Assignment 1

**Problem Statement:**  
Implement Depth First Search (DFS) and Breadth First Search (BFS) for the 8-Puzzle problem.

**Theory**

1. The 8-Puzzle Problem

* The 8-puzzle is a sliding puzzle consisting of a 3×3 grid with 8 numbered tiles and one empty space (blank).
* The blank tile can be moved Up, Down, Left, or Right by swapping it with an adjacent tile.
* The objective is to reach the goal state from the given initial state by a sequence of legal moves.

Example:

Initial State:

1 2 3

5 6 0

7 8 4

Goal State:

1 2 3

4 5 6

7 8 0

**2. Search Algorithms**

(a) Breadth-First Search (BFS)

* BFS explores nodes level by level.
* Uses a queue (FIFO) data structure.
* Properties:
  + Complete (always finds a solution if one exists).
  + Optimal (finds the shortest path).
  + High memory usage.

(b) Depth-First Search (DFS)

* DFS explores one branch completely before backtracking.
* Uses a stack (LIFO) or recursion.
* Properties:
  + Requires less memory than BFS.
  + Not guaranteed to find the shortest solution.
  + May get stuck in deep or infinite paths.

**Algorithm Steps**

BFS Steps

1. Start with the initial state, put it into a queue.
2. Dequeue a state and check if it matches the goal.
3. Generate all possible next states by moving the blank.
4. If a state is new (not visited), enqueue it.
5. Repeat until goal is found or queue is empty.

DFS Steps

1. Start with the initial state, put it into a stack (or recursive call).
2. Pop a state and check if it is the goal.
3. Generate all possible moves and push into the stack.
4. Continue exploring deeply until the goal is found or stack is empty.

**Code (BFS Implementation in C++)**

#include <bits/stdc++.h>

using namespace std;

const int N = 3;

struct State {

vector<vector<int>> board;

int x, y;

string path;

};

vector<vector<int>> goal = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 0}

};

int dx[] = {1, 0, -1, 0};

int dy[] = {0, -1, 0, 1};

char dir[] = {'D', 'L', 'U', 'R'};

bool isValid(int x, int y) {

return (x >= 0 && x < N && y >= 0 && y < N);

}

bool isGoal(vector<vector<int>>& b) {

return b == goal;

}

void printBoard(vector<vector<int>>& b) {

for (auto &row : b) {

for (int val : row) {

cout << val << " ";

}

cout << "\n";

}

cout << "\n";

}

void solvePuzzle(vector<vector<int>> initial, int x, int y) {

queue<State> q;

set<vector<vector<int>>> visited;

State start = {initial, x, y, ""};

q.push(start);

visited.insert(initial);

while (!q.empty()) {

State cur = q.front();

q.pop();

if (isGoal(cur.board)) {

cout << "Solution found in " << cur.path.size() << " moves.\n\n";

vector<vector<int>> tmp = initial;

int blankX = x, blankY = y;

printBoard(tmp);

for (char move : cur.path) {

int dirIndex = string("DLUR").find(move);

int newX = blankX + dx[dirIndex];

int newY = blankY + dy[dirIndex];

swap(tmp[blankX][blankY], tmp[newX][newY]);

blankX = newX;

blankY = newY;

printBoard(tmp);

}

return;

}

for (int i = 0; i < 4; i++) {

int newX = cur.x + dx[i];

int newY = cur.y + dy[i]

if (isValid(newX, newY)) {

vector<vector<int>> newBoard = cur.board;

swap(newBoard[cur.x][cur.y], newBoard[newX][newY]);

if (visited.find(newBoard) == visited.end()) {

visited.insert(newBoard);

q.push({newBoard, newX, newY, cur.path + dir[i]});

}

}

}

}

cout << "No solution exists.\n";

}

int main() {

vector<vector<int>> initial = {

{1, 2, 3},

{5, 6, 0},

{7, 8, 4}

};

int x = 1, y = 2;

cout << "Initial State:\n";

printBoard(initial);

cout << "Goal State:\n";

printBoard(goal);

cout << "\nSolving using BFS...\n\n";

solvePuzzle(initial, x, y);

return 0;

}

**Sample Output**

Initial State:

1 2 3

5 6 0

7 8 4

Goal State:

1 2 3

4 5 6

7 8 0

Solving using BFS...

Solution found in 4 moves.

1 2 3

5 6 0

7 8 4

1 2 3

5 0 6

7 8 4

1 2 3

0 5 6

7 8 4

1 2 3

7 5 6

0 8 4

1 2 3

4 5 6

7 8 0

Conclusion

* BFS always finds the shortest path solution but consumes more memory.
* DFS can be implemented similarly but may not always find the shortest solution.
* For small puzzles like the 8-puzzle, BFS works efficiently, but for larger puzzles, more advanced algorithms (like A\* search) are preferred.