

Basic Traversal and Search Techniques

A Binary Search Tree (BST) is a special type of binary tree in which the nodes are arranged in a specific order to allow efficient searching, insertion, and deletion operations. In a Binary Search Tree, every node contains a key value, and the nodes are organized according to the following properties:

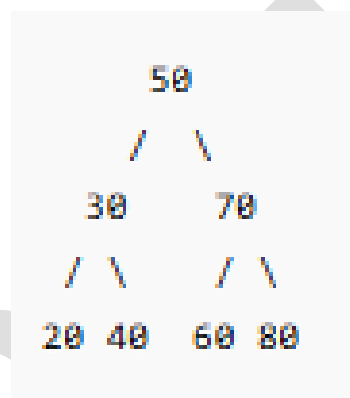
- All nodes in the left subtree of a node contain values smaller than the node's value.
- All nodes in the right subtree contain values greater than the node's value.
- Both left and right subtrees are themselves Binary Search Trees.

Because of this ordered structure, searching operations in BST become faster compared to ordinary binary trees.

For example, consider the following values inserted into a BST:

50, 30, 70, 20, 40, 60, 80

The tree is formed as:



In this tree:

- values smaller than 50 are placed on the left side,
- values greater than 50 are placed on the right side.

The Binary Search Tree supports three important basic operations:

1. Searching: Searching in BST begins from the root node. If the required key matches the root, the search is successful. If the key is smaller, the search moves to the left subtree; otherwise, it moves to the right subtree. This process continues until the element is found or the tree becomes empty. The searching operation is efficient because at each step half of the remaining tree is eliminated.

2. Insertion: Insertion in BST is performed by comparing the new value with the root node. If the value is smaller, it is inserted into the left subtree; otherwise, it is inserted into the right subtree. The process continues recursively until the correct empty position is found.

3. Deletion: Deletion is more complex in BST and occurs in three cases:

Case 1: The node is a leaf node.

Simply remove the node.

Case 2: The node has one child.

Replace the node with its child.

Case 3: The node has two children.

Replace the node with its in order successor or in order predecessor.

Algorithm for Searching in BST

```

Search(root, key)
{
    if root == NULL or root->data == key
        return root

    if key < root->data
        return Search(root->left, key)

    else
        return Search(root->right, key)
}

```

Time Complexity

The time complexity of BST operations depends on the height of the tree.

Operation	Average Case	Worst Case
Search	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(n)$

The worst case occurs when the tree becomes skewed.

Advantages of Binary Search Tree

- Searching is faster compared to ordinary binary trees.
- Insertion and deletion operations are efficient.
- Inorder traversal gives sorted data.
- Dynamic memory allocation is possible.
- Suitable for implementing dictionaries and symbol tables.

Limitations of Binary Search Tree

- Performance decreases if the tree becomes unbalanced.
- Worst-case complexity becomes linear.
- Requires extra memory for pointers.
- Complex deletion operation compared to arrays.

Applications of Binary Search Tree

Binary Search Trees are widely used in:

- database indexing,
- file organization,
- searching applications,
- compiler symbol tables,
- dictionary implementation,
- and expression evaluation.

Techniques for Binary Trees

A binary tree is a hierarchical data structure in which each node has at most two children called the left child and the right child. Binary trees are widely used in searching, sorting, expression evaluation, and hierarchical data representation.

The main techniques associated with binary trees are traversal techniques. Traversal is the process of visiting each node of the tree exactly once in a systematic order. The three important traversal techniques are:

- Inorder Traversal
- Preorder Traversal
- Postorder Traversal

Inorder Traversal

In inorder traversal, the nodes are visited in the order:

Left → Root → Right

This traversal first visits the left subtree, then processes the root node, and finally visits the right subtree.

The algorithm for inorder traversal is:

```
Inorder(x)
{
    if x ≠ NULL
    {
        Inorder(left(x))
        print x
        Inorder(right(x))
    }
}
```

Inorder traversal of a Binary Search Tree gives the elements in sorted order.

Preorder Traversal

In preorder traversal, the nodes are visited in the order:

Root → Left → Right

The root node is processed first, followed by the left subtree and then the right subtree.

The algorithm for preorder traversal is:

```
Preorder(x)
{
    if x ≠ NULL
    {
        print x
        Preorder(left(x))
        Preorder(right(x))
    }
}
```

Preorder traversal is commonly used for creating copies of trees and expression tree evaluation.

Postorder Traversal

In postorder traversal, the nodes are visited in the order:

Left → Right → Root

The left subtree and right subtree are processed first, and the root node is visited last.

The algorithm for postorder traversal is:

```
Postorder(x)
{
```

```

if x ≠ NULL
{
    Postorder(left(x))
    Postorder(right(x))
    print x
}

```

Postorder traversal is mainly used in deleting trees and postfix expression evaluation.

