Vishwakarma Institute of Technology

Department of Computer Science Engineering (CS)

Name	Vaishnavi Vijay Arthamwar
Division	CS-A
Batch	B3
Roll No.	16
PRN No.	12220242
Department	Computer
Subject	Artificial Intelligence
Assignment No.	2

Assingment No. 2

Title: Implementation of Uninformed strategies

Uninformed Search:

Uninformed Search Algorithms These algorithms are given no information about the problem other than its definition. They are also called blind search algorithms. It means that the strategies have no additional information about states beyond that provided in the problem definition. distinguished by the order in which nodes are expanded. Although some of these algorithms can solve any solvable problem, none of them can do so efficiently. E.g., Depth First Search, Breadth First Search, Depth Limited Search, Iterative Deepening Depth First Search.

Key features of uninformed search algorithms:

- **Systematic exploration** -uninformed search algorithms explore the search space systematically, either by expanding all children of a node (e.g. BFS) or by exploring as deep as possible in a single path before backtracking (e.g. DFS).
- **No heuristics** - uninformed search algorithms do not use additional information, such as heuristics or cost estimates, to guide the search process.
- **Blind search** -uninformed search algorithms do not consider the cost of reaching the goal or the likelihood of finding a solution, leading to a blind search process.
- **Simple to implement** - uninformed search algorithms are often simple to implement and understand, making them a good starting point for more complex algorithms.
- **Inefficient in complex problems -** uninformed search algorithms can be inefficient in complex problems with large search spaces, leading to an exponential increase in the number of states explored.

8Puzzle Problem

An 8 puzzle is a simple game consisting of a 3 x 3 grid/matrix (containing 9 squares). One of the squares is empty. The object is to move squares around into different positions and have the numbers displayed in the "goal state".

initial state and goal state is given:

```
INITIAL STATE          GOAL STATE

 1 2 3                    1 2 3

 0 4 6        =>          4 5 6
```

7 5 8   7 8 0

We have been given a 3×3 board with 8 tiles or elements (every tile has one number from 1 to 8) and one empty space which we have denoted here with "0". The objective is to place the numbers on tiles to match the final goal state using the empty space. We can slide four adjacent (left, right, above, and below) tiles. We can move "0" to Left, Right, Above, and below only if it's possible.

As in the above situation, we cannot move 0 to left but we can move it to top, Right, Bottom. we can move in all directions only when 0 is in the center.

We need to slide 0 to all the possible places until we find out our result.

Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

How BFS works:

To implement this algorithm, we need to use a Queue data structure that follows the FIFO first in first out methodology. In BFS, one vertex is selected at a time and when it is visited and marked then its adjacent are visited and stored in the queue and the visited one, we can remove from queue if all the adjacent vertices are stored or explored.

In BFS we expand all possible moves from the current state before progressing to the next level, prioritizing solutions at shallower depths.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

T (b) = 1+b2+b3+.......+ bd= O (bd)

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is O(bd).

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

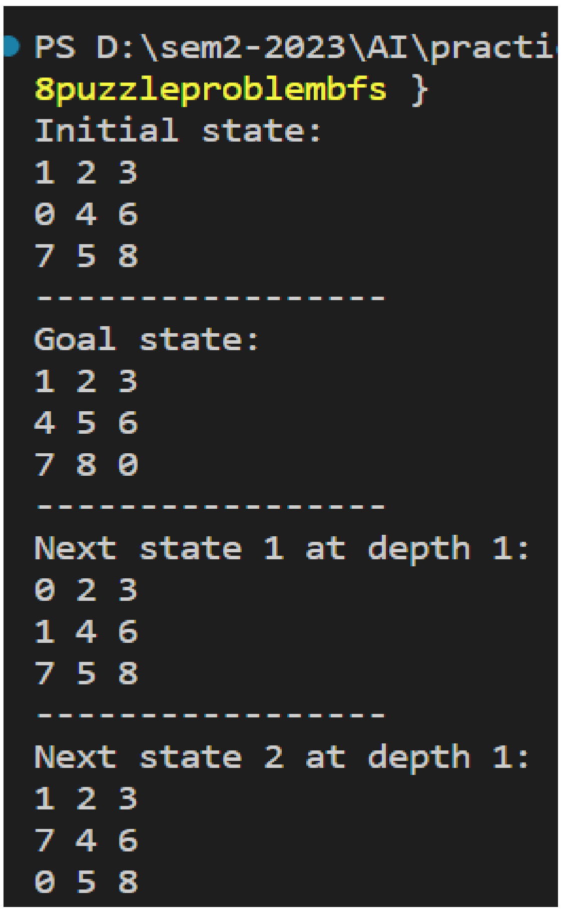
Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Code:

