

1. Implement DFS, BFS for 8-puzzle problem

Step 1: What is the 8-puzzle?

The 8-puzzle is a sliding puzzle consisting of:

- A 3×3 grid
- 8 numbered tiles (1–8) and one empty space (also called the blank or 0).

It looks like this:

1 2 3

4 _ 6

7 5 8

Where _ represents the blank space.

Step 2: Objective of the puzzle

Given an **initial configuration**, the goal is to move the tiles **one at a time** (sliding them into the blank space) until you reach the **goal state**:

1 2 3

4 5 6

7 8 _

Step 3: How can you move?

- You can slide a **tile adjacent to the blank** into the blank space.
- The blank can move:
 - o **up**
 - o **down**
 - o **left**
 - o **right**

For example:

Start: Move blank up:

1 2 3 1 2 3

4 _ 6 → _ 4 6

7 5 8 7 5 8

Step 4: State Space

Every **arrangement of tiles + blank** is a **state**.

- Total possible states = $9! = 362,880$
- But only **half of these (181,440)** are solvable because of the parity of inversions.

Step 5: Approach to solve the problem

You search for a sequence of moves from **initial state** → **goal state**.

Types of Search Algorithms:

Uninformed Search

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

Informed (Heuristic) Search

- A* algorithm
- Greedy Best First Search

Step 6: Cost of a move

- Typically, each move costs 1.
- Goal: Find the **shortest path (least number of moves)** from start to goal.

Step 7: Example of solving the puzzle

Initial State:

```
1 2 3
4 5 6
_ 7 8
```

Goal State:

```
1 2 3
4 5 6
7 8 _
```

Moves:

Move 7 left:

```
1 2 3
```

4 5 6

7 _ 8

Move 8 left:

1 2 3

4 5 6

7 8 _

Solved in **2 moves!**

Why is this problem important?

- Small but challenging.
- Demonstrates search strategies & heuristics.
- Foundation for more complex AI planning & robotics.

What a **search space tree** is — especially in the context of problems like the **8-puzzle**, pathfinding, or optimization problems.

What is a Search Space Tree?

A **search space tree** is a conceptual tree that represents:

- All **possible states** of a problem
- And all the **paths (or sequences of actions)** you can take from the **initial state** to reach the **goal state**.

It helps us systematically explore all possible solutions to a problem.

Key Components of a Search Space Tree:

Term	Meaning
------	---------

Root Node	Represents the initial state of the problem.
------------------	---

Child Nodes	States reachable by applying one action to the parent state.
--------------------	--

Edges	Actions or moves that transform one state into another.
--------------	---

Term	Meaning
------	---------

Leaf Nodes	States with no further moves OR the goal state.
-------------------	---

Path	Sequence of moves (edges) from the root to any node.
-------------	--

Why is it called a “tree”?

Because:

- Each state can lead to one or more next states.
- These states “branch out” like a tree.
- No cycles in this abstract tree (although the real state space might have cycles — we handle that separately in algorithms).

Example: Search space tree for a simple 8-puzzle

Let’s take a very simple **initial state**:

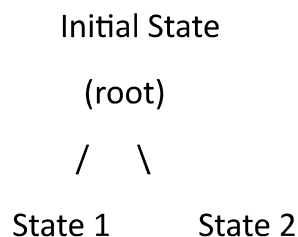
1 2 3

4 5 6

_ 7 8

- The **root node** is this initial state.
- From here, the blank (**_**) can move **right** or **down**, generating two children.
- Each child becomes a new node and can generate its own children.
- This continues until we find the **goal state**.

Visually:



/ \ / \

State 3 State 4 State 5 State 6

...

Each level of the tree represents one more **move** away from the root.

Search in this tree

- **Breadth-First Search (BFS)** explores the tree level by level.
- **Depth-First Search (DFS)** explores as deep as possible along a branch before backtracking.
- **A*** and other informed algorithms explore the most promising nodes first using heuristics.

