ASSIGNMENT 2

8- Puzzle Problem

AQIB HABIB MEMON
Reg: 450062

Chapter 1. Solution of 8-Puzzle Problem (A* Search)

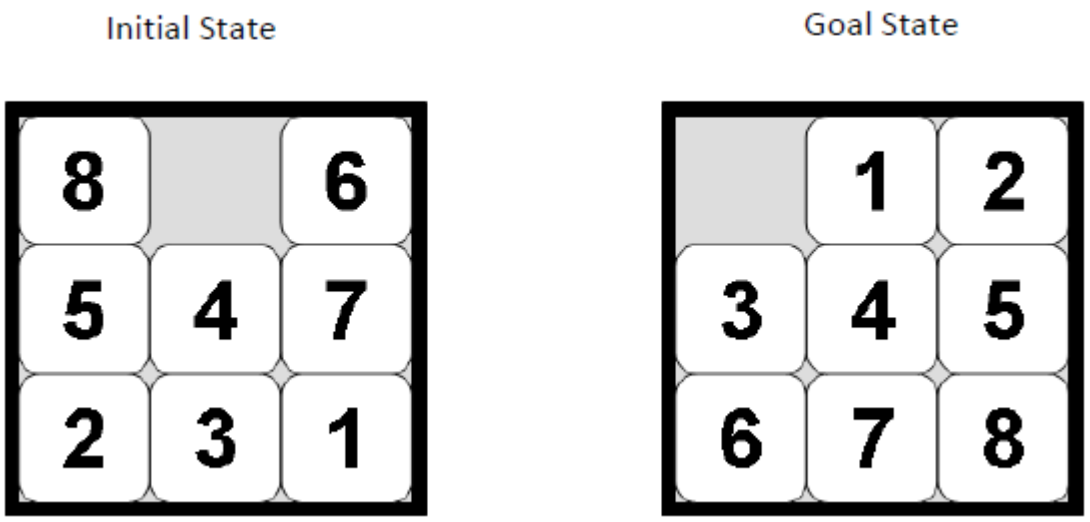


Figure 1 Given 8-Puzzle State

1.1 Pseudo-Code and Logic (Manhattan Heuristic)

```
A* Algorithm:

function AStar(startState, goalState):
    create an empty priority queue (PQ) or min-heap
    add startState to PQ with priority f(startState) = g(startState) + h(startState, goalState)
    create an empty set to store visited states
    mark startState as visited


    while PQ is not empty:
        currentState = remove state with the lowest f-score from PQ
            if currentState is the goalState:
            return currentState (goal found)
            for each neighborState of currentState:
            if neighborState is not in the visited set:
            mark neighborState as visited
            add neighborState to PQ with priority f(neighborState) = g(neighborState) + h(neighborState, goalState)
        return null (goal not reachable)
```

Code Block 1 Pseudo code for A Search*

Code Block 1 represents the pseudo code and logic of the A* algorithm. Based on this logic, the following section will explain the code in chunks of what each part and function does.

```
struct PuzzleState {
    vector<vector<int>> board;
    int heuristic;
    int moves;
    vector<vector<int>> path;
    PuzzleState(const vector<vector<int>>& b, int h, int m) : board(b), heuristic(h), moves(m) {}
};
```

Code Block 2 PuzzleState Structure

This structure represents the state of the puzzle with members containing board of type vector, integer out the code to represent the board's state and to push the puzzle state into the queue.

There are four helper functions used throughout the main code of the A* Search namely,

- printBoard()
- CalculateHeuristic()
- IsBoardsEqual()
- boardtoString()

```
void printBoard(const vector<vector<int>>& board) {
    for (int i=0;i<3;i++) {
        for (int j=0;j<3;j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}
```

Code Block 3 Helper Function printBoard

```
int calculateHeuristic(const vector<vector<int>>& board, const vector<vector<int>>& goal) {
    int h2 = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (board[i][j]==goal[i][j]) {
                h2++;
            }
            else
            continue;
        }
    }
    return h2;
}
```

Code Block 4 Helper Function calculateHeuristic [1]

```
bool isBoardsEqual(const vector<vector<int>>& board1, const vector<vector<int>>& board2) {
    return board1 == board2;
}
```

Code Block 5 Helper Function IsboardsEqual

```
string boardToString(const vector<vector<int>>& board) {
    string str;
    for (const auto& row : board) {
        for (int num : row) {
            str += to_string(num) + " ";
        }
        str += "\n";
    }
    return str;
}
```

Code Block 6 Helper function boardToString

These helper functions are used throughout the main code to calculate the heuristic of each state, to print the sate and to specify whether the goal state has been reached or not.

```
auto comparator = [](const PuzzleState& a, const PuzzleState& b) {
        return (a.moves + a.heuristic) > (b.moves + b.heuristic);
    };
    priority_queue<PuzzleState, vector<PuzzleState>, decltype(comparator)> pq(comparator);
    unordered_set<string> visited;
    int initialHeuristic = calculateHeuristic(initial, goal);
    pq.emplace(initial, initialHeuristic, 0);
```

