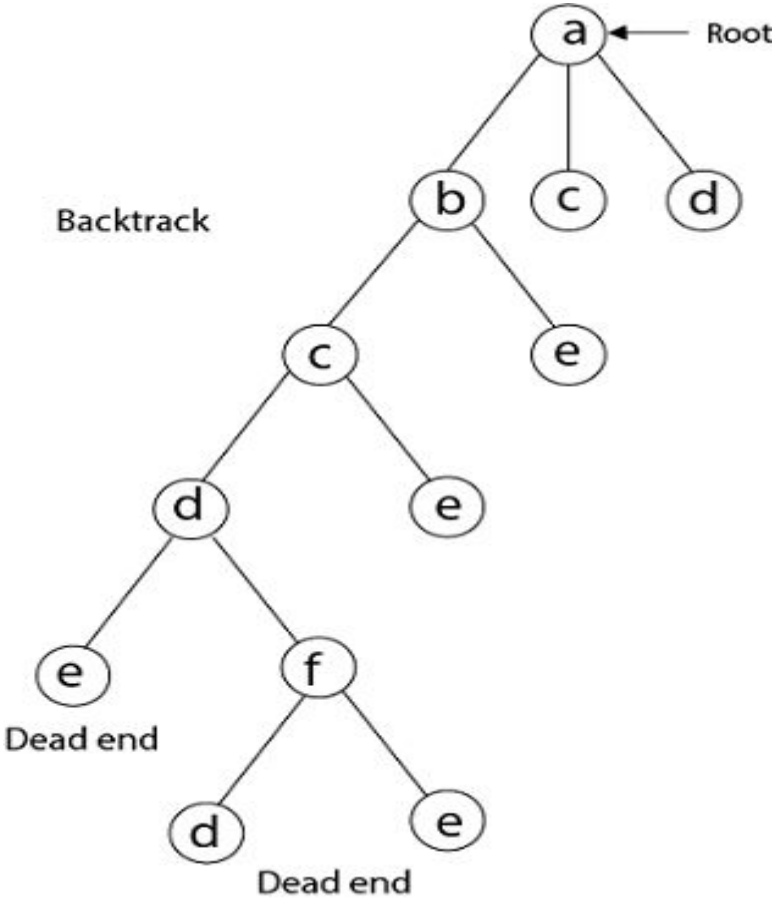
From backtracking, the vertex adjacent to 'e' is b, c, d, and f from which vertex 'f' has already been checked, and b, c, d have already visited. So, again we backtrack one step. Now, the vertex adjacent to d are e, f from which e has already been checked, and adjacent of 'f' are d and e. If 'e' vertex,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

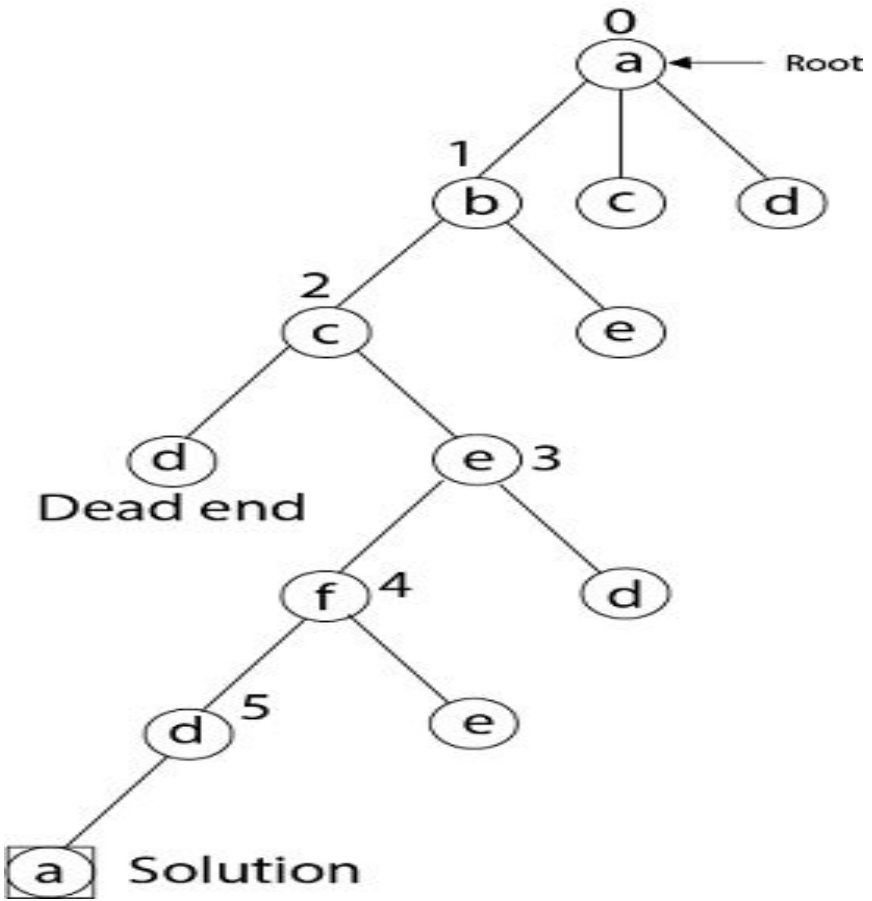


Now, adjacent to c is 'e' and adjacent to 'e' is 'f' and adjacent to 'f' is 'd' and adjacent to 'd' is 'a.' Here, we get the Hamiltonian Cycle as all the vertex other than the start vertex 'a' is visited only once.

(a - b - c - e - f -d - a).





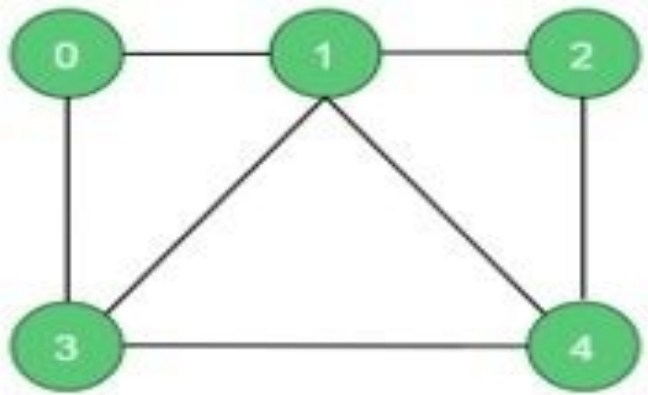


Here we have generated one Hamiltonian circuit, but another Hamiltonian circuit can also be obtained by considering another vertex.

Input:

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

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



Output:

The algorithm finds the Hamiltonian path of the given graph. For this case it is (0, 1, 2, 4, 3, 0). This graph has some other Hamiltonian paths.

If one graph has no Hamiltonian path, the algorithm should return false.

Algorithm
isValid(v, k)
Input - Vertex v and position k.
Output - Checks whether placing v in the position k is valid or not.

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

```
cycleFound(node k)
Input - node of the graph.
Output - True when there is a Hamiltonian Cycle, otherwise false.

Begin
   if all nodes are included, then
      if there is an edge between nodes k and 0, then
         return true
      else
         return false;

   for all vertex v except starting point, do
      if isValid(v, k), then //when v is a valid edge
         add v into the path
         if cycleFound(k+1) is true, then
            return true
         otherwise remove v from the path
   done
   return false
End
```

Sum of Subsets Problem

- In this problem, there is a given set with **some integer elements.**
- And another some value is also provided, we have to find a subset of the **given set whose sum is the same as the given sum value.**
- Here backtracking approach is used for **trying to select a valid subset when an item is not valid, we will backtrack to get the previous subset and add another element to get the solution.**

Steps:

1. Start with an empty set
2. Add the next element from the list to the set
3. If the subset is having sum M, then stop with that subset as solution.
4. If the subset is not feasible or if we have reached the end of the set, then backtrack through the subset until we find the most suitable value.
5. If the subset is feasible (sum of seubset < M) then go to step 2.
6. If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

Input and Output

Input:

This algorithm takes a set of numbers, and a sum value.

The Set: {10, 7, 5, 18, 12, 20, 15}

The sum Value: 35

Output:

All possible subsets of the given set, where sum of each element for every subsets is same as the given sum value.