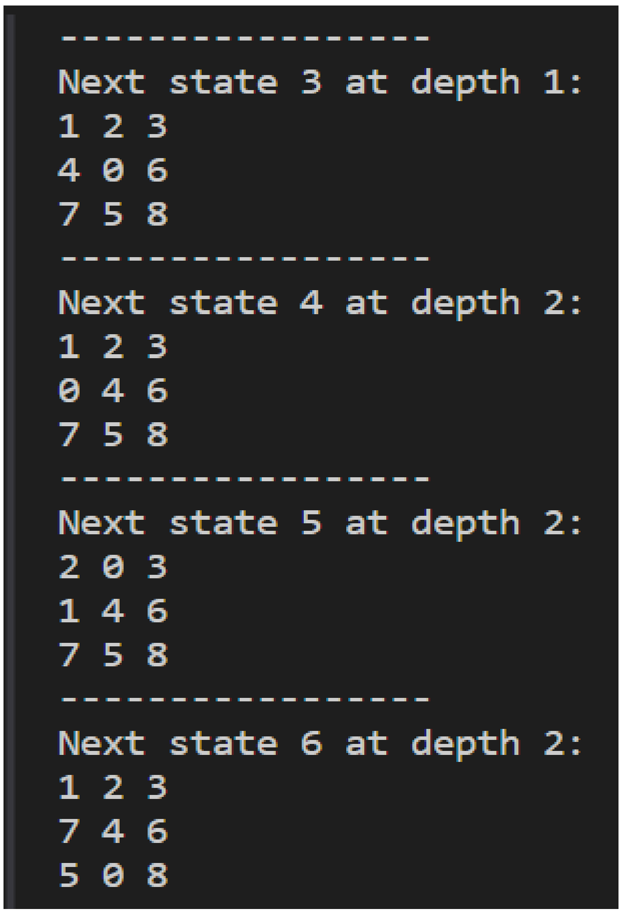
```cpp
#include <iostream>
#include <queue>
#include <set>
#include <vector>
using namespace std;

// Representation of a state of the puzzle
struct State {
    vector<vector<int>> board;
    int zeroRow, zeroCol;
    static const vector<vector<int>> goal;

    // Constructor to initialize the state
    State(vector<vector<int>>& b, int zRow, int zCol) : board(b),
zeroRow(zRow), zeroCol(zCol) {}

    // Function to check if the state is the goal state
    bool isGoal() const {
        return board == goal;
    }

    // Function to generate successor states
    vector<State> successors() const {
        static const int dr[] = {-1, 1, 0, 0}; // Up, Down, Left, Right
        static const int dc[] = {0, 0, -1, 1};
        vector<State> succ;
        for (int d = 0; d < 4; ++d) {
            int newRow = zeroRow + dr[d];
            int newCol = zeroCol + dc[d];
            if (newRow >= 0 && newRow < 3 && newCol >= 0 && newCol < 3) {
                vector<vector<int>> newBoard = board;
                swap(newBoard[zeroRow][zeroCol], newBoard[newRow][newCol]);
                succ.emplace_back(newBoard, newRow, newCol);
            }
        }
        return succ;
    }
```

```cpp
};

const vector<vector<int>> State::goal = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};

// Function to perform BFS to solve the puzzle
void solvePuzzle(State initialState) {
    queue<State> q;
    set<vector<vector<int>>> visited;
    q.push(initialState);

    while (!q.empty()) {
        State currState = q.front();
        q.pop();
        for (const State& nextState : currState.successors()) {
            if (visited.find(nextState.board) == visited.end()) {
                visited.insert(nextState.board);
                cout << "Next state:" << endl;
                for (const auto& row : nextState.board) {
                    for (int val : row) {
                        cout << val << " ";
                    }
                    cout << endl;
                }
                cout << "-----------------" << endl;
                if (nextState.isGoal()) {
                    cout << "Goal state reached!" << endl;
                    cout << "Goal state:" << endl;
                    for (const auto& row : nextState.board) {
                        for (int val : row) {
                            cout << val << " ";
                        }
                        cout << endl;
                    }
                    return;
                }
                q.push(nextState);
            }
        }
    }

    cout << "No solution found!" << endl;
}

int main() {
    vector<vector<int>> initialBoard = {{1, 2, 3}, {4, 0, 6}, {7, 5, 8}}; //
Initial state of the puzzle
    int zeroRow = 1, zeroCol = 1; // Position of the empty cell (zero)
    State initialState(initialBoard, zeroRow, zeroCol);
    cout << "Initial state:" << endl;
    for (const auto& row : initialBoard) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    cout << "-----------------" << endl;

    cout << "Goal state:" << endl;
    for (const auto& row : State::goal) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    cout << "-----------------" << endl;

    solvePuzzle(initialState);
    return 0;
}
```

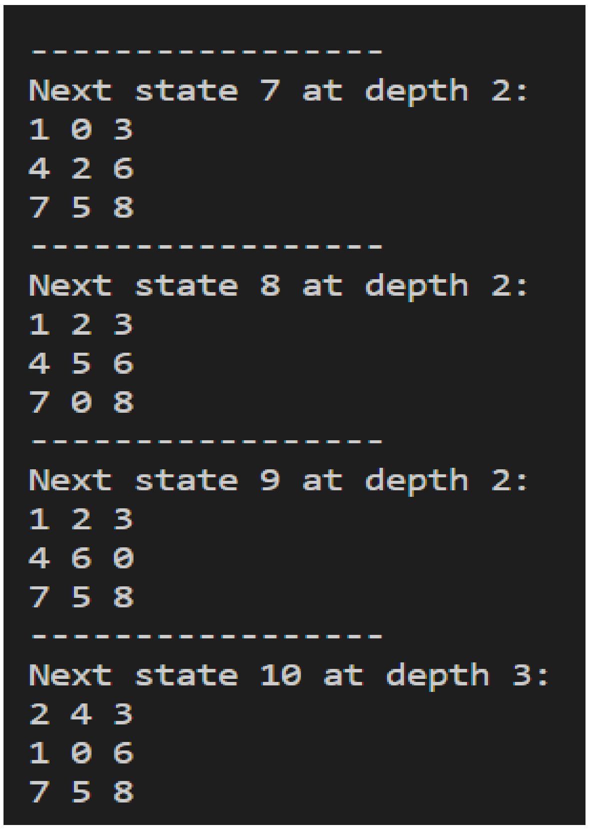
Output:



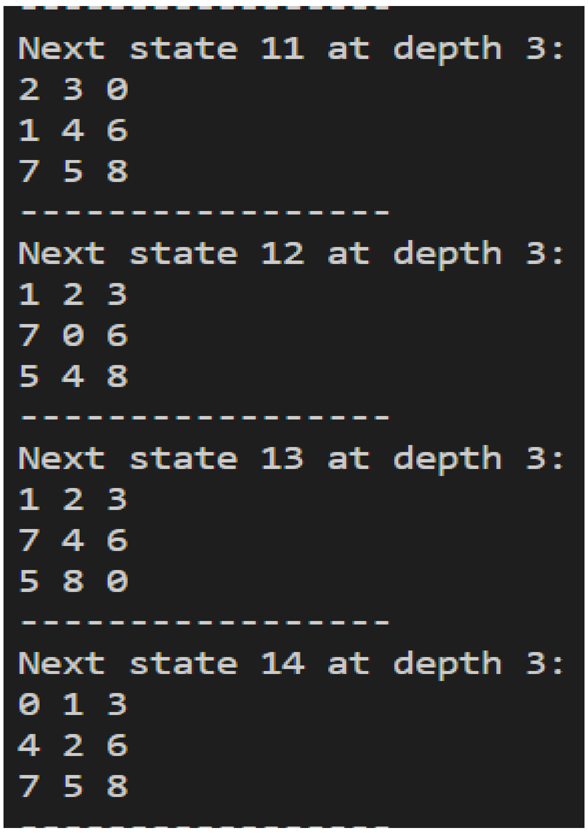
```
PS D:\sem2-2023\AI\practi
8puzzleproblembfs }
Initial state:
1 2 3
0 4 6
7 5 8
-----------------
Goal state:
1 2 3
4 5 6
7 8 0
-----------------
Next state 1 at depth 1:
0 2 3
1 4 6
7 5 8
-----------------
Next state 2 at depth 1:
1 2 3
7 4 6
0 5 8
```



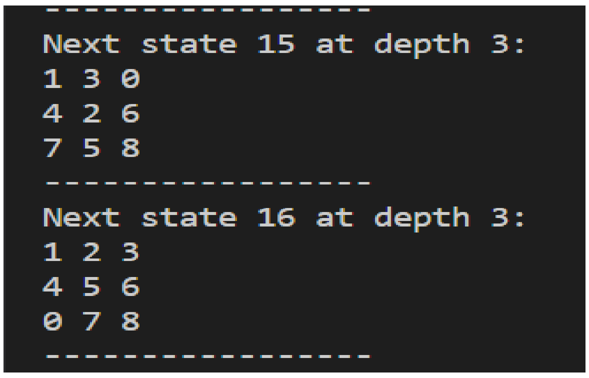
```
-----------------
Next state 3 at depth 1:
1 2 3
4 0 6
7 5 8
-----------------
Next state 4 at depth 2:
1 2 3
0 4 6
7 5 8
-----------------
Next state 5 at depth 2:
2 0 3
1 4 6
7 5 8
-----------------
Next state 6 at depth 2:
1 2 3
7 4 6
5 0 8
```



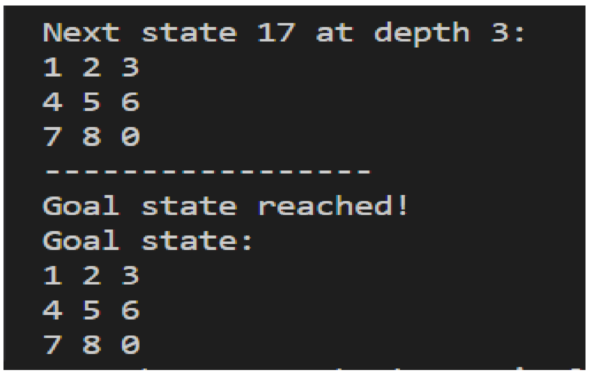
```
------------------
Next state 7 at depth 2:
1 0 3
4 2 6
7 5 8
------------------
Next state 8 at depth 2:
1 2 3
4 5 6
7 0 8
------------------
Next state 9 at depth 2:
1 2 3
4 6 0
7 5 8
------------------
Next state 10 at depth 3:
2 4 3
1 0 6
7 5 8
```



```
------------------
Next state 11 at depth 3:
2 3 0
1 4 6
7 5 8
------------------
Next state 12 at depth 3:
1 2 3
7 0 6
5 4 8
------------------
Next state 13 at depth 3:
1 2 3
7 4 6
5 8 0
------------------
Next state 14 at depth 3:
0 1 3
4 2 6
7 5 8
------------------
```