Code Block 7 priority queue and initialization

The priority queue uses a lamba function named comparator to compare the states of the puzzle based on their heuristic and cost values. The definition can be found here. Unlike GBFS, which is based on only heuristic values, this lamba function also uses the cost.

```
    while (!pq.empty()) {
        PuzzleState current = pq.top();
        pq.pop();
        if (isBoardsEqual(current.board, goal)) {
            cout << "Goal State Found!" << endl;
            cout << "Number of moves: " << current.moves << endl;
            return;
        }
```

Code Block 8 Goal Check

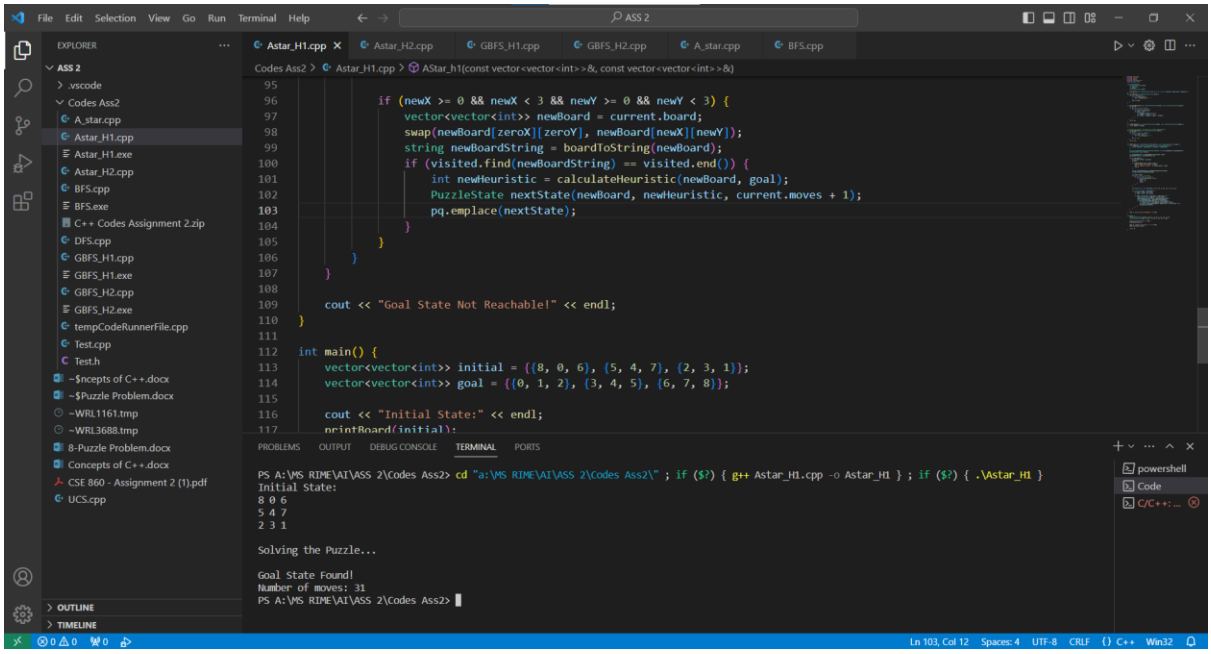
These lines of code check whether the goal has been found or not and also counts the number of moves.

```
int zeroX, zeroY;
      for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
          if (current.board[i][j] == 0) {
            zeroX = i;
            zeroY = j;
            break;
          }
        }
      }
      vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
      for (const auto& dir : directions) {
        int newX = zeroX + dir.first;
        int newY = zeroY + dir.second;
        if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
          vector<vector<int>> newBoard = current.board;
          swap(newBoard[zeroX][zeroY], newBoard[newX][newY]);
          string newBoardString = boardToString(newBoard);
          if (visited.find(newBoardString) == visited.end()) {
            int newHeuristic = calculateHeuristic(newBoard, goal);
            PuzzleState nextState(newBoard, newHeuristic, current.moves + 1);
            pq.emplace(nextState);
          }
        }
      }
    }
    cout << "Goal State Not Reachable!" << endl;
}
```

Code Block 9 Position of Zero and Possible Moves

After the goal check, and selection of state based on heuristic, the position of zero is determined and its possible moves. New puzzle state is generated, its heuristic is calculated and it emplaced into the priority queue.