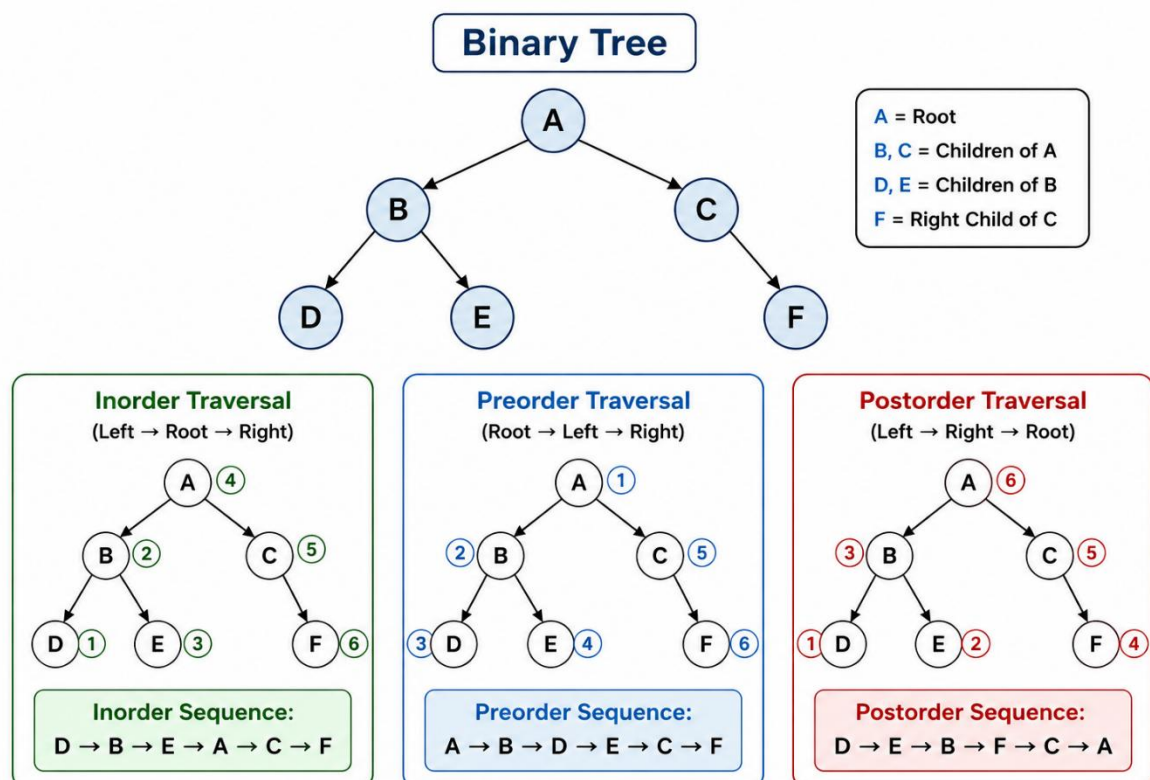
Comparison of Traversal Techniques

Traversal Technique	Visiting Order
Inorder	Left → Root → Right
Preorder	Root → Left → Right
Postorder	Left → Right → Root

Applications of Binary Tree Techniques

Binary tree traversal techniques are widely used in:

- Expression evaluation
- Compiler design
- Searching and sorting
- File system organization
- Arithmetic expression conversion
- Hierarchical data representation



Techniques for Graphs

A graph is a non-linear data structure consisting of a set of vertices and edges. Vertices represent entities or objects, while edges represent the relationship or connection between those vertices. Graphs are one of the most important data structures in computer science and are widely used in networking, transportation systems, social media applications, web searching, and artificial intelligence.

Graph traversal techniques are methods used to systematically visit all vertices and edges of a graph. The two fundamental graph traversal techniques are:

- Depth First Search (DFS)
- Breadth First Search (BFS)

These techniques help in exploring graphs efficiently and form the basis for many advanced graph algorithms.

Depth First Search (DFS)

Depth First Search is a traversal technique in which the exploration moves as deep as possible along a branch before backtracking. Starting from a source vertex, DFS visits one adjacent vertex and continues visiting deeper vertices until no unvisited adjacent vertex remains. Then it backtracks to explore other possible paths.

DFS is usually implemented using recursion or stack data structure. In recursive implementation, the system stack automatically stores the vertices during traversal.

The basic working of DFS is:

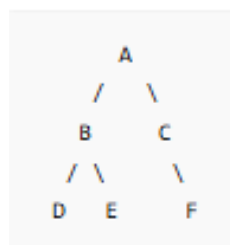
1. Visit the starting vertex.
2. Mark it as visited.
3. Visit one unvisited adjacent vertex.
4. Continue deeper recursively.
5. Backtrack when no adjacent vertex remains unvisited.

The algorithm for DFS

```
DFS(v)
{
    mark v as visited
    print v

    for each adjacent vertex u
    {
        if u is not visited
            DFS(u)
    }
}
```

Consider the graph:



If DFS starts from vertex A, one possible traversal is:

$$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F$$

DFS first completely explores the subtree rooted at B before moving toward C.