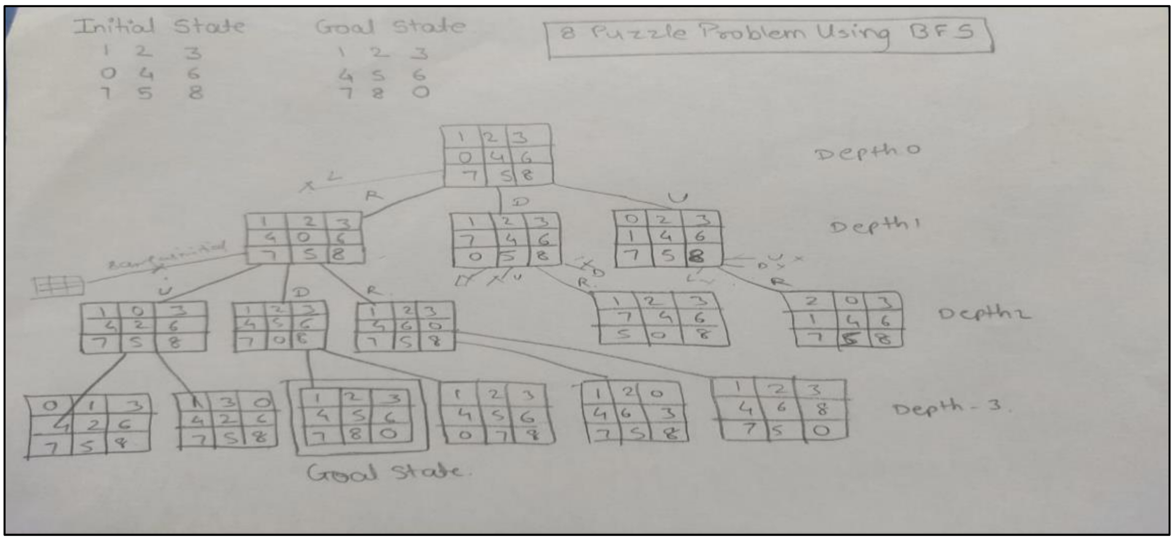


```
------------------
Next state 15 at depth 3:
1 3 0
4 2 6
7 5 8
------------------
Next state 16 at depth 3:
1 2 3
4 5 6
0 7 8
------------------
```



```
Next state 17 at depth 3:
1 2 3
4 5 6
7 8 0
------------------
Goal state reached!
Goal state:
1 2 3
4 5 6
7 8 0
```

Dry run:



Depth-first Search

- Depth-first search is a recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.
- Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Root node--->Left node ----> right node.

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

T(n)= 1+ n2+ n3 +.........+ nm=O(nm)

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is O(bm).

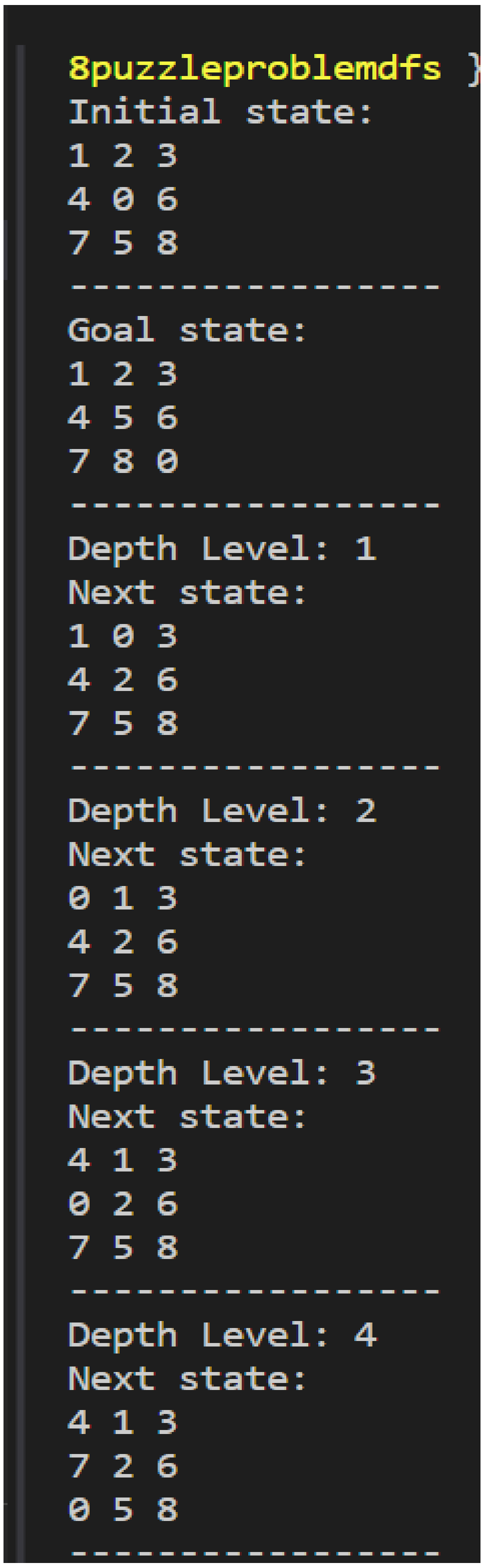
Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Code:

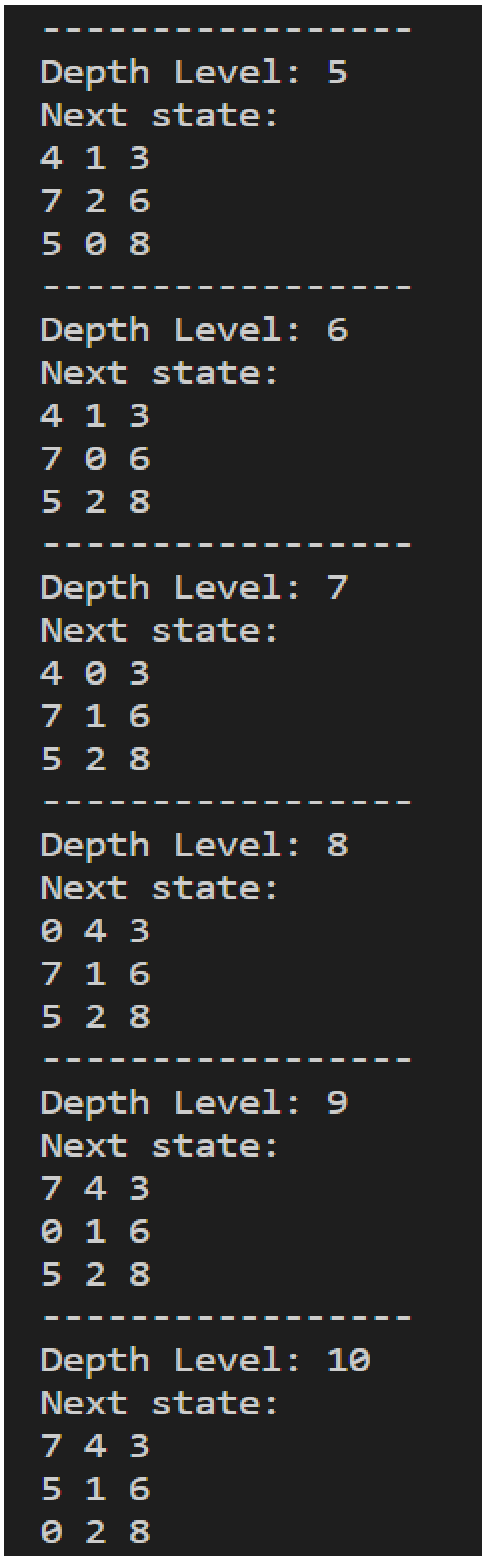
```
#include <iostream>
#include <set>
#include <vector>
using namespace std;
```

```cpp
// Representation of a state of the puzzle
struct State {
    vector<vector<int>> board;
    int zeroRow, zeroCol;
    static const vector<vector<int>> goal;

    // Constructor to initialize the state
    State(vector<vector<int>>& b, int zRow, int zCol) : board(b),
zeroRow(zRow), zeroCol(zCol) {}

    // Function to check if the state is the goal state
    bool isGoal() const {
        return board == goal;
    }

    // Function to generate successor states
    vector<State> successors() const {
        static const int dr[] = {-1, 1, 0, 0}; // Up, Down, Left, Right
        static const int dc[] = {0, 0, -1, 1};
        vector<State> succ;
        for (int d = 0; d < 4; ++d) {
            int newRow = zeroRow + dr[d];
            int newCol = zeroCol + dc[d];
            if (newRow >= 0 && newRow < 3 && newCol >= 0 && newCol < 3) {
                vector<vector<int>> newBoard = board;
                swap(newBoard[zeroRow][zeroCol], newBoard[newRow][newCol]);
                succ.emplace_back(newBoard, newRow, newCol);
            }
        }
        return succ;
    }
};

const vector<vector<int>> State::goal = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};

// Depth-First Search function
bool dfs(State currentState, set<vector<vector<int>>>& visited) {
    if (currentState.isGoal()) {
        cout << "Goal state reached!" << endl;
        cout << "Goal state:" << endl;
        for (const auto& row : currentState.board) {
            for (int val : row) {
                cout << val << " ";
            }
            cout << endl;
        }
        return true;
    }

    visited.insert(currentState.board);

    for (const State& nextState : currentState.successors()) {
        if (visited.find(nextState.board) == visited.end()) {
            cout << "Next state:" << endl;
            for (const auto& row : nextState.board) {
                for (int val : row) {
                    cout << val << " ";
                }
                cout << endl;
            }
            cout << "-----------------" << endl;
            if (dfs(nextState, visited))
                return true;
        }
    }

    return false;
}

int main() {
    vector<vector<int>> initialBoard = {{1, 2, 3}, {4, 0, 6}, {7, 5, 8}}; //
Initial state of the puzzle
    int zeroRow = 1, zeroCol = 1; // Position of the empty cell (zero)
    State initialState(initialBoard, zeroRow, zeroCol);
    cout << "Initial state:" << endl;
    for (const auto& row : initialBoard) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    cout << "-----------------" << endl;

    cout << "Goal state:" << endl;
    for (const auto& row : State::goal) {
        for (int val : row) {
            cout << val << " ";
        }
        cout << endl;
    }
    cout << "-----------------" << endl;

    set<vector<vector<int>>> visited;
    if (!dfs(initialState, visited))
        cout << "No solution found!" << endl;
    return 0;
}
```

