Experiment Number: 3

Aim

To write a program in C++ to implement the 8 Puzzle Problem using Artificial Intelligence techniques (e.g., A* Search Algorithm).

Theory

The **8 Puzzle Problem** is a classical problem in the domain of Artificial Intelligence and search algorithms. It consists of a 3x3 grid with 8 numbered tiles and one blank space. The objective is to move the tiles using the blank space until the puzzle reaches the goal configuration.

Key Concepts:

- **State Space**: All possible configurations of the 3x3 grid.
- **Initial State**: The starting configuration of the puzzle.
- **Goal State**: The target configuration (typically the ordered tiles from 1 to 8 with the blank at the end).
- **Operators**: Up, Down, Left, Right (used to move the blank).
- **Heuristic Function**: Estimates the cost from the current node to the goal.
- _*A* Algorithm_*: A best-first search that uses `f(n) = g(n) + h(n)` where:
  - `g(n)` is the cost to reach the current node.
  - `h(n)` is the estimated cost from current node to the goal (heuristic).

Common Heuristic Functions:

- **Misplaced Tile Heuristic**: Number of misplaced tiles compared to goal state.
- **Manhattan Distance**: Sum of vertical and horizontal distances of each tile from its goal position.

---

*Algorithm (A* Search for 8 Puzzle)*

1. Initialize the open list with the starting node.
2. Repeat until the open list is empty:
   1. Extract the node with the lowest `f(n)` from the open list.
   2. If the node is the goal state, return success.

    3. Generate all possible successors (valid moves of blank).
    4. For each successor:
        - Compute `g(n)` and `h(n)`; calculate `f(n)`.
        - If not in the open or closed list, add to open list.
    5. Add the current node to the closed list.
3. If open list becomes empty, return failure (no solution found).

Code (C++)

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <algorithm>

using namespace std;

#define N 3

struct Node {
    vector<vector<int>> mat;
    int x, y, cost, level;
    Node* parent;

    Node(vector<vector<int>> m, int x, int y, int newX, int newY, int level,
Node* parent) {
        mat = m;
        swap(mat[x][y], mat[newX][newY]);
        this->cost = 0;
        this->level = level;
        this->x = newX;
        this->y = newY;
        this->parent = parent;
    }
};

// Goal state
vector<vector<int>> goal = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 0}
};

// Possible moves
int row[] = {1, 0, -1, 0};
```

```cpp
int col[] = {0, -1, 0, 1};

// Heuristic: Number of misplaced tiles
int calculateCost(vector<vector<int>> &initial) {
    int count = 0;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (initial[i][j] && initial[i][j] != goal[i][j])
                count++;
    return count;
}

// Check if position is within bounds
bool isSafe(int x, int y) {
    return (x >= 0 && x < N && y >= 0 && y < N);
}

// Comparison operator for priority queue (min-heap)
struct comp {
    bool operator()(const Node* a, const Node* b) const {
        return (a->cost + a->level) > (b->cost + b->level);
    }
};

// Print the matrix
void printMatrix(vector<vector<int>> mat) {
    for (auto row : mat) {
        for (int val : row)
            cout << val << " ";
        cout << endl;
    }
    cout << "--------" << endl;
}

// A* algorithm
void solve(vector<vector<int>> initial, int x, int y) {
    priority_queue<Node*, vector<Node*>, comp> pq;

    Node* root = new Node(initial, x, y, x, y, 0, nullptr);
    root->cost = calculateCost(initial);
    pq.push(root);

    while (!pq.empty()) {
        Node* min = pq.top();
        pq.pop();

        if (min->cost == 0) {
            // Goal reached: print path
            vector<Node*> path;
            while (min) {
```

```
                path.push_back(min);
                min = min->parent;
            }
            reverse(path.begin(), path.end());
            for (auto node : path)
                printMatrix(node->mat);
            return;
        }

        // Explore neighbors
        for (int i = 0; i < 4; i++) {
            int newX = min->x + row[i];
            int newY = min->y + col[i];

            if (isSafe(newX, newY)) {
                Node* child = new Node(min->mat, min->x, min->y, newX, newY,
min->level + 1, min);
                child->cost = calculateCost(child->mat);
                pq.push(child);
            }
        }
    }
}
```

Input/Output Examples

Sample Input:

```
vector<vector<int>> initial = {
    {1, 2, 3},
    {5, 6, 0},
    {7, 8, 4}
};
int x = 1, y = 2; // Position of blank tile (0)
solve(initial, x, y);
```

Expected Output:

```
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
7 5 6
8 0 4
--------
1 2 3
7 5 6
8 4 0
--------
1 2 3
7 5 0
8 4 6
--------
...
1 2 3
4 5 6
7 8 0
--------
```

Analysis of Code and Algorithm

- **Time Complexity**:
  The time complexity is **O(b^d)** where `b` is the branching factor (up to 4 moves) and `d` is the depth of the solution. A* reduces time using a good heuristic.
- **Space Complexity**:
  Memory usage is high due to storage of multiple nodes in the open list and paths (approx. **O(b^d)**).
- **Efficiency Considerations**:
  - Using a **priority queue** ensures optimal node selection.
  - A* ensures the shortest path due to its cost function.
- **Key Operations**:
  - Matrix manipulation using 2D vectors.
  - Heuristic evaluation using misplaced tile strategy.
  - Recursive backtracking to retrieve path.

Real-Life Applications

- **Robot Path Planning**: Robots solving navigation or movement constraints in grid-like environments.
- **Game Solvers**: Automated solvers for puzzles and board games.
- **AI in Logistics**: Optimization in warehouse movement of packages or automated guided vehicles (AGVs).

Conclusion

In this experiment, we successfully implemented the **8 Puzzle Problem** using the _*A* Search Algorithm_* in C++. We applied AI principles such as heuristic search, state space exploration, and cost optimization. The program was able to find a sequence of valid moves that transform the initial state to the goal state, demonstrating the application of informed search in AI.