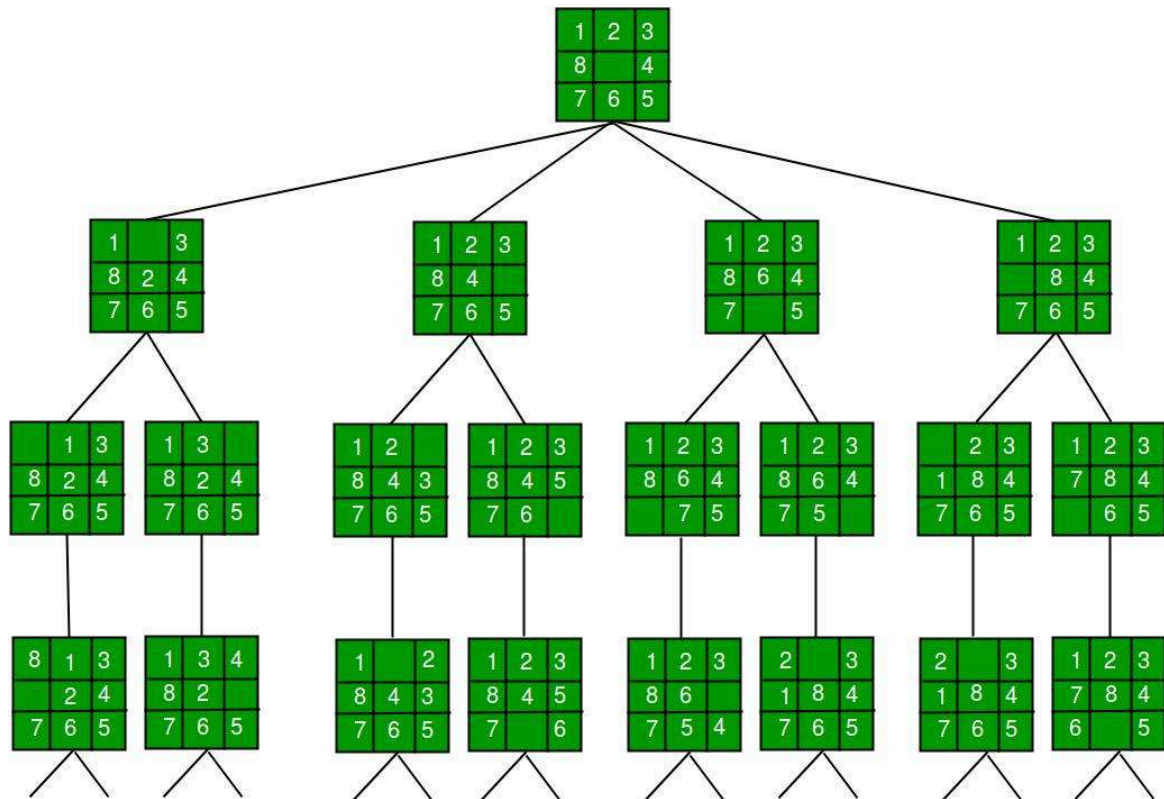
Benefits:

Helps us **visualize** the problem-solving process.

Ensures we don't miss any possible solution.

Makes it clear how states are connected.

Useful for designing and understanding search algorithms.



https://github.com/RajPShinde/8-Puzzle-BFS_Algorithm

<https://www.youtube.com/watch?v=POM4mmlctyo&authuser=0>

<https://www.javatpoint.com/8-puzzle-problem-in-python?authuser=0>

https://github.com/RajPShinde/8-Puzzle-BFS_Algorithm?authuser=0

<https://github.com/AlexP11223/EightPuzzleSolver?authuser=0>

<https://www.kaggle.com/code/muhammadrozy77/8-puzzle-problem-bfs-group-1?authuser=0>

```
#include <iostream>
```

```
#include <vector>
```

```

#include <queue>

#include <set>

using namespace std;


// Define the dimensions of the puzzle
#define N 3


// Structure to represent the state of the puzzle
struct PuzzleState {
    vector<vector<int>> board;

    int x, y;

    int depth;


    // Constructor
    PuzzleState(vector<vector<int>> b, int i, int j, int d) : board(b), x(i), y(j),
depth(d) {}

};


// Possible moves: Left, Right, Up, Down
int row[] = {0, 0, -1, 1};
int col[] = {-1, 1, 0, 0};


// Function to check if the current state is the goal state
bool isGoalState(vector<vector<int>>& board) {
    vector<vector<int>> goal = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};

    return board == goal;
}