{10, 7, 18}

{10, 5, 20}

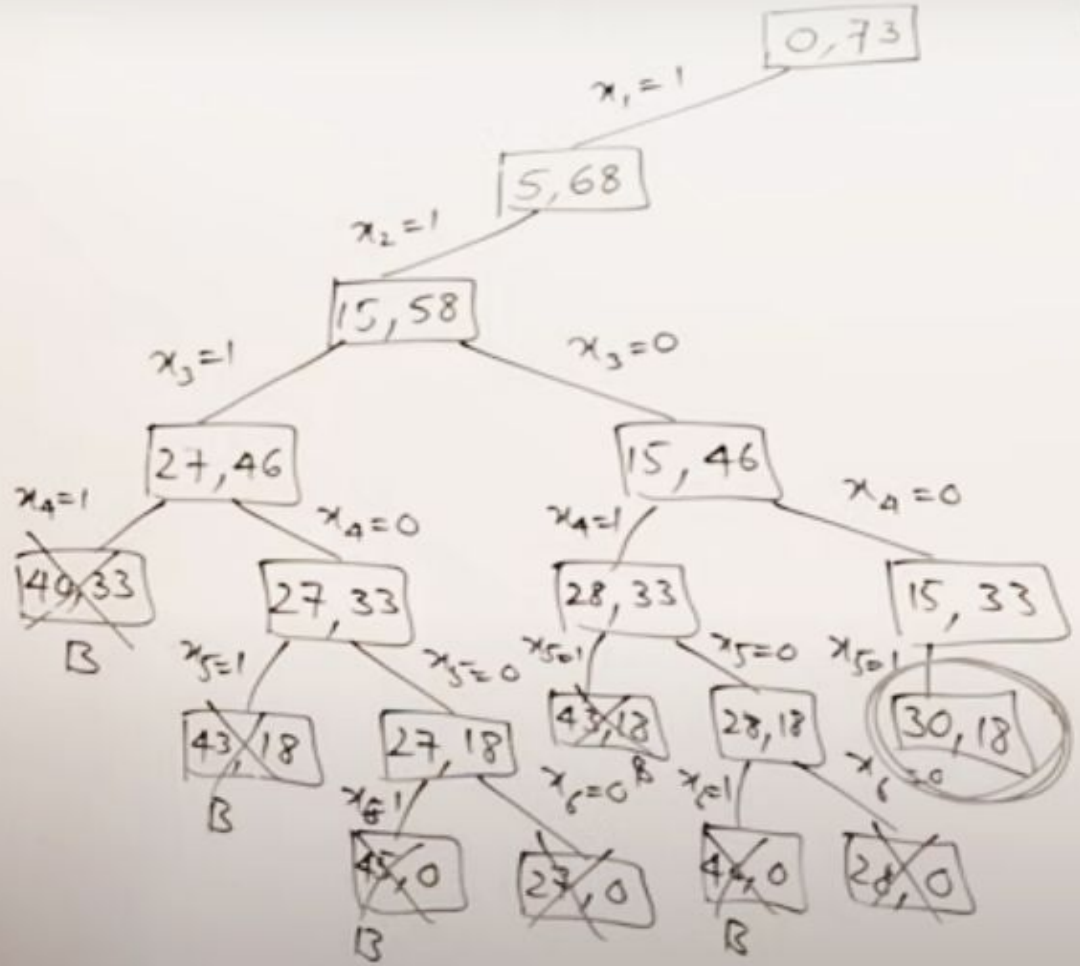
{5, 18, 12}

{20, 15}

Sum of subsets

$w[1:6] = \{5, 10, 12, 13, 15, 18\}$

$n = 6 \quad m = 30$



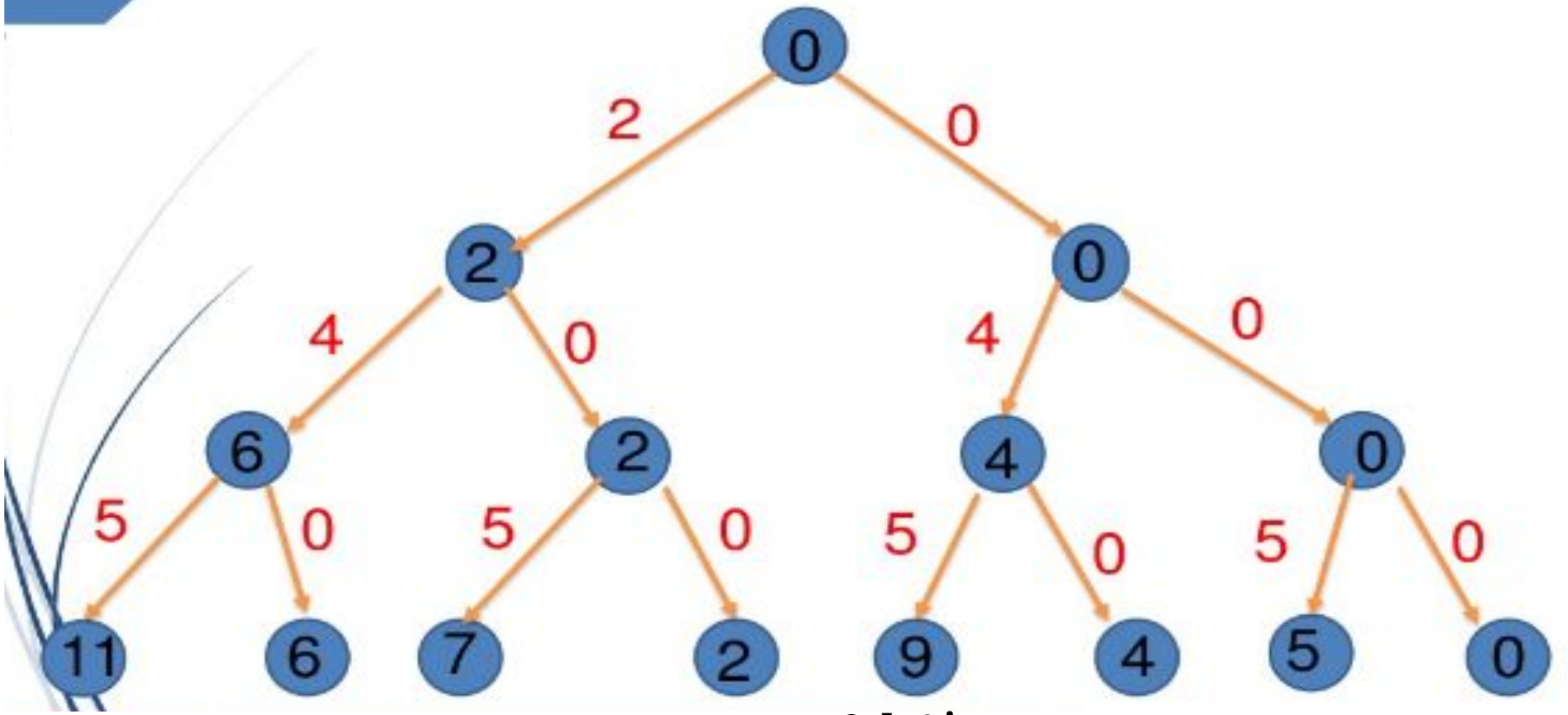
x

1	2	3	4	5	6
1	1	0	0	1	0

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m$$

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

A={2,4,5} and m=9



Solution

Algorithm

```
subsetSum(set, subset, n, subSize, total, node, sum)
Input - The given set and subset, size of set and subset, a total of
the subset, number of elements in the subset and the given sum.
Output - All possible subsets whose sum is the same as the given sum.
Begin
   if total = sum, then
      display the subset
      //go for finding next subset
   subsetSum(set, subset, , subSize-1, total-set[node], node+1, sum)
      return
   else
      for all element i in the set, do
         subset[subSize] := set[i]
     subSetSum(set, subset, n, subSize+1, total+set[i], i+1, sum)
      Done
End
```

Graph Coloring Problem

- Graph Colouring Problem Graph colouring problem of states each vertex in a graph in such a way that no two adjacent vertices have same colour.
- This problem is called as m colouring problem. This is also called a vertex colouring.
- If the degree of a given graph is d then we can colour it with d + 1 colour.
- There exists no efficient algorithm for coloring a graph with minimum number of colors.
- Graph Coloring is a NP complete problem.

What is graph coloring problem?

Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints. This has found applications in numerous fields in computer science.

- **Sudoku**: This game is a variation of Graph coloring problem where every cell denotes a node (or vertex) and there exists an edge between two nodes if the nodes are in same row or same column or same block.
- **Geographical maps**: There can be cases when no two adjacent cities/states can be assigned same color in the maps of countries or states. In this case, only four colors would be sufficient to color any map.

What is graph coloring problem?...

Vertex coloring is the most commonly encountered graph coloring problem. The problem states that given `m` colors, determine a way of coloring the vertices of a graph such that no two adjacent vertices are assigned same color.

Note: The smallest number of colors needed to color a graph G is called its chromatic number.

For example, the following undirected graph can be colored using minimum of 2 colors.

Hence the **chromatic number** of the graph is 2.

Steps..

Step-01:

Color first vertex with the first color.

Step-02:

Now, consider the remaining (V-1) vertices one by one and do the following-

Color the currently picked vertex with the lowest numbered color if it has not been used to color any of its adjacent vertices.

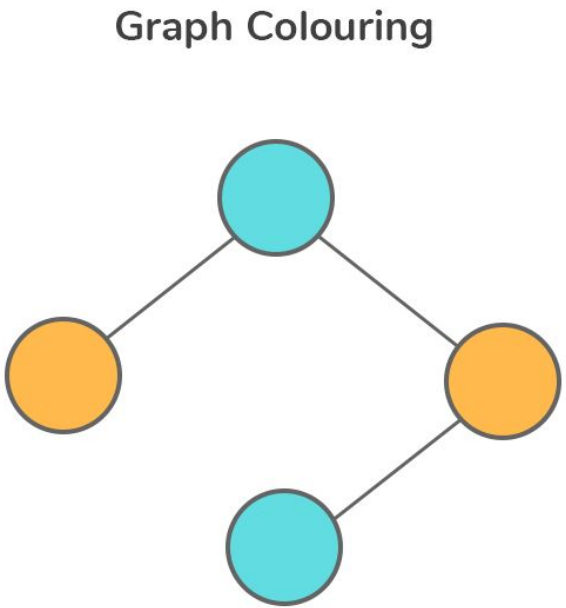
If it has been used, then choose the next least numbered color.