The time complexity of DFS is:

$$O(V + E)$$

where:

- V = number of vertices
- E = number of edges

DFS requires less memory compared to BFS because only one path is stored at a time.

DFS has many important applications such as:

- Cycle detection
- Topological sorting
- Connected component analysis
- Maze solving
- Path finding
- Strongly connected component detection

DFS is especially useful when solutions are located deep inside the graph structure.

Breadth First Search (BFS)

Breadth First Search is a graph traversal technique in which vertices are visited level by level. BFS first visits all neighboring vertices of a node before moving to the next level of vertices.

BFS uses a queue data structure to maintain the order of traversal. The queue ensures that vertices are processed in the same order in which they are discovered.

The working of BFS is:

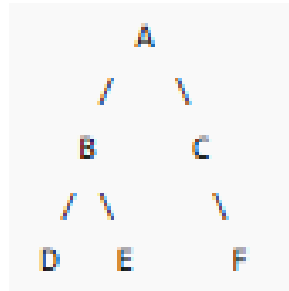
1. Start from a source vertex.
2. Mark it as visited.
3. Insert it into the queue.
4. Remove a vertex from the queue.
5. Visit all unvisited adjacent vertices.
6. Insert newly discovered vertices into the queue.
7. Repeat until the queue becomes empty.

The algorithm for BFS

```

BFS(v)
{
    mark v as visited
    enqueue(v)
    while queue is not empty
    {
        x = dequeue()
        print x
        for each adjacent vertex u
        {
            if u is not visited
            {
                mark u as visited
                enqueue(u)
            }
        }
    }
}

```

Using the same graph:

If BFS starts from vertex A, the traversal becomes:

$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F$$

BFS explores all vertices at one level before moving deeper.

The time complexity of BFS is also:

$$O(V + E)$$

BFS generally requires more memory because all neighboring vertices at each level are stored in the queue.

BFS is widely used in:

- Shortest path problems in unweighted graphs
- Social networking systems
- Broadcasting systems
- Web crawlers
- Peer-to-peer networks
- Level-order traversal.

One important property of BFS is that it guarantees the shortest path in unweighted graphs.

Spanning Tree using Graph Traversal

Both DFS and BFS can generate spanning trees.

A spanning tree is a subgraph that:

- Includes all vertices
- Remains connected
- Contains no cycles

During traversal:

- The discovery edges form the spanning tree.

DFS spanning trees are generally deep, whereas BFS spanning trees are broader and level-wise.

Connected Components

A connected component is a subgraph in which every pair of vertices is connected through some path.

DFS and BFS can efficiently identify connected components by:

- Starting traversal from unvisited vertices
- Marking all reachable vertices
- Repeating the process for remaining unvisited vertices

Cycle Detection

DFS is commonly used for cycle detection in graphs.

If during DFS traversal:

- A visited vertex is encountered again through a different path
- Then a cycle exists in the graph

Cycle detection is important in:

- Deadlock prevention
- Compiler design
- And network analysis

Applications of Graph Techniques

Graph traversal techniques are extensively used in many real-world applications.

In computer networks, BFS and DFS help in routing and connectivity analysis. In social media platforms, BFS is used to determine friend suggestions and shortest connections between users. Search engines use graph traversal for web crawling and indexing webpages. In artificial intelligence and robotics, graphs help in path planning and navigation.

DFS is widely used in:

- Topological sorting
- Puzzle solving
- Dependency resolution

BFS is widely used in:

- Shortest path finding
- Broadcasting systems
- Recommendation systems

Difference Between DFS and BFS

Depth First Search (DFS) and Breadth First Search (BFS) are the two fundamental graph traversal techniques used to visit all vertices and edges of a graph systematically. Although both techniques are used for graph traversal, they differ in their working method, data structures used, memory requirements, and applications.

Depth First Search (DFS)	Breadth First Search (BFS)
DFS explores a graph by moving as deep as possible along a branch before backtracking.	BFS explores a graph level by level by visiting all neighboring vertices first.
DFS uses a stack data structure or recursion for traversal.	BFS uses a queue data structure for traversal.
In DFS, vertices are visited depth-wise.	In BFS, vertices are visited breadth-wise or level-wise.
DFS may not always give the shortest path between vertices.	BFS guarantees the shortest path in an unweighted graph.

Depth First Search (DFS)	Breadth First Search (BFS)
DFS requires less memory because only one path is stored at a time.	BFS requires more memory because all neighboring vertices must be stored in the queue.
Backtracking is an important feature of DFS.	BFS does not require backtracking.
DFS is suitable for problems involving topological sorting, cycle detection, and maze solving.	BFS is suitable for shortest path problems, network broadcasting, and social networking applications.
DFS traversal can go very deep before exploring neighboring vertices.	BFS explores all vertices at the current level before moving deeper.
DFS is generally implemented recursively.	BFS is generally implemented iteratively using a queue.
The time complexity of DFS is $O(V + E)$.	The time complexity of BFS is also $O(V + E)$.