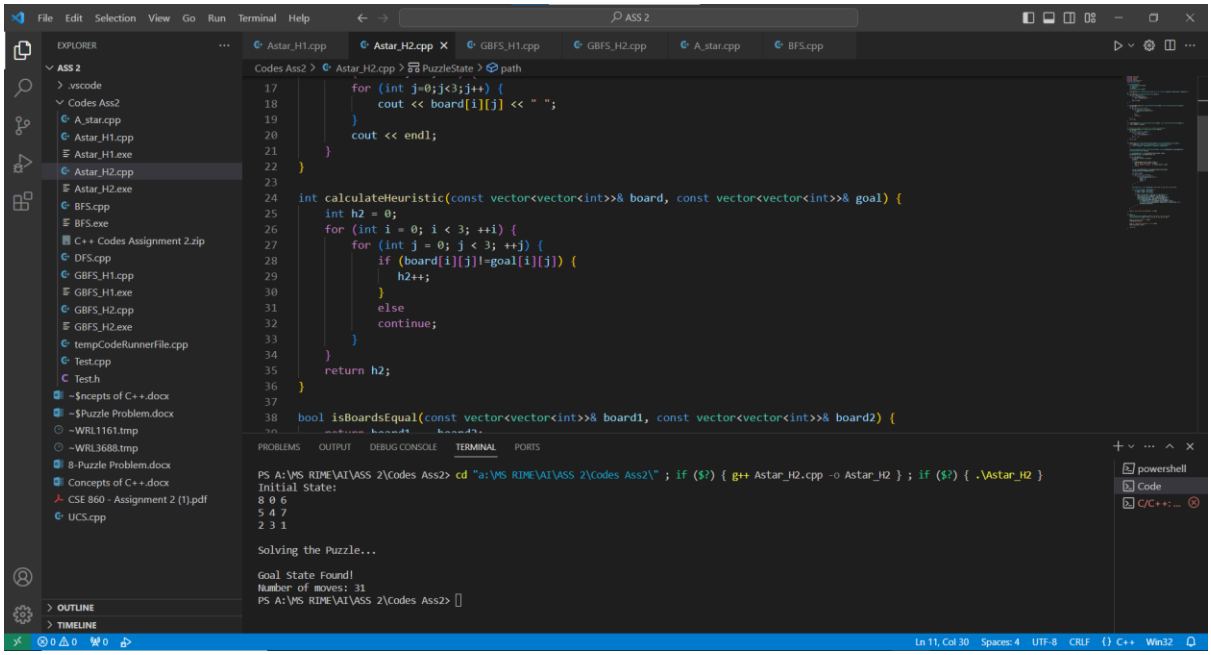
1.2 Results



```
if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
    vector<vector<int>> newBoard = current.board;
    swap(newBoard[zeroX][zeroY], newBoard[newX][newY]);
    string newBoardString = boardToString(newBoard);
    if (visited.find(newBoardString) == visited.end()) {
        int newHeuristic = calculateHeuristic(newBoard, goal);
        PuzzleState nextState(newBoard, newHeuristic, current.moves + 1);
        pq.emplace(nextState);
    }
}
}
}

cout << "Goal State Not Reachable!" << endl;
}

int main() {
    vector<vector<int>> initial = {{8, 0, 6}, {5, 4, 7}, {2, 3, 1}};
    vector<vector<int>> goal = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};

    cout << "Initial State:" << endl;
```

```
PS A:\MS RIME\AI\ASS 2\Codes Ass2> cd "a:\MS RIME\AI\ASS 2\Codes Ass2\" ; if ($?) { g++ Astar_H1.cpp -o Astar_H1 } ; if ($?) { .\Astar_H1 }
Initial State:
8 0 6
5 4 7
2 3 1

Solving the Puzzle...

Goal State Found!
Number of moves: 31
PS A:\MS RIME\AI\ASS 2\Codes Ass2>
```

Code Block 10 Goal State in 31 moves A (Manhattan Heuristic)*



```
        for (int j=0;j<3;j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int calculateHeuristic(const vector<vector<int>>& board, const vector<vector<int>>& goal) {
    int h2 = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (board[i][j]!=goal[i][j]) {
                h2++;
            }
            else
            continue;
        }
    }
    return h2;
}

bool isBoardsEqual(const vector<vector<int>>& board1, const vector<vector<int>>& board2) {
```

```
PS A:\MS RIME\AI\ASS 2\Codes Ass2> cd "a:\MS RIME\AI\ASS 2\Codes Ass2\" ; if ($?) { g++ Astar_H2.cpp -o Astar_H2 } ; if ($?) { .\Astar_H2 }
Initial State:
8 0 6
5 4 7
2 3 1

Solving the Puzzle...

Goal State Found!
Number of moves: 31
PS A:\MS RIME\AI\ASS 2\Codes Ass2>
```

Code Block 11 Goal State in 31 moves A (Misplaced Tiles)*

Based on the given puzzle initial state, it takes 31 moves to reach the goal state with both Manhattan Distance and misplaced tiles. The complete codes can be found in the appendix for both Manhattan and misplaced tiles heuristics.