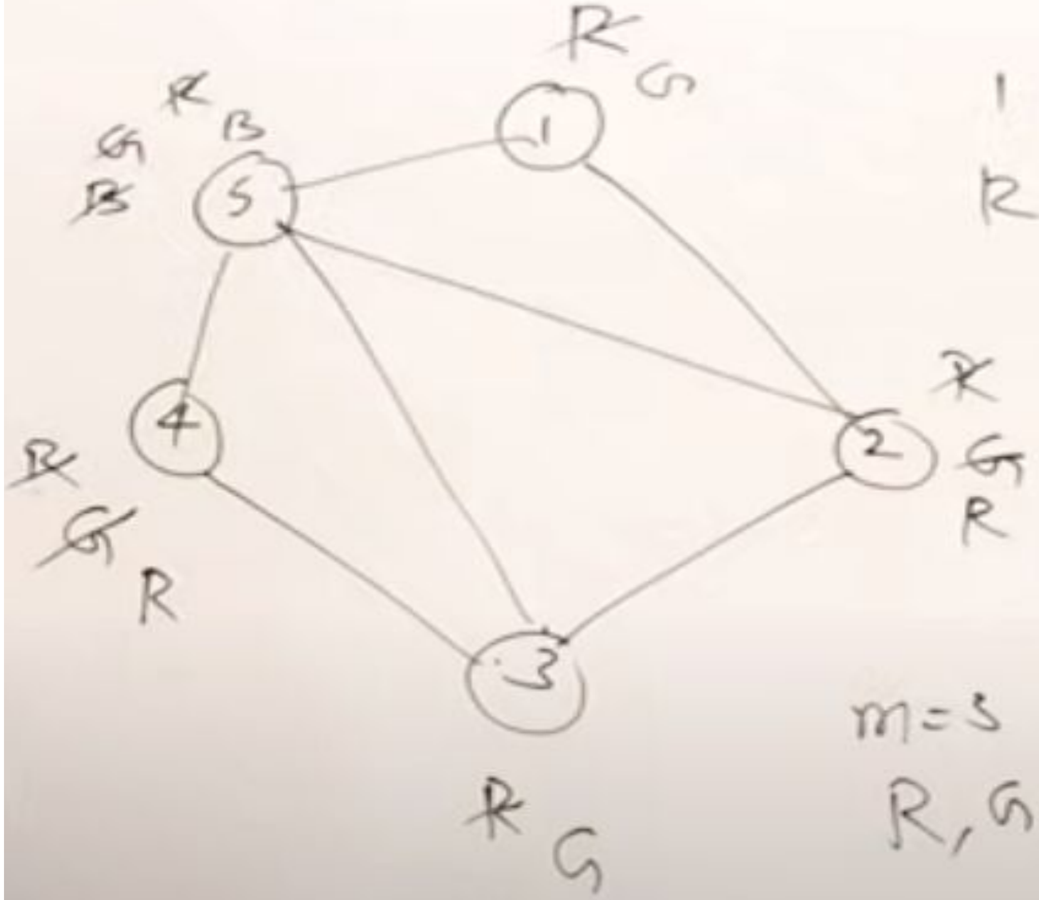
If all the previously used colors have been used, then assign a new color to the currently picked vertex.

Algorithm:

1. Create a recursive function that takes the graph, current index, number of vertices, and output color array.
2. If the current index is equal to the number of vertices. Print the color configuration in output array.
3. Assign a color to a vertex (1 to m).
4. For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) recursively call the function with next index and number of vertices
5. If any recursive function returns true break the loop and return true.
6. If no recursive function returns true then return false.

What is graph coloring problem?...