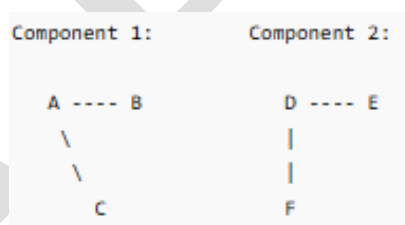
Connected Components

A connected component is an important concept in graph theory. In an undirected graph, a connected component is a subgraph in which every pair of vertices is connected by at least one path. In other words, all vertices within the same connected component are reachable from one another.

If a graph consists of only one connected component, then the graph is said to be a connected graph. If some vertices cannot be reached from others, the graph is called a disconnected graph.

Connected components are identified using graph traversal techniques such as Depth First Search (DFS) and Breadth First Search (BFS). The traversal starts from an unvisited vertex and marks all reachable vertices as part of the same component. The process is repeated until all vertices in the graph are visited.

Consider the following graph:



In this graph:

- A, B, and C form one connected component,
- D, E, and F form another connected component.

Thus, the graph contains two connected components.

Connected components are widely used in:

- Social network analysis
- Communication networks
- Clustering problems
- Image processing
- Network connectivity analysis.

The time complexity for finding connected components using DFS or BFS is:

$$O(V + E)$$

where:

- V = number of vertices,
- E = number of edges.

Connected components help determine whether a graph is fully connected and assist in analyzing the structure of large networks.

Spanning Trees

A spanning tree is a subgraph of a connected graph that contains all the vertices of the graph and forms a tree structure without cycles. A spanning tree connects all vertices using the minimum possible number of edges.

If a graph contains:

$$V$$

vertices, then every spanning tree contains exactly:

$$V - 1$$

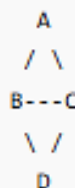
edges.

A graph can have multiple spanning trees depending on the selection of edges.

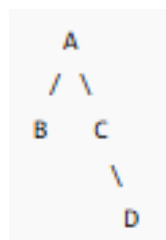
The spanning tree preserves graph connectivity while removing unnecessary cycles. Since it does not contain cycles, there exists exactly one path between any pair of vertices in the spanning tree.

Consider the graph:

Original Graph:



One possible spanning tree is:



The spanning tree:

- Contains all vertices
- Remains connected
- Has no cycles

Spanning trees can be generated using graph traversal techniques such as DFS and BFS. The discovery edges obtained during traversal form the spanning tree.

Spanning trees are extremely important in computer science and networking. They are used in:

- Network design
- Routing
- Broadcasting

- Electrical networks
- Minimum cable connection problems

One important type of spanning tree is the Minimum Spanning Tree (MST), where the total edge weight is minimized.

The important properties of spanning trees are:

1. A spanning tree contains all vertices.
2. It contains exactly $V - 1$ edges.
3. It does not contain cycles.
4. Removing any edge disconnects the tree.
5. Adding any extra edge creates a cycle.

The time complexity for generating spanning trees using DFS or BFS is:

$$O(V + E)$$

Thus, connected components and spanning trees are fundamental concepts in graph theory and are widely used in traversal, connectivity analysis, and network optimization problems.

Backtracking

Introduction

Backtracking is an important algorithmic technique used for solving combinational and constraint satisfaction problems. It systematically searches for a solution by trying different possibilities one by one. If a chosen solution path does not lead to a valid solution, the algorithm abandons that path and returns to the previous step to try another alternative. This process is called backtracking.

Backtracking is mainly used for problems where:

- Multiple possible solutions exist
- All possible combinations must be explored
- Constraints must be satisfied

It is considered an improved form of brute force because unnecessary solution paths are rejected early, reducing computation.

Definition

Backtracking is a problem-solving technique that builds solutions step by step and removes choices whenever they fail to satisfy the required conditions.

In simple words:

“Try a solution, and if it fails, go back and try another solution.”

General Method of Backtracking

The general method of backtracking involves constructing a solution incrementally and checking whether the current partial solution satisfies the problem constraints.

If the partial solution is valid:

- Continue building the solution

If the partial solution becomes invalid:

- Abandon that path
- Return to the previous step
- Try another possibility

Thus, backtracking performs a depth-first systematic search of the solution space.

Working Principle of Backtracking

The backtracking method follows these steps:

1. Start with an empty solution.
2. Add one component to the solution.
3. Check whether the current partial solution is valid.
4. If valid, continue further.
5. If invalid, remove the last component.
6. Try another alternative.
7. Repeat until a complete solution is found.

State Space Tree

Backtracking problems are generally represented using a State Space Tree.

- Each node represents a partial solution.
- Branches represent possible choices.
- Leaf nodes represent either valid solutions or dead ends.

The algorithm explores the tree using Depth First Search (DFS).