Chapter 2. Solution of 8-Puzzle Problem (Greedy Best First Search)

2.1 Pseudo-Code and Logic (Manhattan Heuristic)

```
Best-First Search (BFS):


function BestFirstSearch(startState, goalState):
    create an empty priority queue (PQ) or min-heap
    add startState to PQ with priority based on a heuristic function (heuristic(startState, goalState))
    create an empty set to store visited states
    mark startState as visited

    while PQ is not empty:
        currentState = remove state with the highest priority from PQ

        if currentState is the goalState:
            return currentState (goal found)

        for each neighborState of currentState:
            if neighborState is not in the visited set:
                mark neighborState as visited
                add neighborState to PQ with priority based on heuristic(neighborState, goalState)

    return null (goal not reachable)
```

Code Block 12 Pseudo-Code of Best-First Search

Code Block 12 represents the pseudo code and logic of the GBFS algorithm. Based on this logic, the following section will explain the code in chunks of what each part and function does.

```
struct PuzzleState {
    vector<vector<int>> board;
    int heuristic;
    int moves;
    PuzzleState(const vector<vector<int>>& b, int h, int m) : board(b), heuristic(h), moves(m) {}
};
```

Code Block 13 PuzzleState Structure

This structure represents the state of the puzzle with members containing board of type vector, integer heuristic and integer moves. This structure is used through out the code to represent the board's state and to push the puzzle state into the

There are four helper functions used throughout the main code of the Best-First Search namely,

- printBoard()
- CalculateHeuristic
- IsBoardsEqual
- boardtoString()

```
void printBoard(const vector<vector<int>>& board) {
    for (int i=0;i<3;i++) {
        for (int j=0;j<3;j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}
```

Code Block 14 Helper Function printBoard

```
int calculateHeuristic(const vector<vector<int>>& board, const vector<vector<int>>& goal) {
    int h2 = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (board[i][j]==goal[i][j]) {
                h2++;
            }
            else
            continue;
        }
    }
    return h2;
}
```

Code Block 15 Helper Function calculateHeuristic

```
bool isBoardsEqual(const vector<vector<int>>& board1, const vector<vector<int>>& board2) {
    return board1 == board2;
}
```

Code Block 16 Helper Function IsboardsEqual

```
string boardToString(const vector<vector<int>>& board) {
    string str;
    for (const auto& row : board) {
        for (int num : row) {
```

```
            str += to_string(num) + " ";
        }
        str += "\n";
    }
    return str;
}
```

Code Block 17 Helper Function boardToString

These helper functions are used throughout the main code to calculate the heuristic of each state, to print the sate and to specify whether the goal state has been reached or not.

```
void GBFS_h1(const vector<vector<int>>& initial, const vector<vector<int>>& goal) {
    // ... (initialization and insertion of initial state into priority queue)
    while (!pq.empty()) {
        PuzzleState current = pq.top();
        pq.pop();
        if (isBoardsEqual(current.board, goal)) {
            // Goal state found
            // ...
            return;
        }
        // ... (Explore possible moves, update visited set, and insert new states into priority queue)
    }
    // Goal state not reachable
    // ...
}
```

Code Block 18 GBFS Algorithm

Code Block 18 is the main algorithm block for the GBFS algorithm.

```
auto comparator = [](const PuzzleState& a, const PuzzleState& b) {
        return a.heuristic > b.heuristic;
    };
    priority_queue<PuzzleState, vector<PuzzleState>, decltype(comparator)> pq(comparator);
    unordered_set<string> visited;
    int initialHeuristic = calculateHeuristic(initial, goal);
    pq.emplace(initial, initialHeuristic, 0); // Emplace function is similar to push_back function [1]
```

Code Block 19 Priority Queue and Initialization [2]

The priority queue uses a lamba function named comparator to compare the states of the puzzle based on their heuristic values. The definition can be found here.

```
while (!pq.empty()) {
      PuzzleState current = pq.top();
      pq.pop();
      if (isBoardsEqual(current.board, goal)) {
         cout << "Goal State Found!" << endl;
         cout << "Number of moves: " << current.moves << endl;
                return;
}
```

Code Block 20 Goal Check

These lines of code check whether the goal has been found or not and also counts the number of moves.

```
  // Find the position of the empty space (0)
      int zeroX, zeroY;
      for (int i = 0; i < 3; ++i) {
         for (int j = 0; j < 3; ++j) {
            if (current.board[i][j] == 0) {
               zeroX = i;
               zeroY = j;
               break;
            }
         }
      }
vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
      for (const auto& dir : directions) {
         int newX = zeroX + dir.first;
         int newY = zeroY + dir.second;
         if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
            vector<vector<int>> newBoard = current.board;
            swap(newBoard[zeroX][zeroY], newBoard[newX][newY]);
            string newBoardString = boardToString(newBoard);
            if (visited.find(newBoardString) == visited.end()) {
               int newHeuristic = calculateHeuristic(newBoard, goal);
               PuzzleState nextState(newBoard, newHeuristic, current.moves + 1);
               pq.emplace(nextState);
            }
         }
      }
   }
```

Code Block 21 Position of Zero and Possible Moves

After the goal check, and selection of state based on heuristic, the position of zero is determined and its possible moves. New puzzle state is generated, its heuristic is calculated and it emplaced into the priority queue.

