Assignment 01

Name - Sk Fardeen Hossain
Enrollment No - 2021CSB023
Department - Computer Science and Technology

Question 01:-

Create a program designed to address a classic optimization challenge involving the determination of the most efficient route among a set of cities. In this problem, the objective is to find the shortest possible path that visits each city exactly once, returning to the starting city. The program should adeptly explore and assess potential routes, providing an optimal solution. Accomplishing this task necessitates the incorporation of an algorithm, whether heuristic or exact, capable of efficiently navigating the intricacies inherent in identifying the shortest path within a specified collection of cities.Use the hill climbing algorithm to solve the problem.

Intro to the Hill Climbing Algorithm for Search Space Optimisation -

Hill Climbing is a heuristic search algorithm used primarily for mathematical optimization problems in artificial intelligence (AI). It is a form of local search, which means it focuses on finding the optimal solution by making incremental changes to an existing solution and then evaluating whether the new solution is better than the current one.

It starts by taking a random initial solution to the problem, which in this case will be a sequence of cities that starts and ends at the same point. After that we select the neighbours of the current sequence in the search space (which is a set of city-sequences) which are related by one city swap (we can come up with crossover as well). If the neighbour gives an optimal value, we select the neighbour as the current solution and continue this process till an optima (local/global) is reached or till maximum number of iterations.

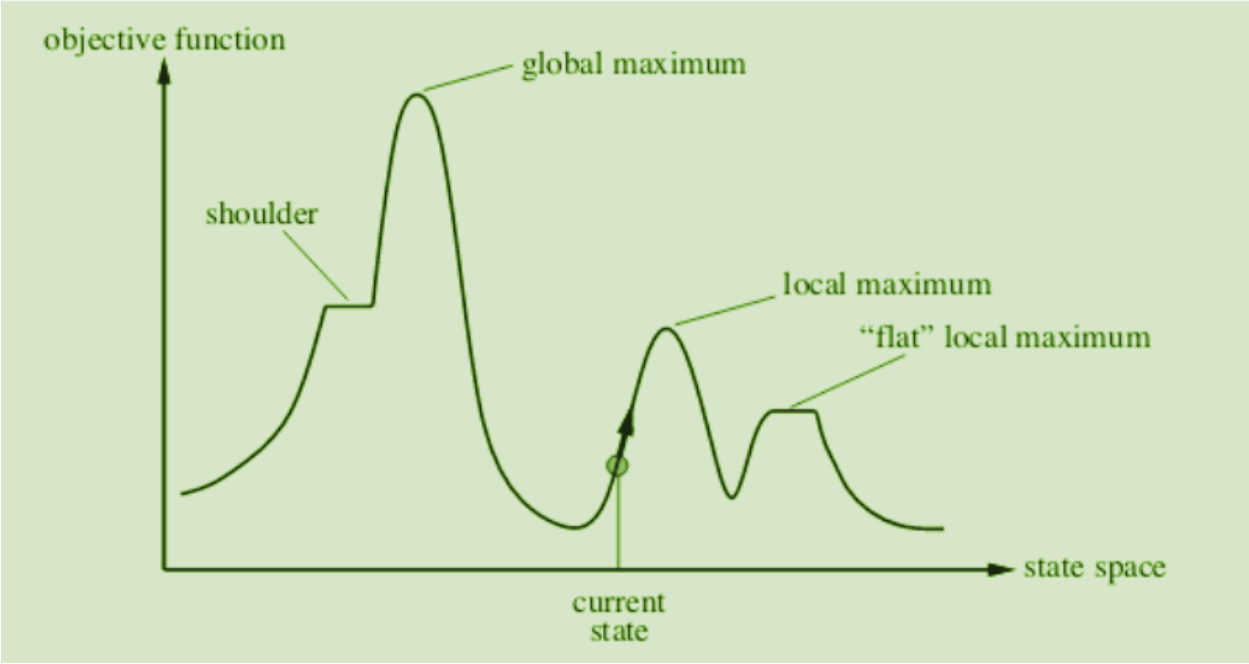


Fig01:- Showing how the hill climbing algorithm works and how it can suffer from the stuck at local optima problem