General Algorithm

```
Backtrack(k)
{
    if solution found
        print solution

    else
    {
        for each candidate x
        {
            if x is valid
            {
                include x in solution

                Backtrack(k+1)

                remove x from solution
            }
        }
    }
}
```

Characteristics of Backtracking

1. Incremental Solution Building: Solutions are constructed step by step.
2. Depth First Search: Explores one solution path completely before trying another.
3. Constraint Checking: Invalid solutions are rejected immediately.
4. Recursive Nature: Backtracking is usually implemented recursively.
5. Pruning: Unnecessary branches are eliminated early to reduce computation.

Example: N-Queens Problem

The N-Queens problem involves placing:

N

queens on an:

$N \times N$

chessboard such that:

- no two queens attack each other.

The algorithm:

- places queens row by row,
- checks validity,
- backtracks whenever conflict occurs.

Example: 4-Queens Solution

```

_ Q _ _
_ _ _ Q
Q _ _ _
_ _ Q _

```

No two queens attack each other horizontally, vertically, or diagonally.

Applications of Backtracking

- N-Queens Problem: Placing queens on a chessboard.
- Graph Coloring Problem: Assigning colors to vertices without adjacent same colors.
- Hamiltonian Cycle: Finding a cycle visiting every vertex exactly once.
- Sudoku Solver: Filling Sudoku grids satisfying constraints.
- Maze Solving: Finding path through a maze.
- Knight's Tour Problem: Moving a knight to visit every square exactly once.
- Subset Sum Problem: Finding subsets whose sum equals a target value.

Advantages of Backtracking

1. Systematic Search: Explores all possible solutions methodically.
2. Reduces Unnecessary Computation: Invalid paths are pruned early.
3. Suitable for Constraint Problems: Efficient for combinatorial optimization problems.
4. Simple Recursive Structure: Easy to understand conceptually.

Limitations of Backtracking

1. High Time Complexity: Worst-case complexity may still be exponential.
2. Large Search Space: Performance decreases for large problems.
3. Recursive Overhead: Requires stack memory for recursion.

Difference Between Backtracking and Dynamic Programming

Backtracking and Dynamic Programming are important algorithm design techniques used for solving complex computational problems. Although both techniques solve problems by dividing them into smaller subproblems, their approach and working principles are different.

Backtracking	Dynamic Programming
Backtracking is a problem-solving technique that searches for solutions by trying different possibilities and abandoning invalid paths.	Dynamic Programming is an optimization technique that solves problems by storing solutions of smaller subproblems and reusing them.
It follows a trial-and-error approach.	It follows an optimization and storage approach.
Solutions are built step by step and checked for validity.	Problems are divided into overlapping subproblems and solved efficiently.
If a solution path becomes invalid, the algorithm backtracks to try another path.	Previously computed solutions are stored and reused to avoid recomputation.
Mainly used for constraint satisfaction and combinatorial problems.	Mainly used for optimization problems.
Does not store results of previously solved subproblems.	Stores results using memoization or tabulation.
Uses Depth First Search (DFS) approach.	Uses Bottom-Up or Top-Down computation methods.
May explore many unnecessary solution paths.	Avoids repeated computations and improves efficiency.
Time complexity is usually exponential.	Time complexity is generally lower compared to brute force methods.
Examples: N-Queens, Sudoku, Graph Coloring.	Examples: Knapsack, Floyd-Warshall, Matrix Chain Multiplication.

N-Queen Problem

The N-Queen Problem is one of the most important applications of the Backtracking technique. It is a classical combinational optimization problem in which:

$$N$$

queens must be placed on an:

$$N \times N$$

chessboard such that no two queens attack each other.

In chess, a queen can move:

- horizontally,
- vertically,
- and diagonally.

Therefore, while placing queens on the chessboard, no two queens should be present:

- in the same row,
- in the same column,
- or in the same diagonal.

The objective of the N-Queen problem is to determine all possible valid arrangements of queens on the chessboard.

Why Backtracking is Used

The N-Queen problem has a very large number of possible arrangements. Checking all arrangements using brute force would require enormous computation. Backtracking improves efficiency by rejecting invalid arrangements immediately without exploring unnecessary possibilities.

The algorithm constructs the solution incrementally. Queens are placed one row at a time. Whenever a queen placement causes conflict, the algorithm removes that queen and tries another position. This process of returning to a previous state and trying a different choice is called backtracking.

Thus, backtracking performs a systematic search of the solution space while avoiding invalid branches.

Rules for Safe Placement

Before placing a queen at a position:

$$row, column$$

the following conditions are checked:

1. Column Condition: No queen should already exist in the same column.

2. Left Diagonal Condition: No queen should exist in the upper left diagonal.

3. Right Diagonal Condition: No queen should exist in the upper right diagonal.

Representation of Solution Space

The N-Queen problem is generally represented using a State Space Tree.