2.2 Results

Based on the given puzzle initial state, it takes 101 moves to reach the goal state with Manhattan Distance and 137 Moves using misplaced tiles. The complete codes can be found in the appendix for both Manhattan and misplaced tiles heuristics.

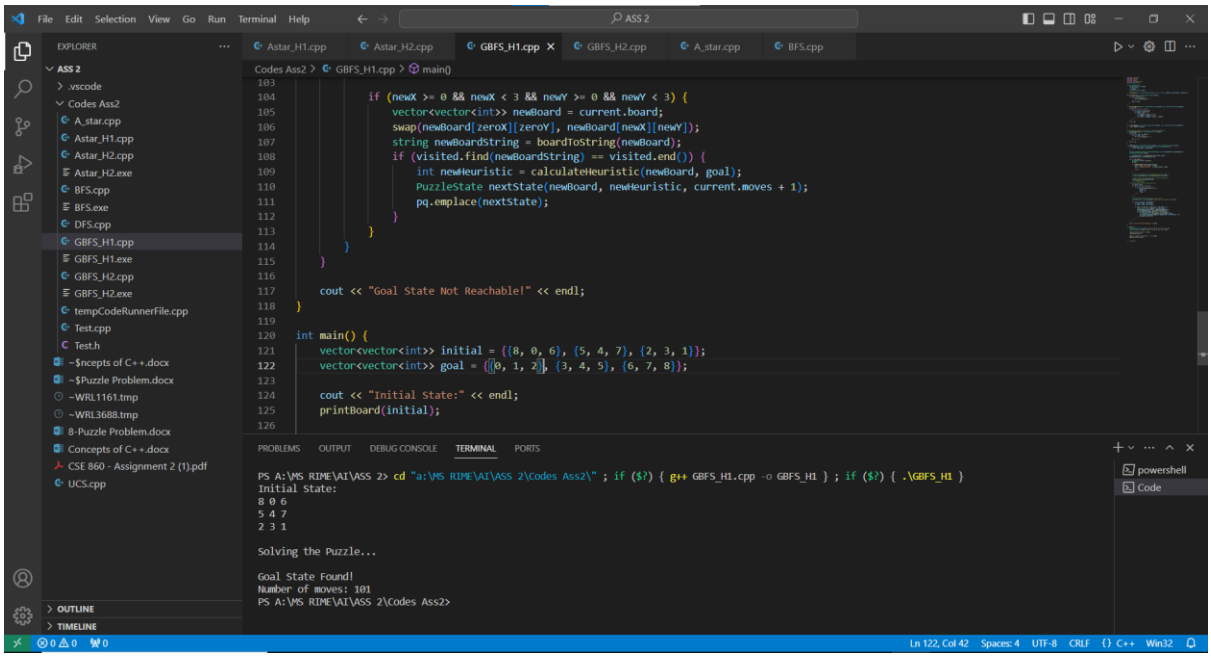


Figure 2 Goal State in 101 moves GBFS (Manhattan Heuristic)