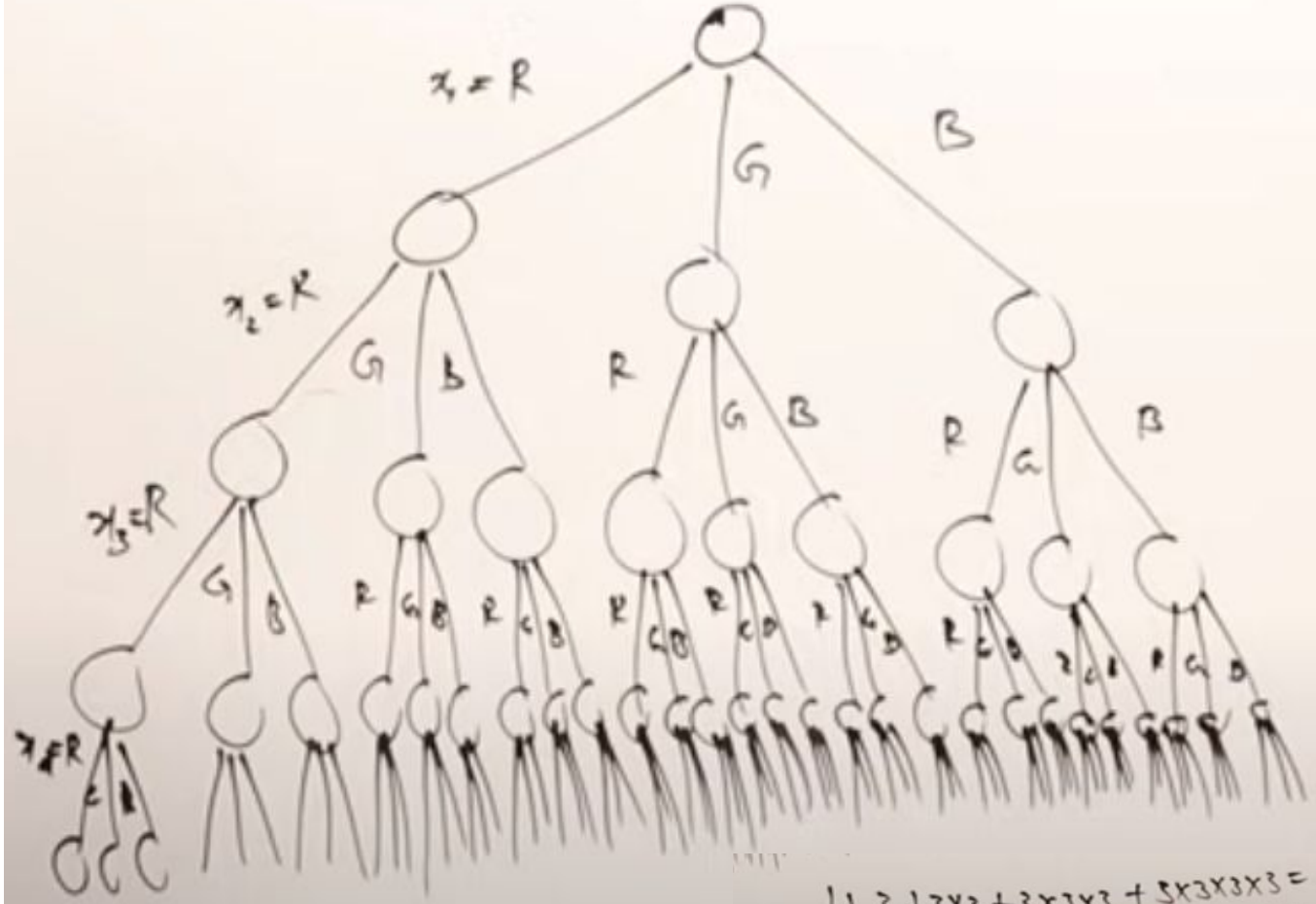


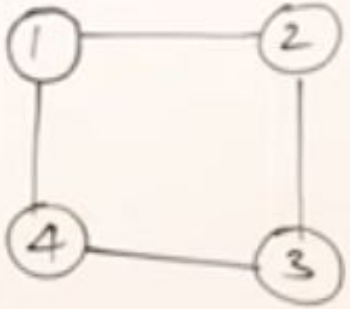


1	2	3	4	5
R	G	R	G	B

m = 3 4

R, G, B, Y





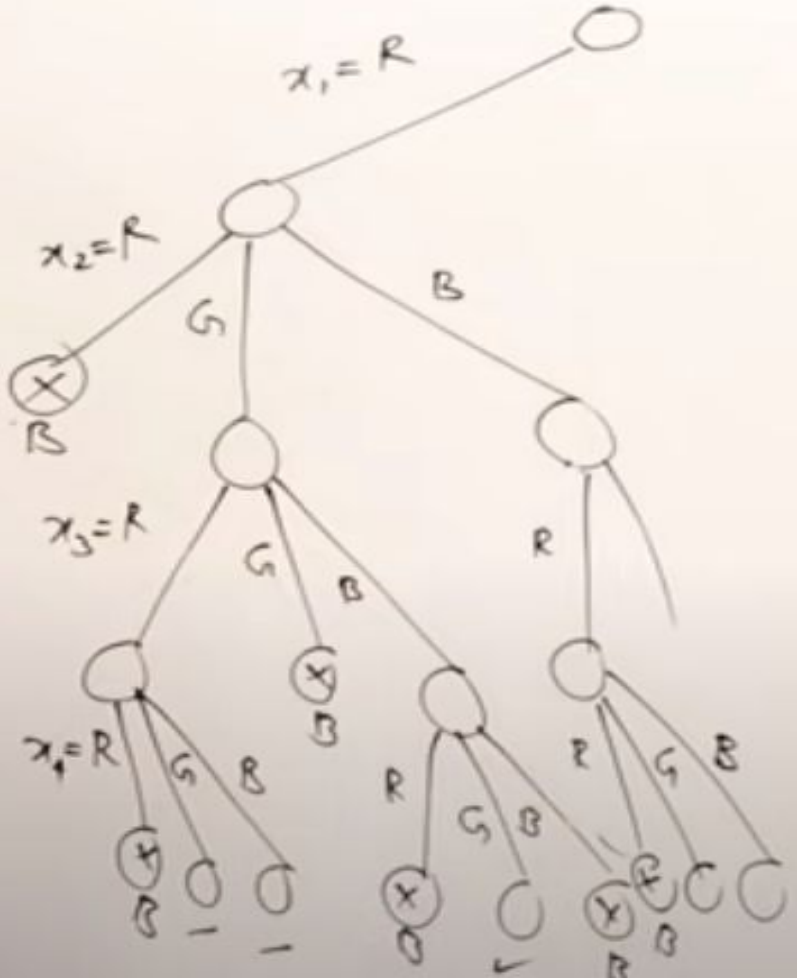
$m = 3$

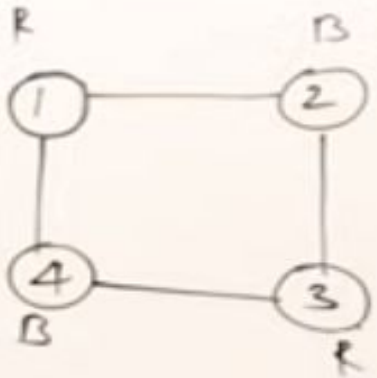
$\{R, G, B\}$

$3^{n+1} = \binom{n+1}{}$

$1 + 3 + 3 \times 3 + 3 \times 3 \times 3 + 3 \times 3 \times 3 \times 3 =$

$1 + 3 + 3^2 + 3^3 + 3^4 = \frac{3^{4+1} - 1}{3 - 1} = \frac{3^{n+1} - 1}{2}$

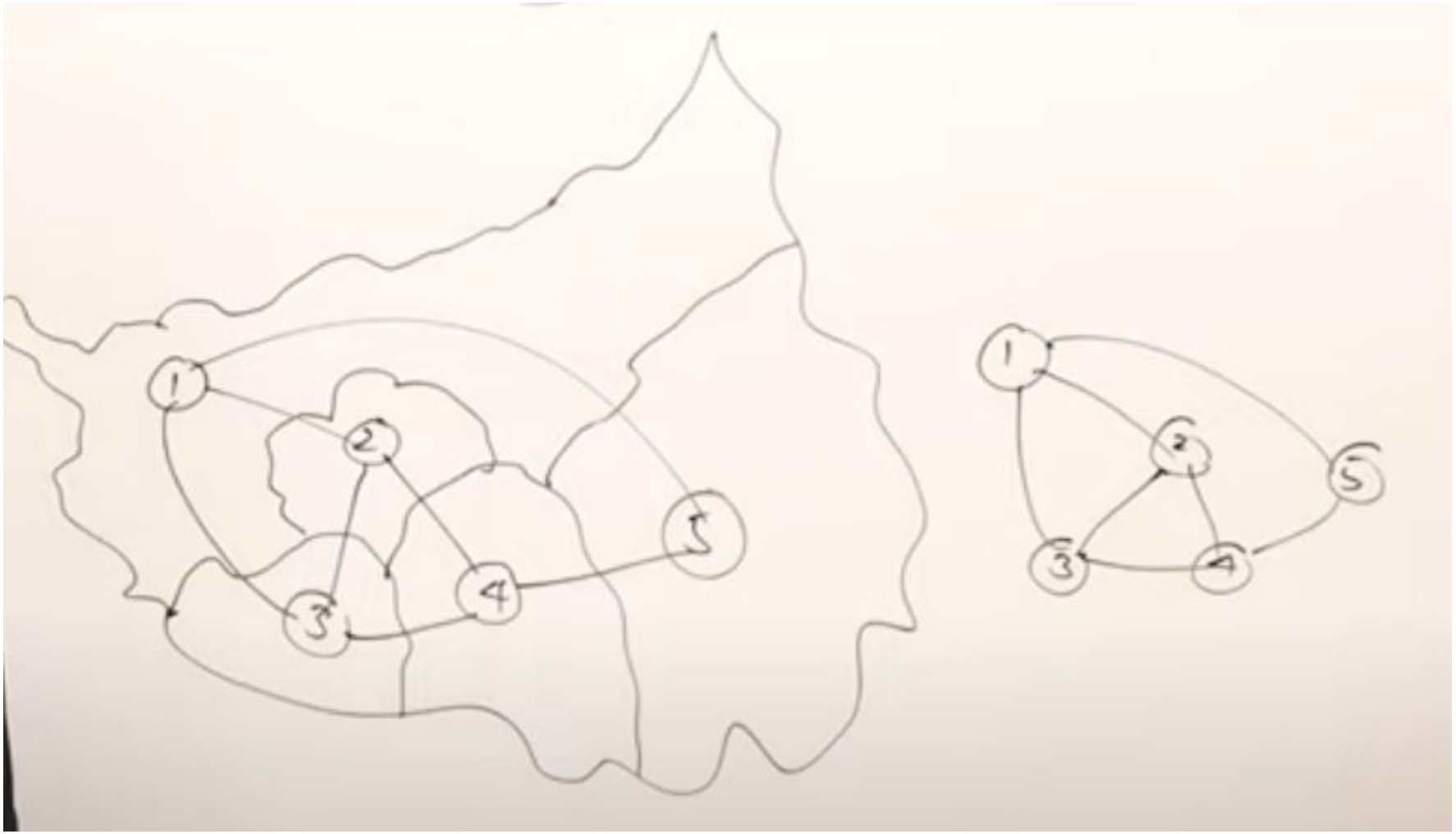




$m = 3$

$\{R, G, B\}$

1) R, G, R, G
2) R, G, R, B
3) R, G, B, G
4) R, B, R, G
5) R, B, R, B


1
2
3
4
5
1
2
3
4
5

Complexity Analysis:

Time Complexity: O(m^V).

There are total O(m^V) combination of colors. So time complexity is O(m^V). The upper bound time complexity remains the same but the average time taken will be less.

Space Complexity: O(V).

Recursive Stack of graphColoring(…) function will require O(V) space.

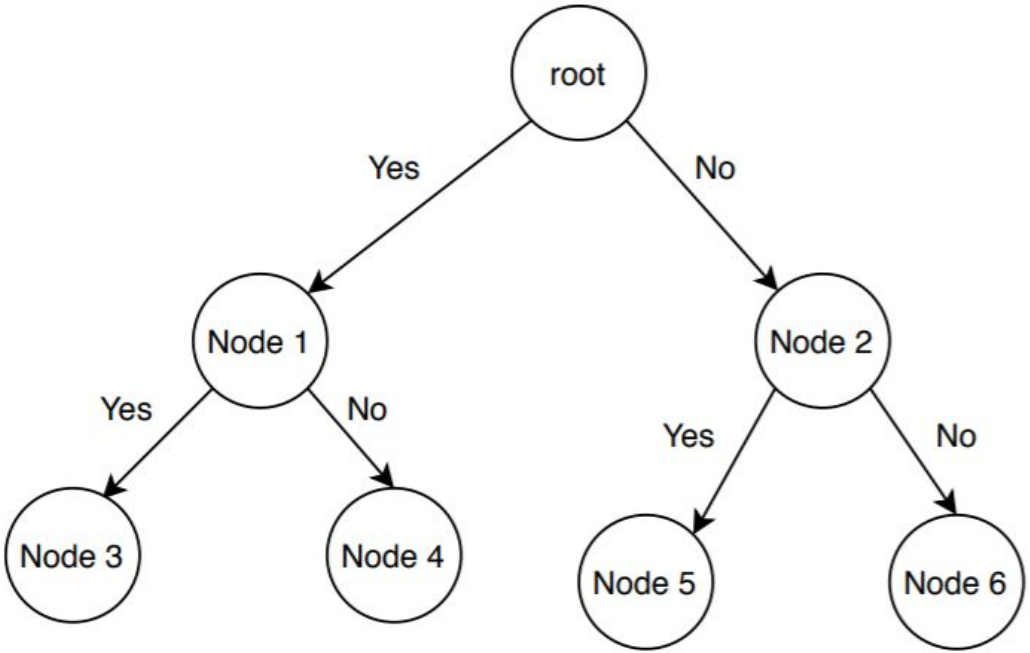
Branch and Bound

- In computer science, there is a **large number of optimization problems which has a finite but extensive number of feasible solutions.**
- Among these, some problems like **finding the shortest path in a graph or Minimum Spanning Tree can be solved in polynomial time.**
- A **significant number of optimization problems like production planning, crew scheduling can’t be solved in polynomial time,** and they belong to the NP-Hard class.
- These problems are the example of NP-Hard combinatorial optimization problem.
- Branch and bound (B&B) is an algorithm paradigm widely used for solving such problems.
- Branch and Bound follows BFS(Breadth First Search.

Branch and Bound....

- Branch and bound algorithms are used to find **the optimal solution for combinatory, discrete, and general mathematical optimization problems.**
- In general, given an NP-Hard problem, a branch and bound algorithm explores the entire search space of possible solutions and provides an optimal solution.
- A branch and bound algorithm consist of **stepwise enumeration of possible candidate** solutions by exploring the entire search space.
- With all the possible solutions, we first build a rooted decision tree.
- The root node represents the entire search space:

https://www.baeldung.com/cs/branch-and-bound



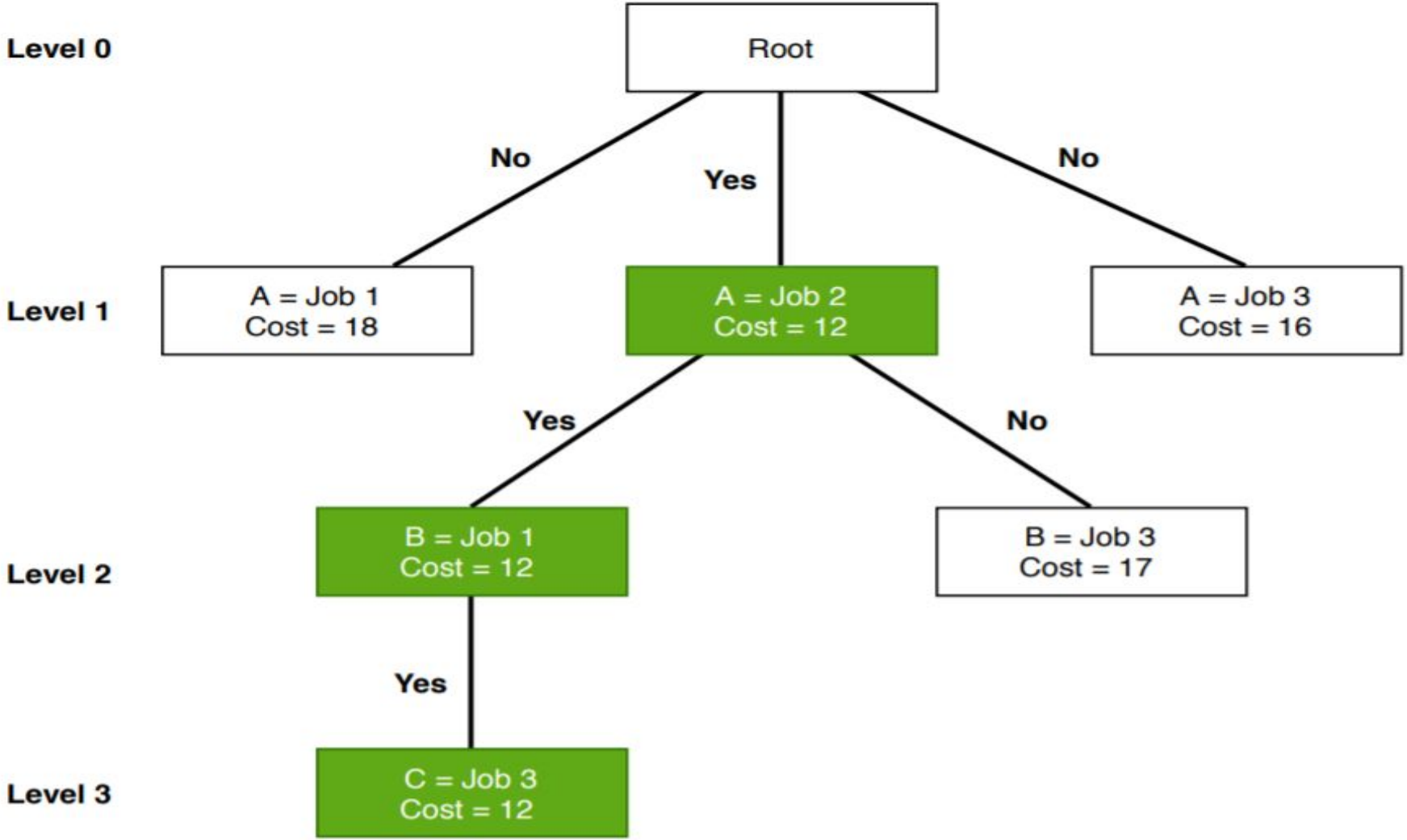
- Here, each child node is a partial solution and part of the solution set.
- Before constructing the rooted decision tree, we set an upper and lower bound for a given problem based on the optimal solution.
- At each level, we need to make a decision about which node to include in the solution set.
- At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.
- Now it is crucial to find a good upper and lower bound in such cases.