Code :

```
/*
```

```
   The given problem statement reduces to finding an element in the set of
all
   possible hamiltonian circuits in a graph G(V,E) that has the least
total cost.
   This is basically the travelling salesman problem (TSP).

   Assumptions:-
       1. The cost of going from city A to city B is the same as that from
city B to city A.
           So the distance matrix should be symmetric.

       2. The distance matrix is generated in a way that does not give
stuck at local optima
           problem most of the time(alpha>0.95), so hill climbing
algorithm can be used

   Heuristic , Stochastic Algorithm:-

   1. Find a random solution and calculate its optimizer mapping(total
cost)
   2. Find the neighbours of the random solution that are 2 hamming
distance away (1 swap) from each other.
   3. Compare the costs and swap if a better neighbour is found.
   4. Perform steps 1 to 3 till max iterations or there exist no better
neighbour
*/
#include <bits/stdc++.h>

using namespace std;

class TSP
{
public:
   vector<vector<int>> distances;
   int n;
   // Random instance generator for shuffling
   random_device rd;
   mt19937 generator;

   TSP(vector<vector<int>> &d) : distances(d),
```

```
                                n(distances.size()),
                                generator(rd()) {}

vector<int> generateInitialSolution()
{
    vector<int> route(n);
    for (int i = 0; i < n; i++)
    {
        route[i] = i;
    }
    shuffle(route.begin(), route.end(), generator);

    return route;
}

int calcTotalCost(vector<int> &route)
{
    int totalCost = 0;
    for (int i = 0; i < n; i++)
    {
        int a = route[i];
        int b = route[(i + 1) % n];
        totalCost += distances[a][b];
    }
    return totalCost;
}

vector<int> bestNeighbour(vector<int> &currRoute, int &bestCost)
{
    vector<int> &bestRoute = currRoute;
    bestCost = calcTotalCost(currRoute);

    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            vector<int> newRoute = currRoute;
            swap(newRoute[i], newRoute[j]);

            int currCost = calcTotalCost(newRoute);
```

```
                if (currCost < bestCost)
                {
                    bestCost = currCost;
                    bestRoute = newRoute;
                }
            }
        }
        return bestRoute;
    }

    pair<vector<int>, int> hillClimbingAlgo(int maxIterations)
    {

        // start with a random solution

        vector<int> currSolution = generateInitialSolution();
        int currCost = calcTotalCost(currSolution);

        for (int i = 0; i < maxIterations; i++)
        {
            int newCost;

            vector<int> newSolution = bestNeighbour(currSolution, newCost);

            if (newCost >= currCost)
            {
                // optima found break here
                break;
            }
            currSolution = newSolution;
            currCost = newCost;
        }

        return {currSolution, currCost};
    }
};

int main()
{
```

```
    int numCities;
    cout << "Enter the number of cities\n";
    cin >> numCities;

    vector<vector<int>> distMatrix(numCities, vector<int>(numCities));
    cout << "Enter the value of distance between 2 cities, distance matrix
should be symmetric\n";

    for (int i = 0; i < numCities; i++)
    {
        for (int j = 0; j < numCities; j++)
        {
            if (i == j)
                distMatrix[i][j] = 0;

            else
            {
                cout << "Distance from city " << i << " to city " << j <<
": ";
                int d;
                cin >> d;
                distMatrix[i][j] = d;
                cout << "\n";
            }
        }
    }

    TSP tsp_solver(distMatrix);

    cout << "Enter the maximum number of iterations:\n";
    int maxIterations;
    cin >> maxIterations;

    pair<vector<int>, int> res =
tsp_solver.hillClimbingAlgo(maxIterations);

    vector<int> bestRoute = res.first;
    int bestDist = res.second;

    cout << "Best Route is:\n";
```

```
    for (int city : bestRoute)
    {
        cout << city << "-->";
    }

    cout << bestRoute[0] << "\n";

    cout << "Best possible total distance is : " << bestDist << "\n";

    return 0;
}

/*
    It is to note that there can be multiple number of TSP's with same best
cost possible for a given Graph
*/
```

Output :-

The given distance matrix was given as input

0	10	20	30	55
10	0	42	45	20
20	42	0	30	15
30	45	30	0	25
55	20	15	25	0

Run 1 :

```
Enter the maximum number of iterations:
10000
Best Route is:
4-->2-->3-->0-->1-->4
Best possible total distance is : 105
```