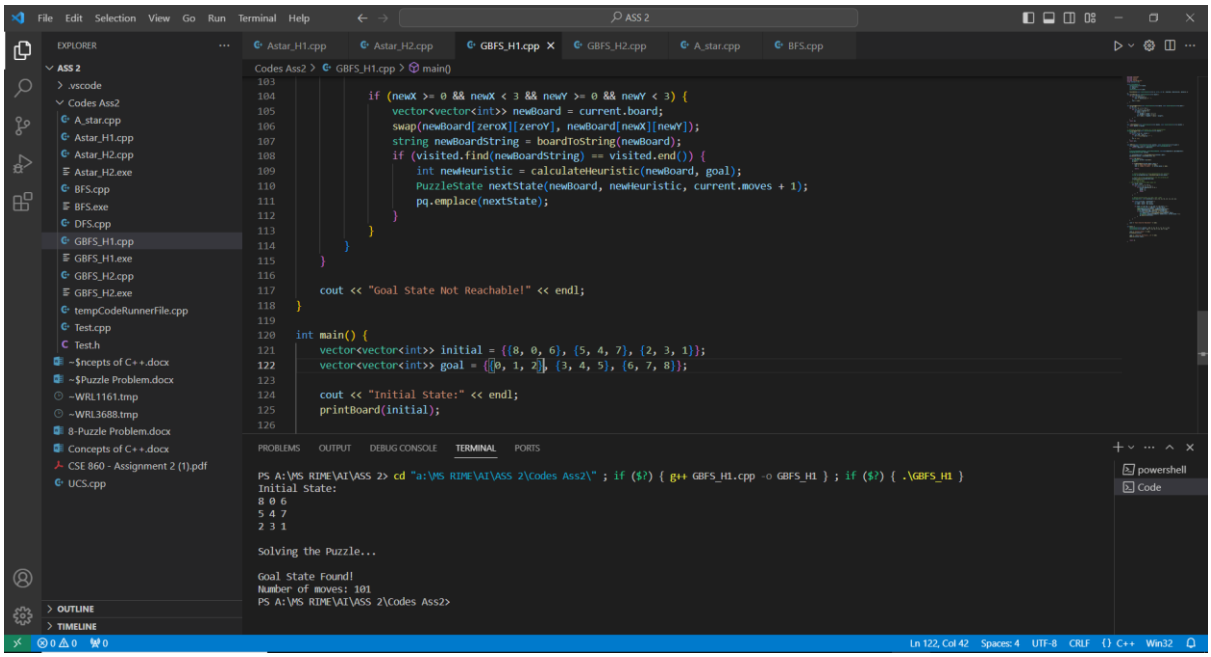


Figure 3 Goal State in 137 moves GBFS (Misplaced Tiles)

Chapter 3. Appendix of Codes:

I. Greedy Best-First Search with Manhattan Distance:

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

struct PuzzleState {
    vector<vector<int>> board;
    int heuristic;
    int moves;

    PuzzleState(const vector<vector<int>>& b, int h, int m) : board(b), heuristic(h), moves(m) {}
};
void printBoard(const vector<vector<int>>& board) {
    for (int i=0;i<3;i++) {
        for (int j=0;j<3;j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int calculateHeuristic(const vector<vector<int>>& board, const vector<vector<int>>& goal) {
    int h1 = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            int value = board[i][j];
            if (value != 0) {
                int targetX = (value - 1) / 3;
                int targetY = (value - 1) % 3;
                h1 += abs(i - targetX) + abs(j - targetY);
            }
        }
    }
    return h1;
}

bool isBoardsEqual(const vector<vector<int>>& board1, const vector<vector<int>>& board2) {
    return board1 == board2;
}

// Function to convert a 2D vector to a string representation
string boardToString(const vector<vector<int>>& board) {
    string str;
    for (const auto& row : board) {
        for (int num : row) {
            str += to_string(num) + " ";
        }
        str += "\n";
    }
    return str;
}

void GBFS_h1(const vector<vector<int>>& initial, const vector<vector<int>>& goal) {
    auto comparator = [](const PuzzleState& a, const PuzzleState& b) {
        return a.heuristic > b.heuristic;
    };

    priority_queue<PuzzleState, vector<PuzzleState>, decltype(comparator)> pq(comparator);
    unordered_set<string> visited;

    int initialHeuristic = calculateHeuristic(initial, goal);
    pq.emplace(initial, initialHeuristic, 0);

    while (!pq.empty()) {
        PuzzleState current = pq.top();
        pq.pop();

        if (isBoardsEqual(current.board, goal)) {
```

```
            cout << "Goal State Found!" << endl;
            cout << "Number of moves: " << current.moves << endl;
            return;
        }

        // Convert the board to string representation for insertion
        string currentBoardString = boardToString(current.board);

        // Insert the string representation into the unordered_set
        visited.insert(currentBoardString);
        // printBoard(curr);
        //     cout << endl;
   // Find the position of the empty space (0)
        int zeroX, zeroY;
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                if (current.board[i][j] == 0) {
                    zeroX = i;
                    zeroY = j;
                    break;
                }
            }
        }

        // Define possible moves: up, down, left, right
        vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        for (const auto& dir : directions) {
            int newX = zeroX + dir.first;
            int newY = zeroY + dir.second;

            if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
                vector<vector<int>> newBoard = current.board;
                swap(newBoard[zeroX][zeroY], newBoard[newX][newY]);
                string newBoardString = boardToString(newBoard);
                if (visited.find(newBoardString) == visited.end()) {
                    int newHeuristic = calculateHeuristic(newBoard, goal);
                    PuzzleState nextState(newBoard, newHeuristic, current.moves + 1);
                    pq.emplace(nextState);
                }
            }
        }
    }

    cout << "Goal State Not Reachable!" << endl;
}

int main() {
    vector<vector<int>> initial = {{8, 0, 6}, {5, 4, 7}, {2, 3, 1}};
    vector<vector<int>> goal = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};