- Each level of the tree represents a row of the chessboard.
- Each node represents a possible queen placement.
- Branches represent choices of columns.
- Invalid nodes are pruned immediately.

The traversal of the state space tree follows Depth First Search (DFS).

General Working Procedure

The algorithm begins by placing a queen in the first row. Then it moves to the second row and searches for a safe position. If a safe position is found, another queen is placed. This process continues until all queens are placed successfully.

If at any row no safe position is available, the algorithm backtracks to the previous row, removes the previously placed queen, and tries another possible position.

The process continues recursively until:

- all queens are placed successfully,
- or all possible arrangements are exhausted.

Algorithm

```
NQueen(row)
{
    if row > N
        print solution

    else
    {
        for col = 1 to N
        {
            if Safe(row,col)
            {
                place queen at(row,col)

                NQueen(row+1)

                remove queen from(row,col)
            }
        }
    }
}
```


Safe Function

The Safe function checks whether a queen can be placed at a particular position.

```
Safe(row,col)
{
    check column

    check left diagonal

    check right diagonal

    if no conflict
        return true
    else
        return false
}
```

Example: 4-Queen Problem

Place 4 queens on a:

4 × 4

chessboard.

One valid arrangement is:

```
_ Q _ _
_ _ _ Q
Q _ _ _
_ _ Q _
```

Where:

- Q represents a queen
- _ represents an empty square

Explanation of Solution

Queens are placed at:

- Row 1 → Column 2
- Row 2 → Column 4
- Row 3 → Column 1
- Row 4 → Column 3

Verification:

- No two queens are in the same row.
- No two queens are in the same column.
- No two queens are in the same diagonal.

Hence, this is a valid solution.

Another Valid Solution

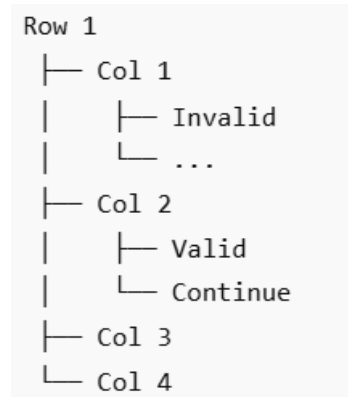
```

_ _ Q _
Q _ _ _
_ _ _ Q
_ Q _ _

```

Thus, the 4-Queen problem contains two possible valid solutions.

State Space Tree Illustration



The algorithm explores one branch deeply and backtracks whenever conflicts occur.

Time Complexity

The worst-case time complexity of the N-Queen problem is approximately:

$$O(N!)$$

because many possible arrangements are explored recursively.

However, Backtracking significantly reduces computation by pruning invalid paths early.

Space Complexity

The space complexity is:

$$O(N)$$

for recursion stack and storage of queen positions.

Advantages of Backtracking in N-Queen Problem

Backtracking provides a systematic way of exploring all possible arrangements. It avoids unnecessary computation by rejecting invalid solutions immediately. Compared to brute force methods, backtracking is much more efficient and practical for moderate values of:

$$N$$

The algorithm is also easy to implement recursively and clearly demonstrates the concept of recursion and state space search.

Limitations

Although backtracking reduces computation, the search space still grows rapidly for large values of:

$$N$$

For very large chessboards, execution time becomes high because the number of possible arrangements increases exponentially.

Recursive implementation may also require additional stack memory.

Applications of N-Queen Problem

The N-Queen problem is widely used in:

- Artificial Intelligence
- Combinational Optimization
- Scheduling Systems
- Puzzle Solving
- Constraint Satisfaction Problems

It is also an important educational example for studying:

- Recursion
- Backtracking
- Pruning
- State Space Trees

Sum of Subsets Problem

The Sum of Subsets Problem is an important combinatorial optimization problem solved using the Backtracking technique. The problem involves determining whether a subset of given numbers exists whose sum is equal to a specified target value.

The objective of the problem is to find all possible subsets whose elements add exactly to the required sum. Since many possible combinations may exist, Backtracking is used to systematically explore valid subsets while eliminating unnecessary combinations.

Definition

Given:

- a set of positive integers,
- and a target sum M ,

the Sum of Subsets Problem is to determine all subsets whose sum is exactly equal to:

$$M$$

Example

Suppose the given set is:

$$\{5, 10, 12, 13, 15, 18\}$$

and the target sum is:

$$30$$

Possible valid subsets are:

$$\{5, 10, 15\}$$

and

$$\{12, 18\}$$

because:

$$5 + 10 + 15 = 30$$

and

$$12 + 18 = 30$$

Why Backtracking is Used

In the Sum of Subsets Problem, many subsets can be formed from a given set. Checking all possible subsets using brute force requires large computation.

Backtracking improves efficiency by:

- Constructing subsets incrementally
- Checking partial sums
- And abandoning branches that cannot produce the required sum

Thus, unnecessary computations are avoided.

Working Principle

The algorithm considers elements one by one.