Run 2 :

```
Enter the maximum number of iterations:
10000
Best Route is:
3-->0-->1-->4-->2-->3
Best possible total distance is : 105
```

Run 3 :

```
Enter the maximum number of iterations:
10000
Best Route is:
0-->2-->3-->4-->1-->0
Best possible total distance is : 105
```

Question 02:-

In the realm of Artificial Intelligence, contemplate a problem involving two containers of indeterminate capacity, referred to as jugs. One jug has a capacity of 3 units, while the other holds up to 4 units. There are no markings or additional measuring instruments, the objective is to develop a strategic approach to precisely fill the 4 unit jug with 2 units of water. The restriction stipulates the use of solely the aforementioned jugs, excluding any supplementary tools. Both jugs initiate the scenario in an empty state. The aim is to attain the desired water quantity in the 4 unit jug by executing a sequence of permissible operations, including filling, emptying, and pouring water between the jugs. The challenge in this scenario involves crafting an algorithm, such as Depth First Search, to systematically explore and determine the optimal sequence of moves for accomplishing the task while adhering to the defined constraints.

Idea of Depth First Search in this problem

The first thing to notice about this problem is that it asks us to find a sequence of moves such that the second jug (4 unit capacity) gets filled with 2 units of water. To perform moves there are specific restrictions imposed :

1. Fill either of the jugs completely
2. Empty either of the jugs
3. Pour all the water from one jug to another

We can reconstruct this problem as a Finite automaton when the initial state will be <0,0> indicating that both the jugs are empty [<p,q> defines that the jug1 currently has p units of water while the jug2 currently has q units of water. It is to note that anytime, p<=3 and q<=4].
The transition among states are defined by the restriction, and the final state will be the set of states F: [<k,2>: 0<=k<=3].

So the solution boils down to a DFS from the start state and determines whether any of the final states is reachable or not. Important thing to note is that since there are a range of final states, DFS does not guarantee optimal (minimum) number of moves which can be guaranteed by the BFS, where we can keep track of the level in the search tree with the start state at root.