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

    cout << "\nSolving the Puzzle...\n" << endl;
    GBFS_h1(initial, goal);



    return 0;
}
```

II. Greedy Best-First Search with Misplaced Tiles:

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

struct PuzzleState {
    vector<vector<int>> board;
    int heuristic;
    int moves;
    vector<vector<int>> path;

    PuzzleState(const vector<vector<int>>& b, int h, int m) : board(b), heuristic(h), moves(m) {}
};
void printBoard(const vector<vector<int>>& board) {
    for (int i=0;i<3;i++) {
        for (int j=0;j<3;j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int calculateHeuristic(const vector<vector<int>>& board, const vector<vector<int>>& goal) {
    int h2 = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (board[i][j]!=goal[i][j]) {
               h2++;
            }
            else
            continue;
        }
    }
    return h2;
}

bool isBoardsEqual(const vector<vector<int>>& board1, const vector<vector<int>>& board2) {
    return board1 == board2;
}

// Function to convert a 2D vector to a string representation
string boardToString(const vector<vector<int>>& board) {
    string str;
    for (const auto& row : board) {
        for (int num : row) {
            str += to_string(num) + " ";
        }
        str += "\n";
    }
    return str;
}

void GBFS_h1(const vector<vector<int>>& initial, const vector<vector<int>>& goal) {
    auto comparator = [](const PuzzleState& a, const PuzzleState& b) {
        return a.heuristic > b.heuristic;
    };

    priority_queue<PuzzleState, vector<PuzzleState>, decltype(comparator)> pq(comparator);
    unordered_set<string> visited;

    int initialHeuristic = calculateHeuristic(initial, goal);
    pq.emplace(initial, initialHeuristic, 0);

    while (!pq.empty()) {
        PuzzleState current = pq.top();
        pq.pop();

        if (isBoardsEqual(current.board, goal)) {
            cout << "Goal State Found!" << endl;
```

```
            cout << "Number of moves: " << current.moves << endl;
            return;
        }

        // Convert the board to string representation for insertion
        string currentBoardString = boardToString(current.board);

        // Insert the string representation into the unordered_set
        visited.insert(currentBoardString);
        // printBoard(curr);
        //     cout << endl;
   // Find the position of the empty space (0)
        int zeroX, zeroY;
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                if (current.board[i][j] == 0) {
                    zeroX = i;
                    zeroY = j;
                    break;
                }
            }
        }

        // Define possible moves: up, down, left, right
        vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        for (const auto& dir : directions) {
            int newX = zeroX + dir.first;
            int newY = zeroY + dir.second;

            if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
                vector<vector<int>> newBoard = current.board;
                swap(newBoard[zeroX][zeroY], newBoard[newX][newY]);
                string newBoardString = boardToString(newBoard);
                if (visited.find(newBoardString) == visited.end()) {
                    int newHeuristic = calculateHeuristic(newBoard, goal);
                    PuzzleState nextState(newBoard, newHeuristic, current.moves + 1);
                    pq.emplace(nextState);
                }
            }
        }
    }

    cout << "Goal State Not Reachable!" << endl;
}

int main() {
    vector<vector<int>> initial = {{8, 0, 6}, {5, 4, 7}, {2, 3, 1}};
    vector<vector<int>> goal = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};

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

    cout << "\nSolving the Puzzle...\n" << endl;
    GBFS_h1(initial, goal);



    return 0;
}
```

III. A* Search with Manhattan Distance:

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

struct PuzzleState {
    vector<vector<int>> board;
    int heuristic;
    int moves;
    vector<vector<int>> path;

    PuzzleState(const vector<vector<int>>& b, int h, int m) : board(b), heuristic(h), moves(m) {}
};
void printBoard(const vector<vector<int>>& board) {
    for (int i=0;i<3;i++) {
        for (int j=0;j<3;j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int calculateHeuristic(const vector<vector<int>>& board, const vector<vector<int>>& goal) {
    int h1 = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            int value = board[i][j];
            if (value != 0) {
                int targetX = (value - 1) / 3;
                int targetY = (value - 1) % 3;
                h1 += abs(i - targetX) + abs(j - targetY);
            }
        }
    }
    return h1;
}

bool isBoardsEqual(const vector<vector<int>>& board1, const vector<vector<int>>& board2) {
    return board1 == board2;
}

// Function to convert a 2D vector to a string representation
string boardToString(const vector<vector<int>>& board) {
    string str;
    for (const auto& row : board) {
```