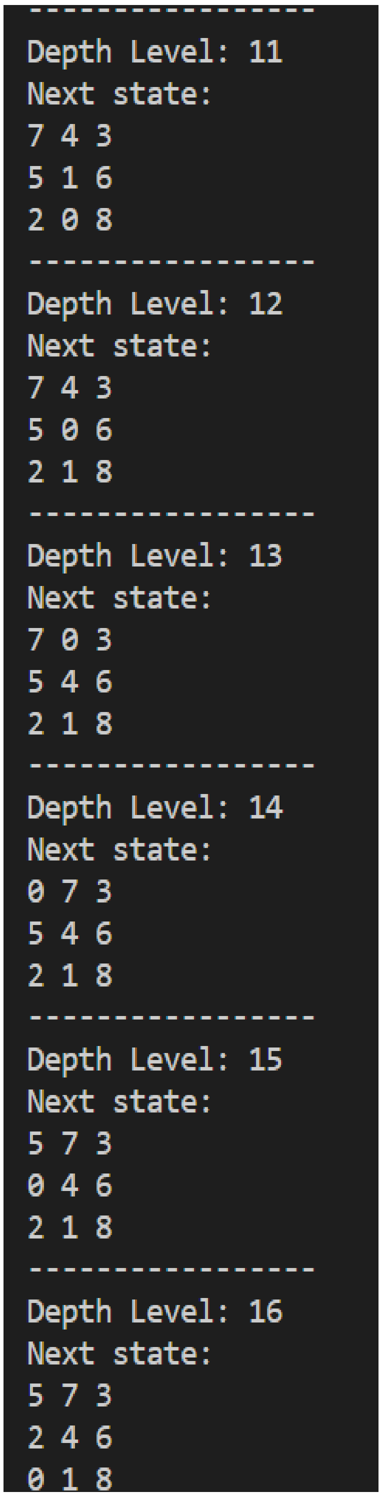
Output:



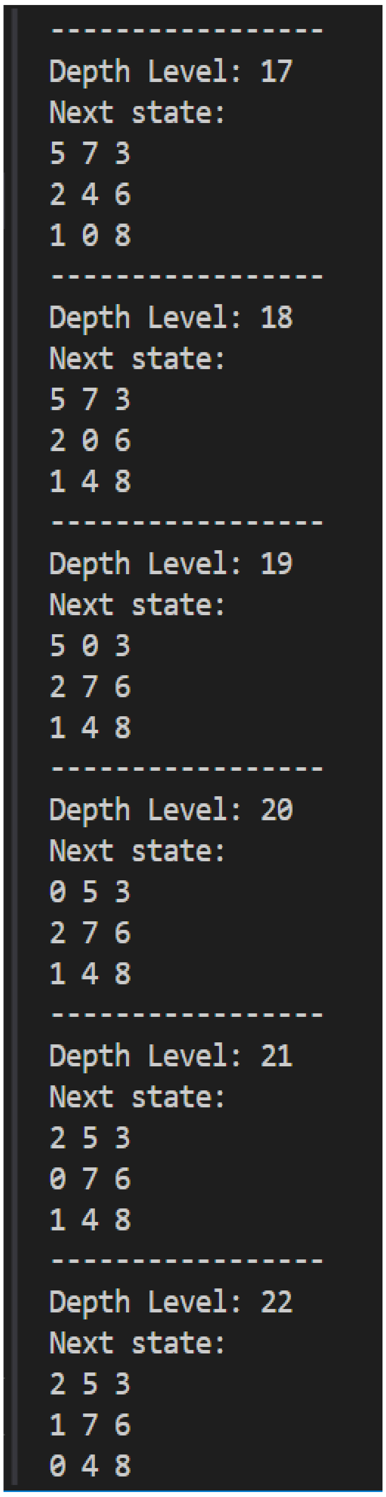
```
8puzzleproblemdfs }
Initial state:
1 2 3
4 0 6
7 5 8
-----------------
Goal state:
1 2 3
4 5 6
7 8 0
-----------------
Depth Level: 1
Next state:
1 0 3
4 2 6
7 5 8
-----------------
Depth Level: 2
Next state:
0 1 3
4 2 6
7 5 8
-----------------
Depth Level: 3
Next state:
4 1 3
0 2 6
7 5 8
-----------------
Depth Level: 4
Next state:
4 1 3
7 2 6
0 5 8
-----------------
```



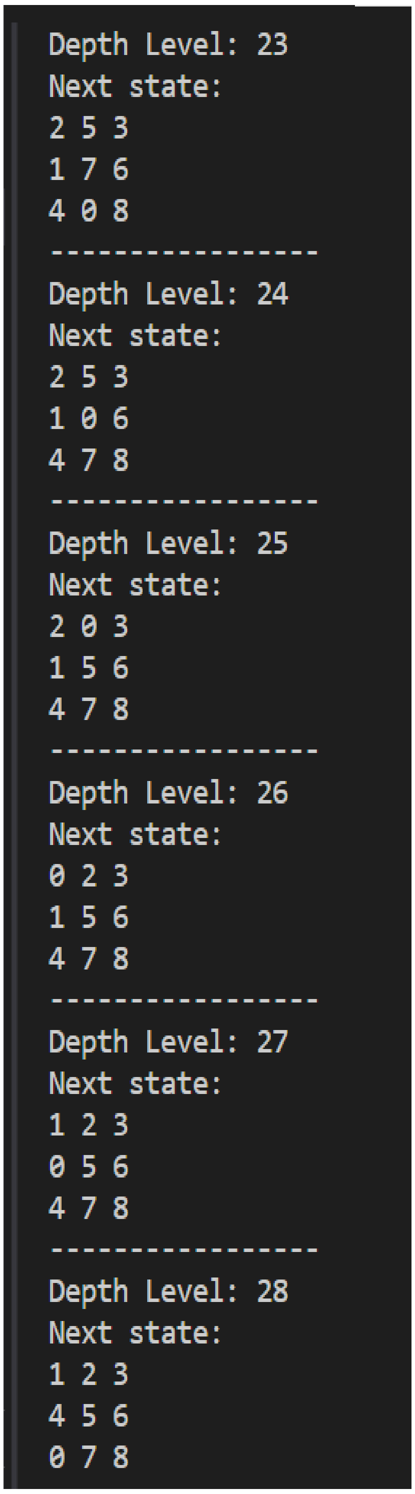
```
-----------------
Depth Level: 5
Next state:
4 1 3
7 2 6
5 0 8
-----------------
Depth Level: 6
Next state:
4 1 3
7 0 6
5 2 8
-----------------
Depth Level: 7
Next state:
4 0 3
7 1 6
5 2 8
-----------------
Depth Level: 8
Next state:
0 4 3
7 1 6
5 2 8
-----------------
Depth Level: 9
Next state:
7 4 3
0 1 6
5 2 8
-----------------
Depth Level: 10
Next state:
7 4 3
5 1 6
0 2 8
```