Code :

```
/*
   As the problem statement talks of DFS, we need to think what are the
nodes and what are the edges first.

   A node here as we have seen in DFA can be thought of as a state.

   Here state can be represented as a pair of the current volume of water
in the two jugs

   Jug1 has capacity of 3 units while Jug2 has capacity of 4 units

   The edges between nodes can be thought of as transitions described in
the question:-
       1. Empty jug1
       2. Empty jug2
       3. Fill jug1
       4. Fill jug2
       5. Pour from jug1 to jug2
       6. Pour from jug2 to jug1

   Initial state as per the question is <0,0>

   The question also mentions a target T, which should be finally present
   in the 4 unit jug after all the transitions.

   Final state will be <k,T>. k can be any arbitrary non-negative number
less than 3
```

```
   A simple idea will be to start the DFS from the initial state and find
whether the final state
   is reachable or not.

*/

#include <bits/stdc++.h>

using namespace std;

class DFSSolver
{
public:
   const int capacity1, capacity2;
   const int target;
   // state is a pair<int,int>p, where p.first=jug1 volume and
p.second=jug2 volume
   set<pair<int, int>> visited;                // keep track of visited
states to prevent loops
   map<pair<int, int>, pair<int, int>> parent; // To keep track of the
previous state
   pair<int, int> targetState;
   int bfs_len;

   DFSSolver(int cap1, int cap2, int t) : capacity1(cap1),
capacity2(cap2), target(t)
   {
       bfs_len = 0;
       targetState = {-1, -1};
   }

   vector<pair<int, int>> nextStates(pair<int, int> &currState)
   {
       vector<pair<int, int>> res;

       // fill jug1
       if (currState.first < capacity1)
       {
           res.push_back({capacity1, currState.second});
       }
```

```
        // fill jug2
        if (currState.second < capacity2)
        {
            res.push_back({currState.first, capacity2});
        }

        // empty jug1

        if (currState.first > 0)
        {
            res.push_back({0, currState.second});
        }

        // empty jug2

        if (currState.second > 0)
        {
            res.push_back({currState.first, 0});
        }

        // pour from jug1 to jug2
        if (currState.first > 0 && currState.second < capacity2)
        {
            int pour = min(currState.first, capacity2 - currState.second);
            res.push_back({currState.first - pour, currState.second +
pour});
        }

        // pour from jug2 ro jug1

        if (currState.second > 0 && currState.first < capacity1)
        {
            int pour = min(currState.second, capacity1 - currState.first);
            res.push_back({currState.first + pour, currState.second -
pour});
        }

        return res;
    }
```

```
bool bfs()
{
    queue<pair<int, int>> q;
    pair<int, int> initialState = {0, 0};
    parent.clear();
    visited.clear();

    q.push(initialState);
    visited.insert(initialState);

    int level = 0;
    while (!q.empty())
    {
        int len = q.size();

        for (int i = 0; i < len; i++)
        {
            pair<int, int> curr = q.front();
            q.pop();

            if (curr.second == target)
            {
                targetState = curr;
                bfs_len = level;
                return true;
            }

            for (auto neighbour : nextStates(curr))
            {
                if (visited.find(neighbour) == visited.end())
                {
                    visited.insert(neighbour);
                    parent[neighbour] = curr;
                    q.push(neighbour);
                }
            }
        }
        level++;
    }
```

```
    return false;
}

bool dfs()
{
    stack<pair<int, int>> st;
    pair<int, int> initialState = {0, 0};

    st.push(initialState);
    visited.insert(initialState);

    while (!st.empty())
    {
        pair<int, int> curr = st.top();
        st.pop();

        if (curr.second == target)
        {
            targetState = curr;
            return true;
        }

        for (auto neighbour : nextStates(curr))
        {
            if (visited.find(neighbour) == visited.end())
            {
                visited.insert(neighbour);
                parent[neighbour] = curr;
                st.push(neighbour);
            }
        }
    }
    return false;
}

void printSequence()
{
    vector<pair<int, int>> path;
    pair<int, int> curr = targetState;
```

```
        int len = 0;

        while (!(curr.first == 0 && curr.second == 0))
        {
            len++;
            path.push_back(curr);
            curr = parent[curr];
        }

        path.push_back({0, 0});
        cout << "Solution exists and has " << len << " number of
transitions\n";

        for (int i = path.size() - 1; i >= 0; i--)
        {
            cout << "Step " << path.size() - i << ": ";
            cout << "<" << path[i].first << "," << path[i].second << ">";

            if (i < path.size() - 1)
            {
                pair<int, int> curr = path[i];
                pair<int, int> prev = path[i + 1];

                if (curr.first > prev.first && curr.second == prev.second)
                {
                    cout << " - Fill jug1";
                }
                else if (curr.second > prev.second && curr.first ==
prev.first)
                {
                    cout << " - Fill jug2";
                }
                else if (curr.first < prev.first && curr.second >
prev.second)
                {
                    cout << " - Pour from jug1 to jug2";
                }
                else if (curr.first > prev.first && curr.second <
prev.second)
                {
```

```
                    cout << " - Pour from jug2 to jug1";
                }
                else if (curr.first == 0 && prev.first > 0)
                {
                    cout << " - Empty jug1";
                }
                else if (curr.second == 0 && prev.second > 0)
                {
                    cout << " - Empty jug2";
                }
            }
            cout << "\n";
        }
        cout << "Total Path Cost is:- " << path.size() - 1 << "\n";
    }

    void printSequenceBFS()
    {
        vector<pair<int, int>> path;
        pair<int, int> curr = targetState;

        int len = 0;

        while (!(curr.first == 0 && curr.second == 0))
        {
            len++;
            path.push_back(curr);
            curr = parent[curr];
        }

        path.push_back({0, 0});
        cout << "Solution exists and has " << len << " number of
transitions\n";

        for (int i = path.size() - 1; i >= 0; i--)
        {
            cout << "Step " << path.size() - i << ": ";
            cout << "<" << path[i].first << "," << path[i].second << ">";

            if (i < path.size() - 1)
```

```
            {
                pair<int, int> curr = path[i];
                pair<int, int> prev = path[i + 1];

                if (curr.first > prev.first && curr.second == prev.second)
                {
                    cout << " - Fill jug1";
                }
                else if (curr.second > prev.second && curr.first ==
prev.first)
                {
                    cout << " - Fill jug2";
                }
                else if (curr.first < prev.first && curr.second >
prev.second)
                {
                    cout << " - Pour from jug1 to jug2";
                }
                else if (curr.first > prev.first && curr.second <
prev.second)
                {
                    cout << " - Pour from jug2 to jug1";
                }
                else if (curr.first == 0 && prev.first > 0)
                {
                    cout << " - Empty jug1";
                }
                else if (curr.second == 0 && prev.second > 0)
                {
                    cout << " - Empty jug2";
                }
            }
            cout << "\n";
        }
        cout << "Total Path Cost is:- " << bfs_len << "\n";
    }
};

int main()
{
```

```
    int jug1, jug2, target;
    cout << "Enter the capacity of jug1:\n";
    cin >> jug1;
    cout << "Enter the capacity of jug2:\n";
    cin >> jug2;
    cout << "Enter the target volume in jug2:\n";
    cin >> target;

    DFSSolver solver(jug1, jug2, target);

    cout << "DFS Solution\n";
    if (solver.dfs())
    {
        solver.printSequence();
    }
    else
    {
        cout << "No solution exists!\n";
    }

    cout << "BFS Solution\n";
    if (solver.bfs())
    {
        solver.printSequenceBFS();
    }
    else
    {
        cout << "No solution exists!\n";
    }

    return 0;
}
```