- We can find an upper bound by using any local optimization method or by picking any point in the search space.
- On the other hand, we can obtain a lower bound from convex relaxation or duality.
- In general, we want to partition the solution set into smaller subsets of solution.
- Then we construct a rooted decision tree, and finally, we choose the best possible subset (node) at each level to find the best possible solution set.

Branch and Bound Algorithm Example

Let’s first define a job assignment problem. In a standard version of a job assignment problem, there can be N jobs and N workers. To keep it simple, we’re taking 3 jobs and 3 workers in our example:

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


Level 0
Root
No
Yes
No
Level 1
A = Job 1
Cost = 18
A = Job 2
Cost = 12
A = Job 3
Cost = 16
Yes
No
Level 2
B = Job 1
Cost = 12
B = Job 3
Cost = 17
Yes
Level 3
C = Job 3
Cost = 12

Algorithm 1: Job Assignment Problem Using Branch And Bound

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

Advantages

1. In a branch and bound algorithm, we don’t explore all the nodes in the tree. That’s why the time complexity of the branch and bound algorithm is less when compared with other algorithms.
2. If the problem is not large and if we can do the branching in a reasonable amount of time, it finds an optimal solution for a given problem.
3. The branch and bound algorithm find a minimal path to reach the optimal solution for a given problem. It doesn’t repeat nodes while exploring the tree.

Disadvantages

1. The branch and bound algorithm are time-consuming. Depending on the size of the given problem, the number of nodes in the tree can be too large in the worst case.
2. Also, parallelization is extremely difficult in the branch and bound algorithm.

Travelling Salesman Problem-

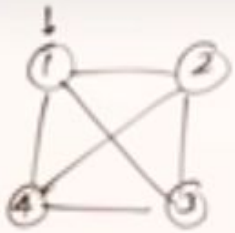
You are given-

A set of some cities

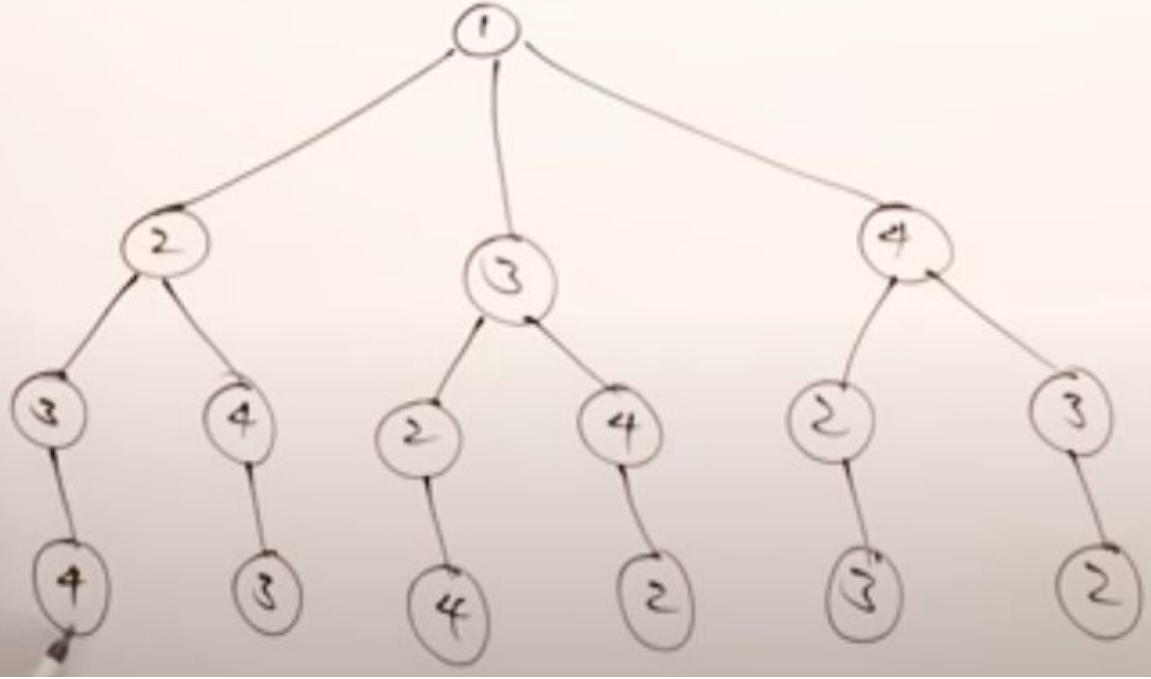
Distance between every pair of cities

Travelling Salesman Problem states-

1. A salesman has to visit every city exactly once.
2. He has to come back to the city from where he starts his journey.
3. What is the shortest possible route that the salesman must follow to complete his tour?

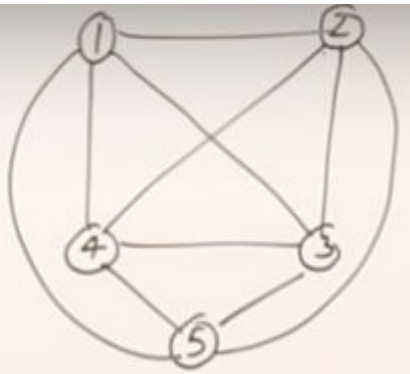


	1	2	3	4	5
1	∞	20	30	10	11
2	15	∞	16	4	2
3	3	5	∞	2	4
4	19	6	18	∞	3
5	16	4	7	16	∞



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

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



	1	2	3	4	5	
1	∞	10	20	0	1	10
2	13	∞	14	2	0	2
3	1	3	∞	0	2	2
4	16	3	15	∞	0	3
5	12	0	3	12	∞	4
						21

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

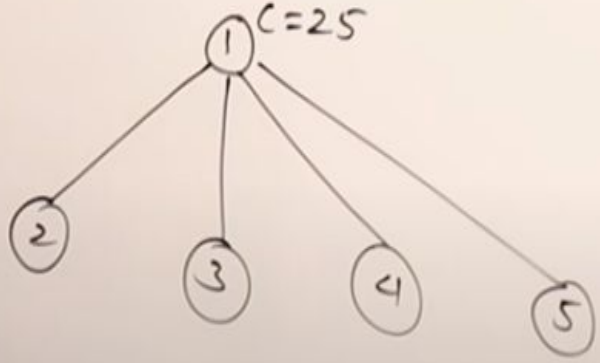
Traveling Salesperson

	1	2	3	4	5	
1	∞	10	17	0	1	10
2	12	∞	11	2	0	2
3	0	3	∞	0	2	2
4	15	3	12	∞	0	3
5	11	0	0	12	∞	4
	1	0	3	0	0	+ 21 + 4 = 25

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

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

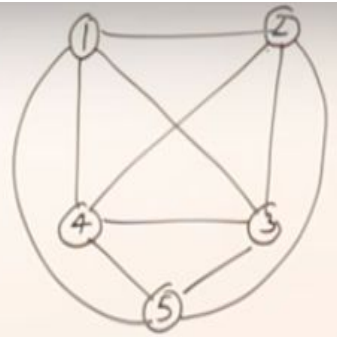
upper = ∞



	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

reduced cost = 25

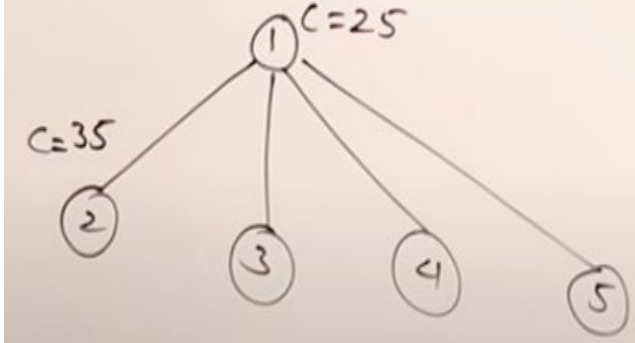
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	∞	∞	11	2	0
3	0	∞	∞	0	2
4	15	∞	12	∞	0
5	11	∞	0	12	∞