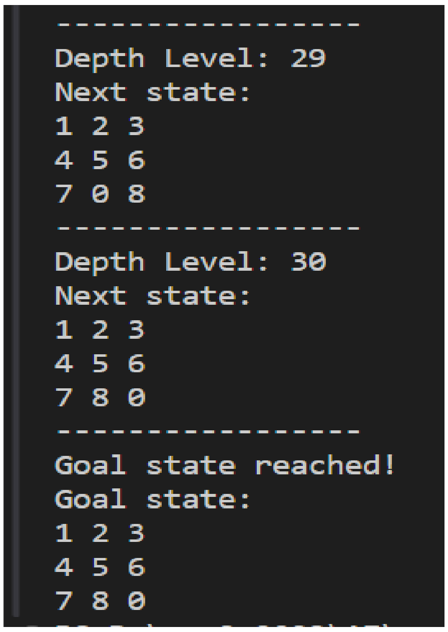
```
-----------------
Depth Level: 11
Next state:
7 4 3
5 1 6
2 0 8
-----------------
Depth Level: 12
Next state:
7 4 3
5 0 6
2 1 8
-----------------
Depth Level: 13
Next state:
7 0 3
5 4 6
2 1 8
-----------------
Depth Level: 14
Next state:
0 7 3
5 4 6
2 1 8
-----------------
Depth Level: 15
Next state:
5 7 3
0 4 6
2 1 8
-----------------
Depth Level: 16
Next state:
5 7 3
2 4 6
0 1 8
```



```
-----------------
Depth Level: 17
Next state:
5 7 3
2 4 6
1 0 8
-----------------
Depth Level: 18
Next state:
5 7 3
2 0 6
1 4 8
-----------------
Depth Level: 19
Next state:
5 0 3
2 7 6
1 4 8
-----------------
Depth Level: 20
Next state:
0 5 3
2 7 6
1 4 8
-----------------
Depth Level: 21
Next state:
2 5 3
0 7 6
1 4 8
-----------------
Depth Level: 22
Next state:
2 5 3
1 7 6
0 4 8
```

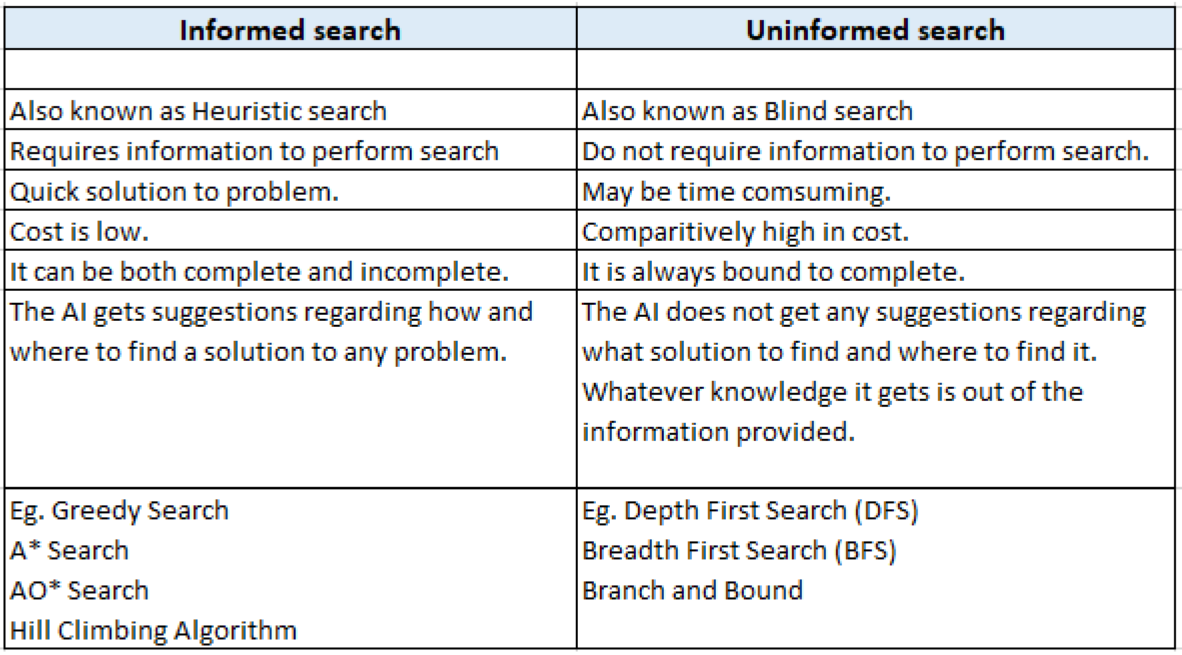


```
Depth Level: 23
Next state:
2 5 3
1 7 6
4 0 8
-----------------
Depth Level: 24
Next state:
2 5 3
1 0 6
4 7 8
-----------------
Depth Level: 25
Next state:
2 0 3
1 5 6
4 7 8
-----------------
Depth Level: 26
Next state:
0 2 3
1 5 6
4 7 8
-----------------
Depth Level: 27
Next state:
1 2 3
0 5 6
4 7 8
-----------------
Depth Level: 28
Next state:
1 2 3
4 5 6
0 7 8
```