```
        for (int num : row) {
            str += to_string(num) + " ";
        }
        str += "\n";
    }
    return str;
}

void AStar_h1(const vector<vector<int>>& initial, const vector<vector<int>>& goal) {
    auto comparator = [](const PuzzleState& a, const PuzzleState& b) {
        return (a.moves + a.heuristic) > (b.moves + b.heuristic);
    };

    priority_queue<PuzzleState, vector<PuzzleState>, decltype(comparator)> pq(comparator);
    unordered_set<string> visited;

    int initialHeuristic = calculateHeuristic(initial, goal);
    pq.emplace(initial, initialHeuristic, 0);

    while (!pq.empty()) {
        PuzzleState current = pq.top();
        pq.pop();

        if (isBoardsEqual(current.board, goal)) {
            cout << "Goal State Found!" << endl;
            cout << "Number of moves: " << current.moves << endl;
            return;
        }

        string currentBoardString = boardToString(current.board);
        visited.insert(currentBoardString);

        int zeroX, zeroY;
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                if (current.board[i][j] == 0) {
                    zeroX = i;
                    zeroY = j;
                    break;
                }
            }
        }

        vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        for (const auto& dir : directions) {
            int newX = zeroX + dir.first;
            int newY = zeroY + dir.second;
```

```
            if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
                vector<vector<int>> newBoard = current.board;
                swap(newBoard[zeroX][zeroY], newBoard[newX][newY]);
                string newBoardString = boardToString(newBoard);
                if (visited.find(newBoardString) == visited.end()) {
                    int newHeuristic = calculateHeuristic(newBoard, goal);
                    PuzzleState nextState(newBoard, newHeuristic, current.moves + 1);
                    pq.emplace(nextState);
                }
            }
        }
    }

    cout << "Goal State Not Reachable!" << endl;
}

int main() {
    vector<vector<int>> initial = {{8, 0, 6}, {5, 4, 7}, {2, 3, 1}};
    vector<vector<int>> goal = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};

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

    cout << "\nSolving the Puzzle...\n" << endl;
    AStar_h1(initial, goal);

    return 0;
}
```

IV. A* Search with Misplaced Tiles:

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

struct PuzzleState {
    vector<vector<int>> board;
    int heuristic;
    int moves;
    vector<vector<int>> path;

    PuzzleState(const vector<vector<int>>& b, int h, int m) : board(b), heuristic(h), moves(m) {}
};
void printBoard(const vector<vector<int>>& board) {
    for (int i=0;i<3;i++) {
        for (int j=0;j<3;j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int calculateHeuristic(const vector<vector<int>>& board, const vector<vector<int>>& goal) {
    int h2 = 0;
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j) {
            if (board[i][j]!=goal[i][j]) {
               h2++;
            }
            else
            continue;
        }
    }
    return h2;
}

bool isBoardsEqual(const vector<vector<int>>& board1, const vector<vector<int>>& board2) {
    return board1 == board2;
}

// Function to convert a 2D vector to a string representation
string boardToString(const vector<vector<int>>& board) {
    string str;
    for (const auto& row : board) {
        for (int num : row) {
            str += to_string(num) + " ";
```

```
        }
        str += "\n";
    }
    return str;
}

void AStar_h2(const vector<vector<int>>& initial, const vector<vector<int>>& goal) {
    auto comparator = [](const PuzzleState& a, const PuzzleState& b) {
        return (a.moves + a.heuristic) > (b.moves + b.heuristic);
    };

    priority_queue<PuzzleState, vector<PuzzleState>, decltype(comparator)> pq(comparator);
    unordered_set<string> visited;

    int initialHeuristic = calculateHeuristic(initial, goal);
    pq.emplace(initial, initialHeuristic, 0);

    while (!pq.empty()) {
        PuzzleState current = pq.top();
        pq.pop();

        if (isBoardsEqual(current.board, goal)) {
            cout << "Goal State Found!" << endl;
            cout << "Number of moves: " << current.moves << endl;
            return;
        }

        string currentBoardString = boardToString(current.board);
        visited.insert(currentBoardString);

        int zeroX, zeroY;
        for (int i = 0; i < 3; ++i) {
            for (int j = 0; j < 3; ++j) {
                if (current.board[i][j] == 0) {
                    zeroX = i;
                    zeroY = j;
                    break;
                }
            }
        }

        vector<pair<int, int>> directions = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

        for (const auto& dir : directions) {
            int newX = zeroX + dir.first;
            int newY = zeroY + dir.second;

            if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {
```

```
                vector<vector<int>> newBoard = current.board;
                swap(newBoard[zeroX][zeroY], newBoard[newX][newY]);
                string newBoardString = boardToString(newBoard);
                if (visited.find(newBoardString) == visited.end()) {
                    int newHeuristic = calculateHeuristic(newBoard, goal);
                    PuzzleState nextState(newBoard, newHeuristic, current.moves + 1);
                    pq.emplace(nextState);
                }
            }
        }
    }

    cout << "Goal State Not Reachable!" << endl;
}

int main() {
    vector<vector<int>> initial = {{8, 0, 6}, {5, 4, 7}, {2, 3, 1}};
    vector<vector<int>> goal = {{0, 1, 2}, {3, 4, 5}, {6, 7, 8}};

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

    cout << "\nSolving the Puzzle...\n" << endl;
    AStar_h2(inital, goal);

    return 0;
}
```