$C(1,2) + r + \hat{r}$

$10 + 25 + 0$

upper = ∞



	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

reduced cost = 25

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

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

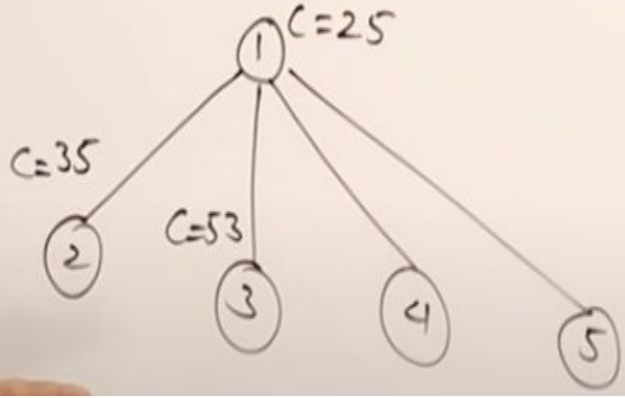
	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

reduced cost = 25

upper = ∞

$C(1,3) + r + \hat{r}$

17 + 25 + 11



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

(2)

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

(3)

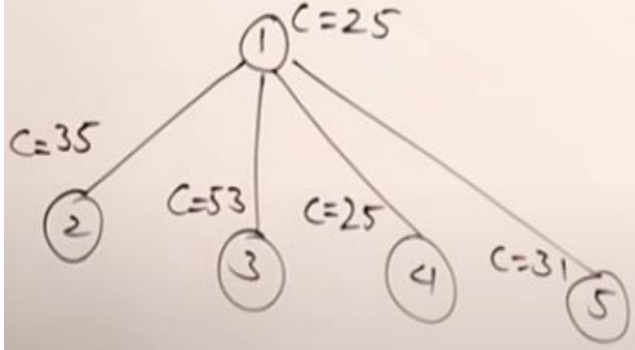
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

(4)

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

(5)

upper = ∞



	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

reduced cost = 25

Traveling Salesperson

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

(2)

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

(3)

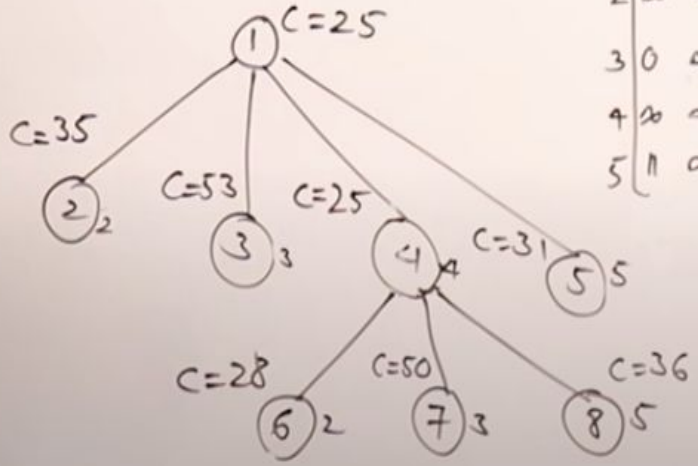
	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	12	∞	11	∞	0
3	0	3	∞	∞	2
4	∞	3	12	∞	0
5	11	0	0	∞	∞

(4)

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

(5)

upper = ∞



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

(6)

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

(7)

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

(8)

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

reduced cost = 25

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

(2)

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

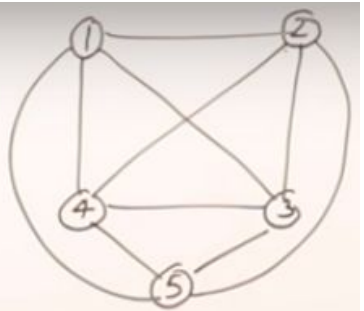
(3)

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

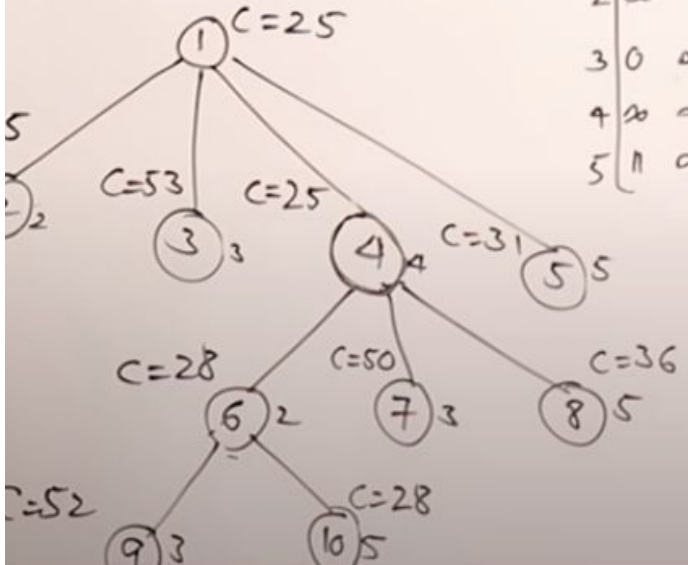
(4)

	1	2	3	4	5
1	∞	∞	∞	∞	∞
2	10	∞	9	0	∞
3	0	3	∞	0	∞
4	12	0	9	∞	∞
5	∞	0	0	12	∞

(5)



upper = ∞



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

(6)

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

(7)

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

(8)

	1	2	3	4	5
1	∞	10	17	0	1
2	12	∞	11	2	0
3	0	3	∞	0	2
4	15	3	12	∞	0
5	11	0	0	12	∞

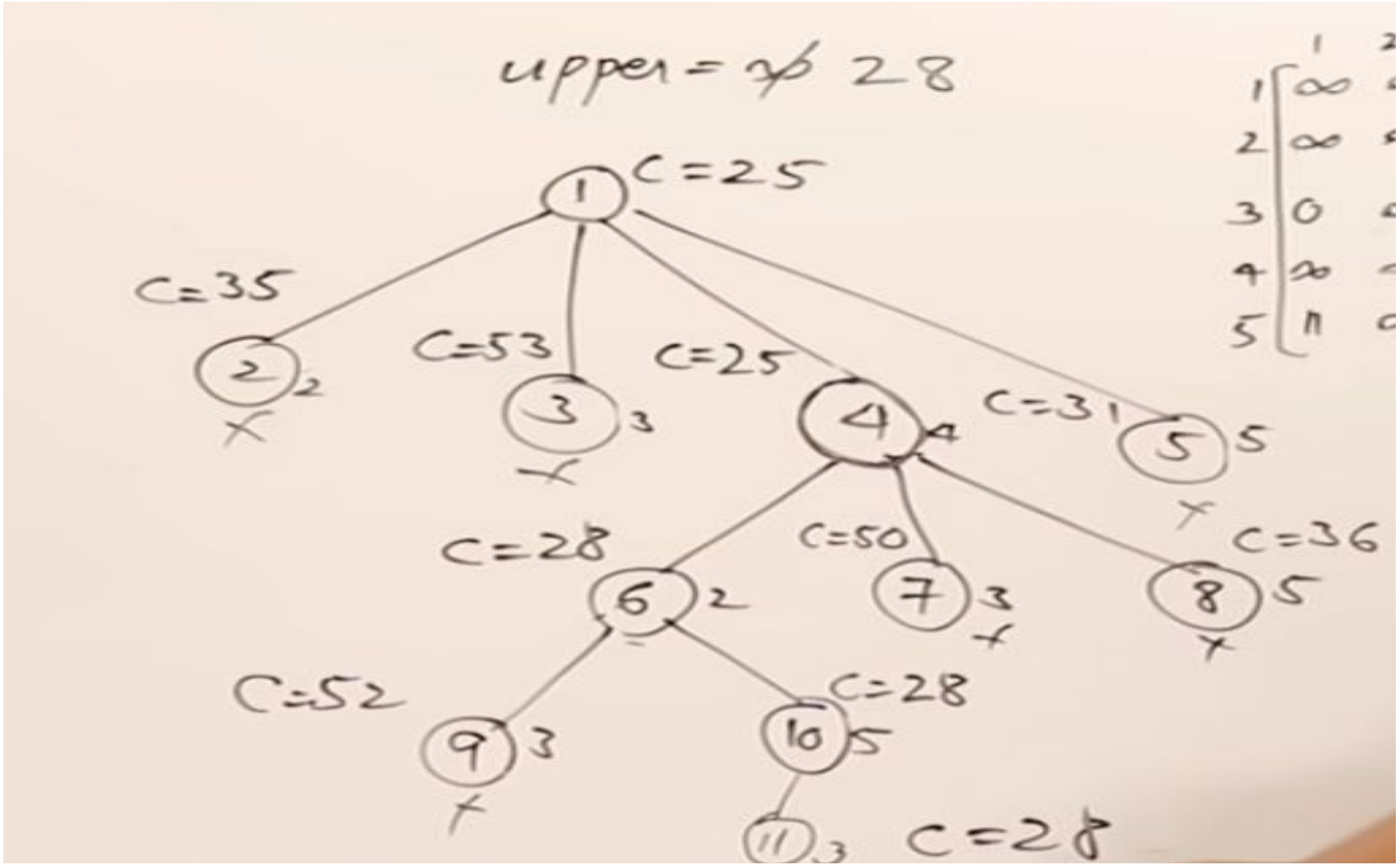
reduced cost = 25

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

(9)

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

(10)


upper = ∞ 28
C=25
C=35
C=53
C=25
C=31
C=28
C=50
C=36
C=52
C=28
C=28

Step 1: Draw the cost matrix(Self loop Infinity)

	N_0	N_1	N_2	N_3	N_4
N_0	INF	20	30	10	11
N_1	15	INF	16	4	2
N_2	3	5	INF	2	4
N_3	19	6	18	INF	3
N_4	16	4	7	16	INF

Let's start from node N0,;lets consider N0 as our first live node.