For each element, there are two choices:

- Include the element in the subset.
- Exclude the element from the subset.

At every stage:

- the current subset sum is calculated,
- and checked against the target sum.

If:

- the sum becomes equal to the target,
- the subset is printed.

If:

- the sum exceeds the target,
- or no valid solution is possible, the algorithm backtracks.

State Space Tree

The Sum of Subsets Problem is represented using a State Space Tree.

- Each level represents an element.
- Left branch → include element.
- Right branch → exclude element.
- Leaf nodes represent solutions or dead ends.

The tree is explored using Depth First Search.

Algorithm

```

SumOfSubsets(k, currentSum, remainingSum)
{
    if currentSum == target
        print subset
    else
    {
        if currentSum + nextElement <= target
        {
            include nextElement
            SumOfSubsets(k+1,
                currentSum + nextElement,
                remainingSum - nextElement)
        }

        if currentSum + remainingSum - nextElement >= target
        {
            exclude nextElement
            SumOfSubsets(k+1,
                currentSum,
                remainingSum - nextElement)
        }
    }
}

```

Example Problem

Given set:

{5,10,12,13,15,18}

Target sum:

30

Solution Process**Step 1**

Start with empty subset.

Current sum:

0

Step 2

Include:

5

Current sum:

5

Step 3

Include:

10

Current sum:

15

Step 4

Include:

15

Current sum:

30

Solution found:

{5,10,15}

Another Solution

Include:

12

and:

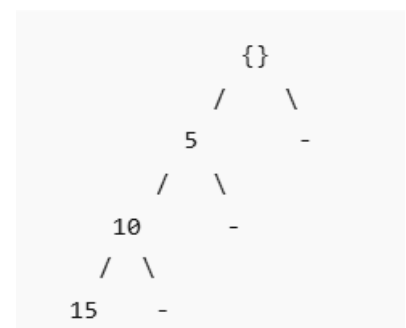
18

Current sum:

 $12 + 18 = 30$

Solution:

{12,18}

State Space Tree Illusion:

Bounding Function

Backtracking uses bounding conditions to reduce unnecessary computation.

A branch is abandoned if:

1. Current sum exceeds target sum.
2. Remaining elements cannot produce target sum.

This process is called pruning.

Time Complexity

Worst-case time complexity is:

$$O(2^n)$$

because every element may either:

- be included,
- or excluded.

However, pruning significantly reduces practical computation.

Space Complexity

The space complexity is:

$$O(n)$$

due to recursion stack and subset storage.

Advantages

1. Eliminates unnecessary subsets using pruning.
2. More efficient than brute force approach.
3. Systematically explores solution space.
4. Suitable for combinational optimization problems.

Limitations

1. Worst-case complexity remains exponential.
2. Performance decreases for large datasets.
3. Recursive calls increase memory usage.

Applications of Sum of Subsets Problem

The Sum of Subsets Problem is used in:

- Cryptography
- Resource allocation
- Scheduling
- Partition problems

- Combinational optimization.

It is also important in:

- Artificial intelligence
- Decision making
- Subset analysis problems

Hamiltonian Cycles:

A **Hamiltonian cycle** is a fundamental concept in graph theory and Design and Analysis of Algorithms (DAA). It refers to a cycle in a graph that visits every vertex exactly once and returns to the starting vertex, forming a closed loop.

Definition

A Hamiltonian cycle in a graph $G = (V, E)$ is a cycle of the form:

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \dots \rightarrow v_n \rightarrow v_1$$

such that:

- Every vertex $v_i \in V$ is visited exactly once
- The starting and ending vertex are the same
- Each consecutive pair of vertices is connected by an edge in E

Hamiltonian Path vs Hamiltonian Cycle

- A **Hamiltonian Path** visits every vertex exactly once but does not return to the starting vertex.
- A **Hamiltonian Cycle** visits every vertex exactly once and returns to the starting vertex.
- Thus, a cycle is a closed form of a Hamiltonian path.

Important Properties

- It is vertex-based (focuses on vertices, not edges).
- Not all connected graphs contain a Hamiltonian cycle.
- A complete graph K_n ($n \geq 3$) always contains a Hamiltonian cycle.
- The existence of a Hamiltonian cycle is not guaranteed even if the graph is connected.

Theorems (Sufficient Conditions)

- **Dirac's Theorem:** If every vertex in a graph with n vertices has degree $\geq n/2$, then the graph contains a Hamiltonian cycle.
- **Ore's Theorem:** If for every pair of non-adjacent vertices u, v ,

$$\deg(u) + \deg(v) \geq n$$

then the graph contains a Hamiltonian cycle.

Algorithm (Backtracking Approach)

The Hamiltonian cycle problem is typically solved using backtracking:

1. Start from any vertex.
2. Add a vertex to the path if:
 - It is adjacent to the previous vertex.
 - It has not been visited before.
3. Continue this process recursively.
4. If all vertices are included and the last vertex is connected to the first vertex, a Hamiltonian cycle is found.
5. If no valid vertex can be added, backtrack and try another path.