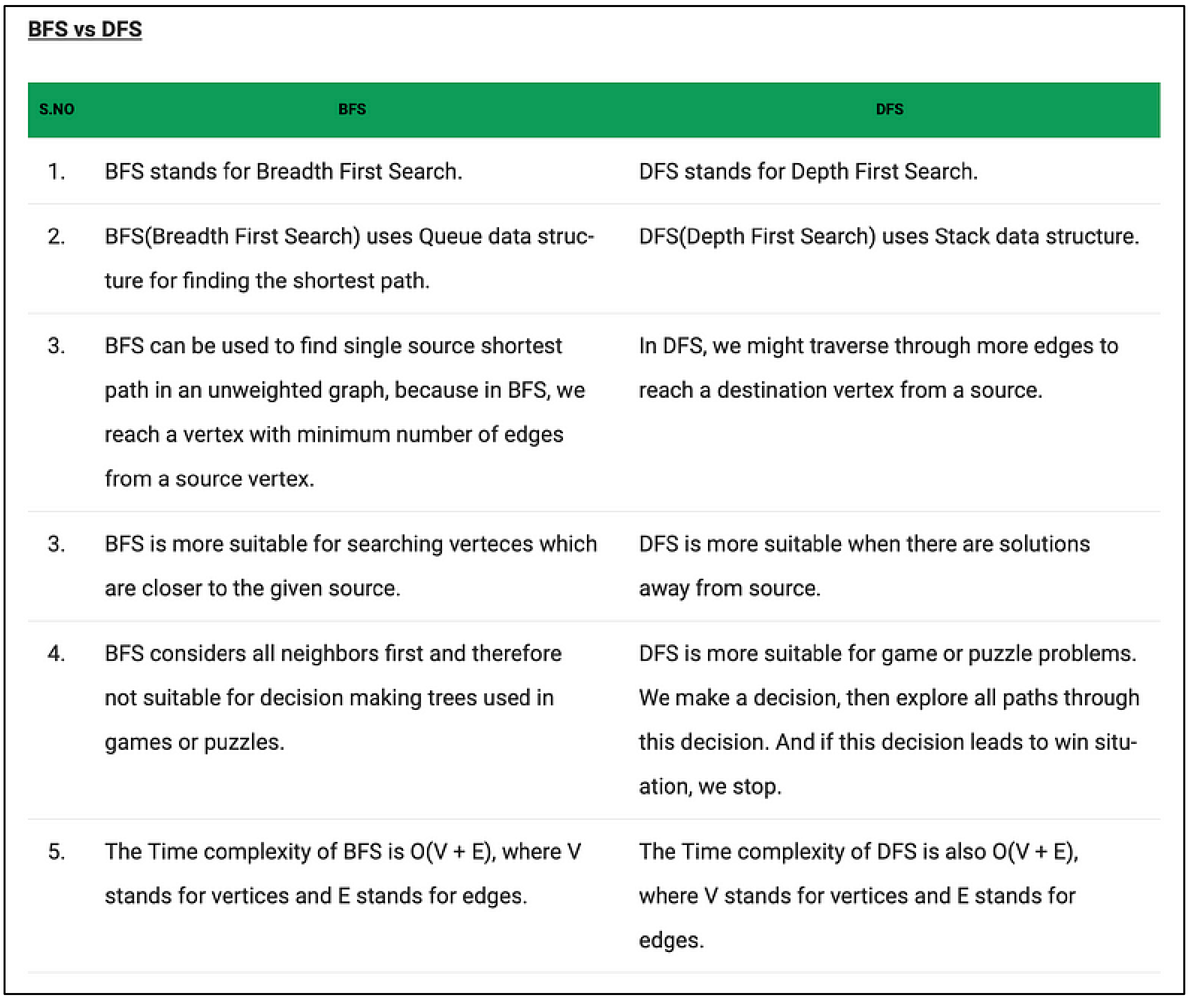
```
-----------------
Depth Level: 29
Next state:
1 2 3
4 5 6
7 0 8
-----------------
Depth Level: 30
Next state:
1 2 3
4 5 6
7 8 0
-----------------
Goal state reached!
Goal state:
1 2 3
4 5 6
7 8 0
```

Difference between Informed and Uninformed search



Informed search	Uninformed search
Also known as Heuristic search	Also known as Blind search
Requires information to perform search	Do not require information to perform search.
Quick solution to problem.	May be time comsuming.
Cost is low.	Comparitively high in cost.
It can be both complete and incomplete.	It is always bound to complete.
The AI gets suggestions regarding how and where to find a solution to any problem.	The AI does not get any suggestions regarding what solution to find and where to find it. Whatever knowledge it gets is out of the information provided.
Eg. Greedy Search A* Search AO* Search Hill Climbing Algorithm	Eg. Depth First Search (DFS) Breadth First Search (BFS) Branch and Bound

Differnce between BFS and DFS



BFS vs DFS

S.NO	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
3.	BFS is more suitable for searching verteces which are closer to the given source.	DFS is more suitable when there are solutions away from source.
4.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop.
5.	The Time complexity of BFS is O(V + E), where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E), where V stands for vertices and E stands for edges.