First of all we will perform the row operations and to do the same, we need to subtract the minimum value in each row from each element in that row. The minimums and the row reduced matrix is shown below,

	N_0	N_1	N_2	N_3	N_4
N_0	INF	20	30	**10**	11
N_1	15	INF	16	4	**2**
N_2	3	5	INF	**2**	4
N_3	19	6	18	INF	**3**
N_4	16	**4**	7	16	INF

And after row reduction,

	N_0	N_1	N_2	N_3	N_4
N_0	INF	10	20	**0**	1
N_1	13	INF	14	2	**0**
N_2	1	3	INF	**0**	2
N_3	16	3	15	INF	**0**
N_4	12	**0**	3	13	INF

Now we need to perform the column operation in the same manner and we will get a final matrix as below,

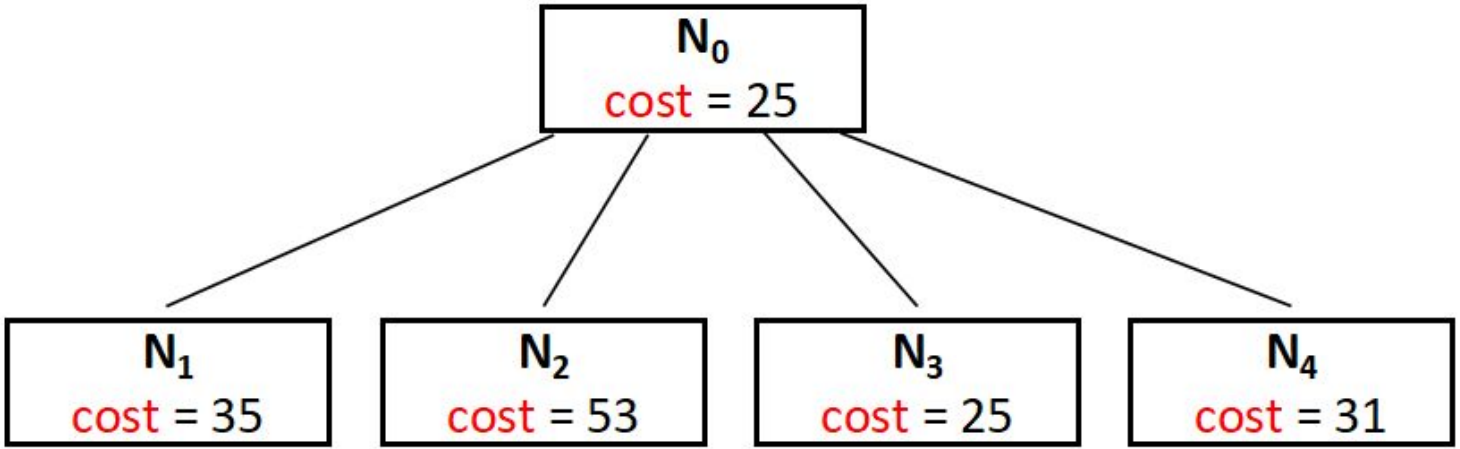
	N_0	N_1	N_2	N_3	N_4
N_0	INF	10	17	0	1
N_1	12	INF	11	2	0
N_2	0	3	INF	0	2
N_3	15	3	12	INF	0
N_4	11	0	0	13	INF

- The total cost at any node is calculated by making a sum of all reductions, hence the cost for the node N0 can be depicted as,
- cost = 10 + 2 + 2 + 3 + 4 + 1 + 3 = 25
- Now as we have founded the cost for the root node, it's the time for the expansion, and to do so, we have four more nodes namely, N1,N2,N3,N4.
- Let's add the edge from N0 to N1 in our search space. To do so, we need to set outgoing routes for city N0 to INF as well as all incoming routes to city N1 to INF.Also we will set the (N0,N1) position of the cost matrix as INF. By doing so, we are just focusing on the cost of the node N1, that's an E-node. In an easier note, we have just forgotten that the graph has a N0 node, but we are focusing on something that the graph has been started from the N1 node. Following the transformation we have something as below,

	N_0	N_1	N_2	N_3	N_4
N_0	INF	INF	INF	INF	INF
N_1	INF	INF	11	2	0
N_2	0	INF	INF	0	2
N_3	15	INF	12	INF	0
N_4	11	INF	0	13	INF

- Now it's the time to repeat the lower bound finding(row-reduction and the column-reduction) part that we have performed earlier as well, following which we will get that the matrix is already in reduced form, i.e. all rows and all columns have zero value.

- Now it's the time to calculate the cost.
- cost = cost of node N0 + cost of the (N0,N1) position(before setting INF) + lower bound of the path starting at N1 = 25 + 10 + 0 = 35
- Hence the cost for the E-Node N1 is 35.
- In a similar way we can find that,
- For N2 the cost is 53,
- For N3 the cost is 25,
- And for N4 the cost is 31.

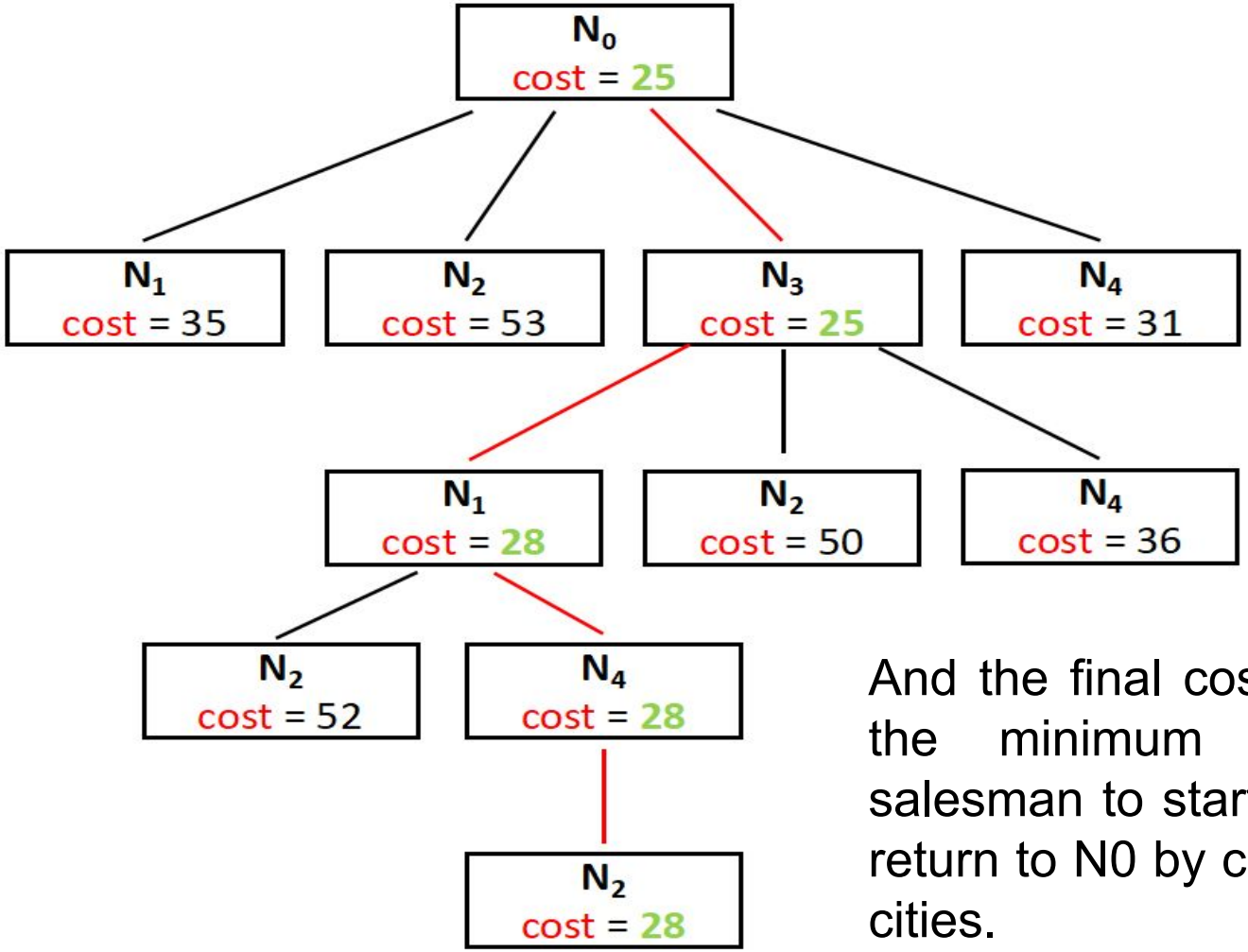

N_0
cost = 25
N_1
cost = 35
N_2
cost = 53
N_3
cost = 25
N_4
cost = 31

How to get towards the optimal solution ?

To get the optimal solution, we have to choose the next live node from the list of the extended nodes (E-nodes) and repeat the same procedure of extending the tree further by calculating the lower bound.

Here in the partially built space search tree above we have the Node N3 as the node having the minimum cost and we can consider it as the live node and extend the tree further. This step is repeated until unless we find a dead node and can not extend the tree further. The cost of the dead node (leaf node) will be the answer.

The entire space search tree can be drawn as follow,



And the final cost is **28**, that's the minimum cost for a salesman to start from N0 and return to N0 by covering all the cities.

15 PUZZLE PROBLEM

Branch and Bound

- The search for an answer node can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node.
- It is similar to backtracking technique but uses BFS-like search.
- There are basically three types of nodes involved in Branch and Bound

1. **Live node** is a node that has been generated but whose children have not yet been generated.
2. **E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3. **Dead node** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

15-Puzzle

- The 15 puzzle problem is invented by sam loyd in 1878.
- In this problem there are 15 tiles, which are numbered from 0 – 15.
- The objective of this problem is to transform the arrangement of tiles from initial arrangement to a goal arrangement.
- The initial and goal arrangement is shown by following figure.