```

```
// Function to check if a move is valid
```

```
bool isValid(int x, int y) {  
    return (x >= 0 && x < N && y >= 0 && y < N);  
}
```

```
// Function to print the puzzle board
```

```
void printBoard(vector<vector<int>>& board) {  
    for (auto& row : board) {  
        for (auto& num : row)  
            cout << num << " ";  
        cout << endl;  
    }  
    cout << "-----" << endl;  
}
```

```
// BFS function to solve the 8-puzzle problem
```

```
void solvePuzzleBFS(vector<vector<int>>& start, int x, int y) {  
    queue<PuzzleState> q;  
    set<vector<vector<int>>> visited;  
  
    // Enqueue initial state  
    q.push(PuzzleState(start, x, y, 0));  
    visited.insert(start);  
  
    while (!q.empty()) {
```



```
PuzzleState curr = q.front();
q.pop();

// Print the current board state
cout << "Depth: " << curr.depth << endl;
printBoard(curr.board);

// Check if goal state is reached
if (isGoalState(curr.board)) {
    cout << "Goal state reached at depth " << curr.depth << endl;
    return;
}

// Explore all possible moves
for (int i = 0; i < 4; i++) {
    int newX = curr.x + row[i];
    int newY = curr.y + col[i];

    if (isValid(newX, newY)) {
        vector<vector<int>> newBoard = curr.board;
        swap(newBoard[curr.x][curr.y], newBoard[newX][newY]);

        // If this state has not been visited before, push to queue
        if (visited.find(newBoard) == visited.end()) {
            visited.insert(newBoard);
            q.push(PuzzleState(newBoard, newX, newY, curr.depth + 1));
        }
    }
}
```

```

        }
    }
}

cout << "No solution found (BFS Brute Force reached depth limit)" << endl;
}

// Driver Code
int main() {
    vector<vector<int>> start = {{1, 2, 3}, {4, 0, 5}, {6, 7, 8}}; // Initial state
    int x = 1, y = 1;

    cout << "Initial State: " << endl;
    printBoard(start);

    solvePuzzleBFS(start, x, y);

    return 0;
}

```

Output

Initial State:

1 2 3

4 0 5

6 7 8

Depth: 0

1 2 3

4 0 5

6 7 8

Depth: 1

1 2 3

0 4 5

6 7 8

Depth: 1

1 2 3

4 5 0

6 7 8

Depth: 1

1 0 3

4 2 5

6 7 8

-----...

imitations of DFS and BFS in the 8-Puzzle Problem

- ***DFS:*** Can get stuck in deep, unproductive paths, leading to excessive memory usage and slow performance.
- ***BFS:*** Explores all nodes at the current depth level, making it inefficient as it does not prioritize promising paths.

Optimizing with Branch and Bound (B&B)

*Branch and Bound enhances search efficiency by using a **cost function** to guide exploration.*

- 1. **Intelligent Node Selection**: Prioritizes nodes **closer to the goal**, unlike DFS (blind) or BFS (equal priority).*
- 2. **Pruning**: Eliminates unpromising paths to save time and memory.*

Approach:

- Use a **priority queue** to store live nodes.
- Initialize the **cost function** for the root node.
- Expand the **least-cost** node first.
- Stop when a goal state is reached or when the queue is empty.

Types of Nodes in B&B:

- **Live Node**: Generated but not yet explored.
- **E-node (Expanding Node)**: Currently being expanded.
- **Dead Node**: No longer considered (either solution found or cost too high).