Complexity

- The problem is **NP-Complete**.
- Time complexity is **$O(n!)$** in the worst case due to exhaustive search.
- No polynomial-time algorithm is known for general graphs.

Applications

- Travelling Salesman Problem (TSP)
- Network routing and communication design
- Circuit board layout design
- Robot motion planning
- Bioinformatics (DNA sequencing)

Consider the following undirected graph with vertices **A, B, C, D, E** and edges:

- A–B, A–C, A–E
- B–C, B–D
- C–D, C–E
- D–E

Using the **Backtracking method**, construct the state space tree and determine whether a **Hamiltonian cycle** exists in the given graph. If it exists, write all possible Hamiltonian cycles.

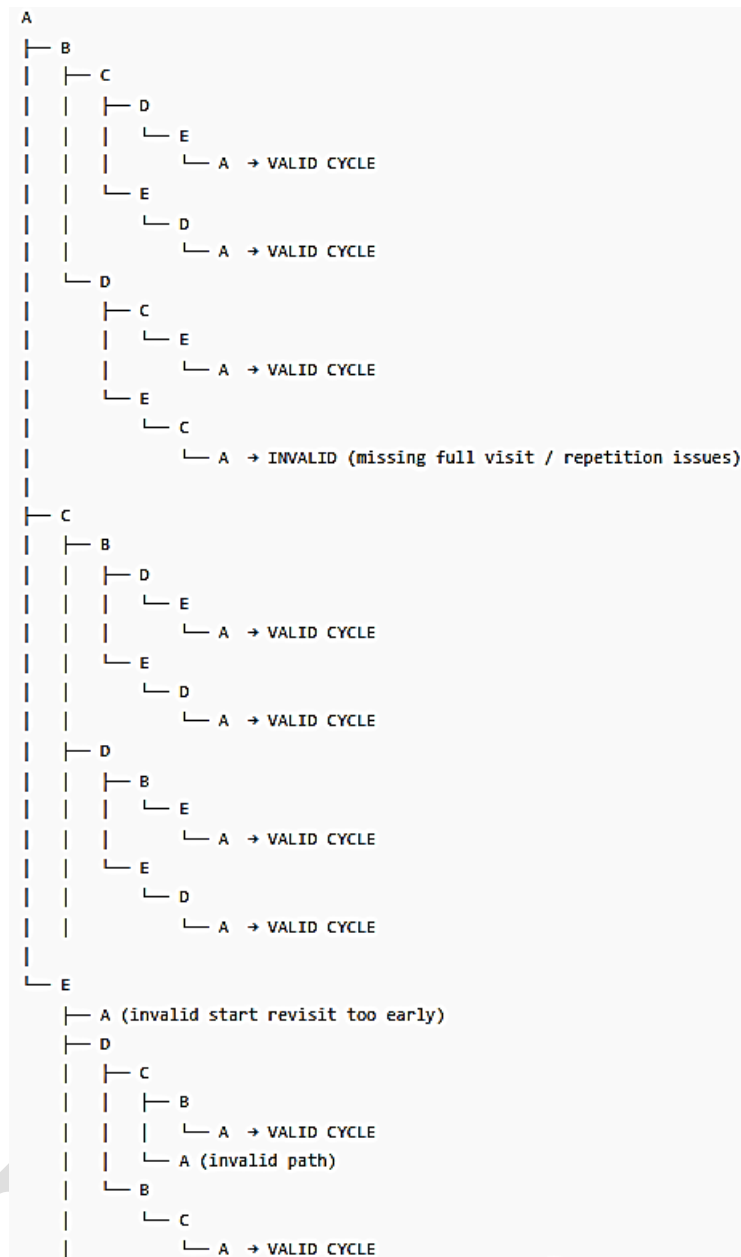
Step 1: Fix starting vertex

Start from **A** (standard for backtracking problems).

We build paths by adding only:

- unvisited vertices
- adjacent vertices only

State Space Tree (Backtracking)



Step 3: Valid Hamiltonian Cycles Found

From the tree, multiple valid cycles exist:

All Hamiltonian cycles:

1. $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$
2. $A \rightarrow B \rightarrow D \rightarrow C \rightarrow E \rightarrow A$
3. $A \rightarrow C \rightarrow B \rightarrow D \rightarrow E \rightarrow A$
4. $A \rightarrow C \rightarrow D \rightarrow B \rightarrow E \rightarrow A$
5. $A \rightarrow E \rightarrow D \rightarrow C \rightarrow B \rightarrow A$
6. $A \rightarrow E \rightarrow C \rightarrow D \rightarrow B \rightarrow A$

Step 4: Conclusion

- A Hamiltonian cycle **exists** in the given graph.
- The graph is highly connected, allowing **multiple Hamiltonian cycles**.
- Backtracking confirms all valid permutations that satisfy constraints.