1	2	3	4
5	6		8
9	10	7	11
12	13	14	15

Initial arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal arrangement

15-Puzzle Continue....

- There is always an empty slot in the initial arrangement as shown in fig.
- The legal moves are the moves in which the tiles adjacent to ES are moved to either left, right, up or down.
- Each move creates a new arrangement in a tile.
- These arrangements are called as states of the puzzle.
- The state space tree for 15 puzzle is very large because there can be 16! Different arrangements.
- In state space tree, the nodes are numbered as per the level.

1	2	3	4
5	6		8
9	10	7	11
12	13	14	15

Fig: Initial arrangement

15-Puzzle Continue....

- Each next move is generated based on empty slot positions.
- Edges are label according to the direction in which the empty space moves.
- The root node becomes the E – node.
- We can decide which node to become an E – node based on estimation formula.
- C (x) = f (x) + G(x)
- Where, f (x) = length of path from root to node x.
- G (x)= Estimated shortest path length from x to goal node
- = i.e. number of non-blank tiles which are not in their goal position for node x.
- C (x) = the lower bound cost of node x.

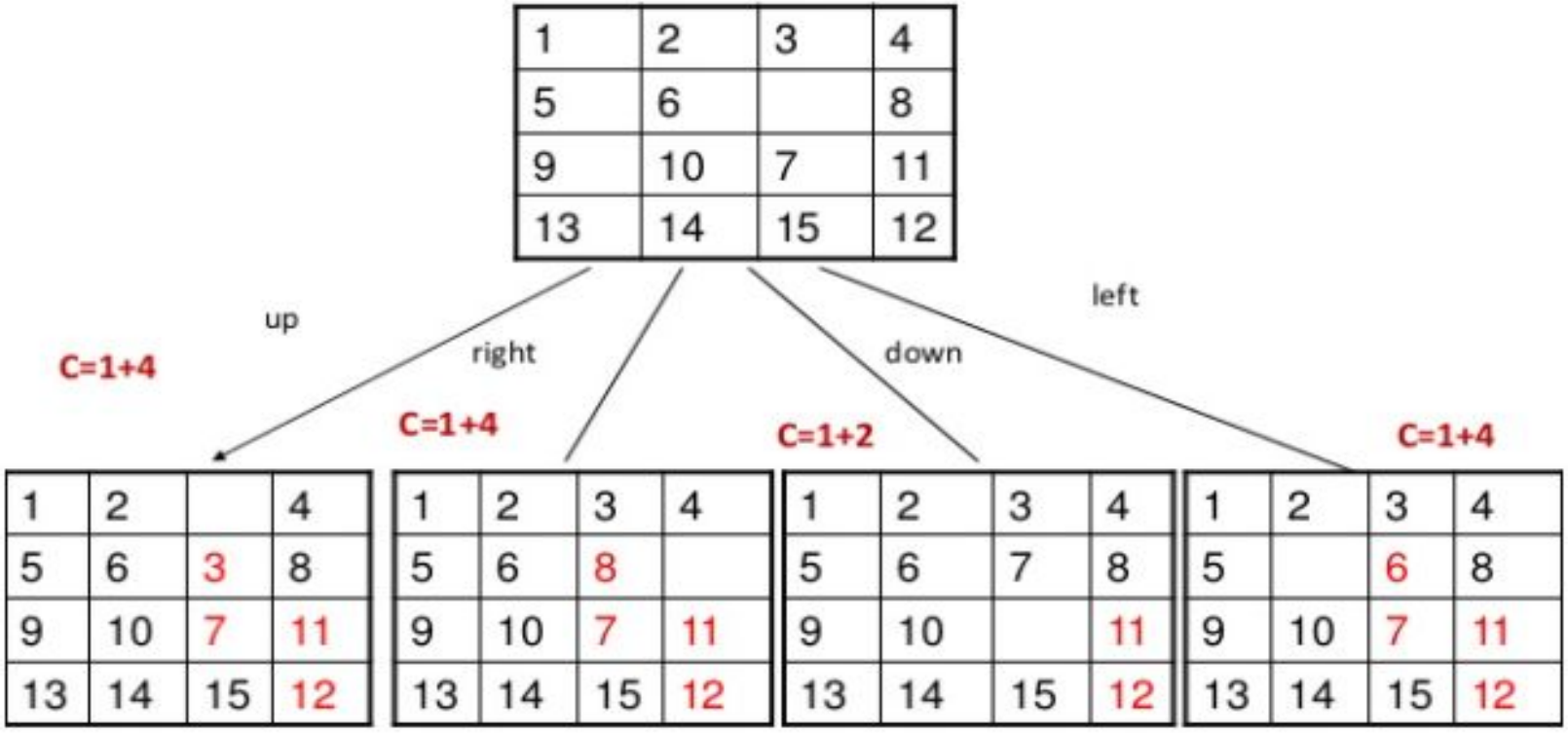
Example 1

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Initial arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Goal arrangement

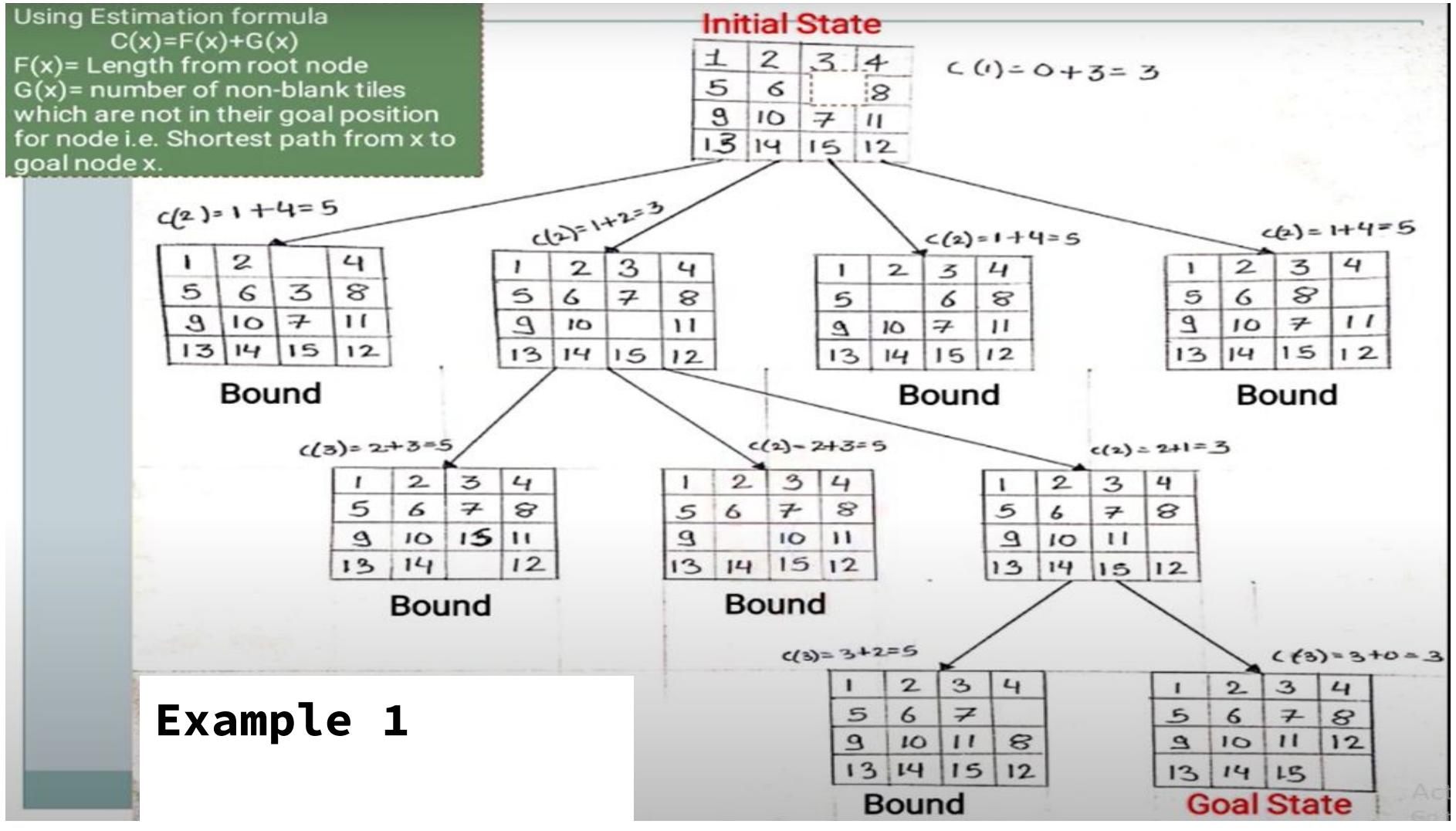

1 2 3 4
5 6 8
9 10 7 11
13 14 15 12
up
C=1+4
right
C=1+4
down
C=1+2
left
C=1+4
1 2 3 4
5 6 3 8
9 10 7 11
13 14 15 12
1 2 3 4
5 6 8
9 10 7 11
13 14 15 12
1 2 3 4
5 6 7 8
9 10 11
13 14 15 12
1 2 3 4
5 6 8
9 10 7 11
13 14 15 12

Using Estimation formula

$C(x)=F(x)+G(x)$

F(x)= Length from root node

G(x)= number of non-blank tiles which are not in their goal position for node i.e. Shortest path from x to goal node x.



Example 1

What are the differences between 'Backtracking' and 'Branch & Bound' Algorithm techniques?

Backtracking	Branch-and-Bound
[1] It is used to find all possible solutions available to the problem.	[1] It is used to solve optimization problem.
[2] It traverse tree by DFS (Depth First Search).	[2] It may traverse the tree in any manner, DFS or BFS.
[3] It realizes that it has made a bad choice & undoes the last choice by backing up.	[3] It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.
[4] It search the state space tree until it found a solution.	[4] It completely searches the state space tree to get optimal solution.
[5] It involves feasibility function	[5] It involves bounding function.

Thank You