Output :

1. Capacity of jug1 is 3 and that of jug2 is 4. Target is 2 unit in jug2

```
Solution exists and has 6 number of transitions
Step 1: <0,0>
Step 2: <0,4> - Fill jug2
Step 3: <3,1> - Pour from jug2 to jug1
Step 4: <0,1> - Empty jug1
Step 5: <1,0> - Pour from jug2 to jug1
Step 6: <1,4> - Fill jug2
Step 7: <3,2> - Pour from jug2 to jug1
```

2. Capacity of jug1 is 9 and that of jug2 is 5. Target is 3 unit in jug2

```
Solution exists and has 16 number of transitions
Step 1: <0,0>
Step 2: <0,5> - Fill jug2
Step 3: <5,0> - Pour from jug2 to jug1
Step 4: <5,5> - Fill jug2
Step 5: <9,1> - Pour from jug2 to jug1
Step 6: <0,1> - Empty jug1
Step 7: <1,0> - Pour from jug2 to jug1
Step 8: <1,5> - Fill jug2
Step 9: <6,0> - Pour from jug2 to jug1
Step 10: <6,5> - Fill jug2
Step 11: <9,2> - Pour from jug2 to jug1
Step 12: <0,2> - Empty jug1
Step 13: <2,0> - Pour from jug2 to jug1
Step 14: <2,5> - Fill jug2
Step 15: <7,0> - Pour from jug2 to jug1
Step 16: <7,5> - Fill jug2
Step 17: <9,3> - Pour from jug2 to jug1
```

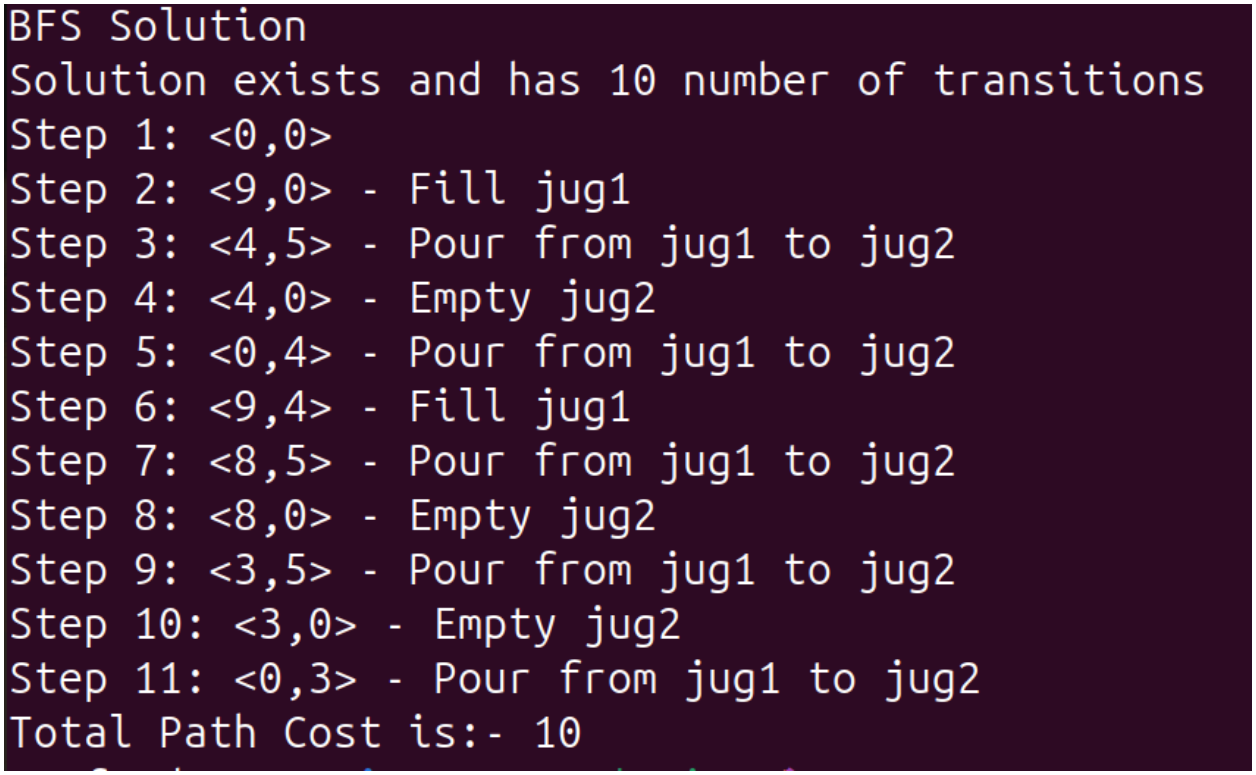
One thing to note about the difference in problems 1 and 2 is that since 1 uses a stochastic approach to solve the problem, the optimal sequence and sometimes even the optimal value is different for the same test case, while for problem 2, the deterministic DFA transitions along with deterministic start state ensure that the sequence remains same all the time for a given test case

.

DFS Solution

```
Enter the capacity of jug1:
9
Enter the capacity of jug2:
5
Enter the target volume in jug2:
3
DFS Solution
Solution exists and has 16 number of transitions
Step 1: <0,0>
Step 2: <0,5> - Fill jug2
Step 3: <5,0> - Pour from jug2 to jug1
Step 4: <5,5> - Fill jug2
Step 5: <9,1> - Pour from jug2 to jug1
Step 6: <0,1> - Empty jug1
Step 7: <1,0> - Pour from jug2 to jug1
Step 8: <1,5> - Fill jug2
Step 9: <6,0> - Pour from jug2 to jug1
Step 10: <6,5> - Fill jug2
Step 11: <9,2> - Pour from jug2 to jug1
Step 12: <0,2> - Empty jug1
Step 13: <2,0> - Pour from jug2 to jug1
Step 14: <2,5> - Fill jug2
Step 15: <7,0> - Pour from jug2 to jug1
Step 16: <7,5> - Fill jug2
Step 17: <9,3> - Pour from jug2 to jug1
Total Path Cost is:- 16
```

BFS Solution



BFS gives a more optimal solution than DFS for this case

Question 03:-

Develop a comprehensive program that effectively addresses a puzzle problem. The puzzle involves a 3x3 grid with eight numbered tiles and an empty space (Given in the diagram). The task is to create a program that can systematically rearrange the tiles, around the empty cells, to reach to the predefined goal state from the initial configuration adhering to the constraints of permissible moves.

a. Use two different heuristic functions: one, the total count of the number of misplaced cells to reach to the goal state, second, consider the Manhattan distance as a heuristic function to determine the distance to reach to the goal state.

1	2	3
8		4
7	6	5

Initial State

2	8	1
	4	3
7	6	5

Goal State

Here we have to use the A* search algorithm, where we expand the node of a state that has the optimal f(n) value, where f(n)=g(n)+h(n),
g:- Actual cost to reach the state from the start state
h:- Predictive(Heuristic) cost to reach the goal state from the current state.

State in this problem can be described as a tuple containing 2 things:-

A. Current Board Formation
B. Position of the empty cell

We stop the algorithm whenever we find the goal state. We maintain a priority queue to order the states based on their f(n) values. Optimality is guaranteed when the heuristic function (h(n)) suffices admissibility

condition, that is it never overestimates or underestimates the actual cost.

Code:-

```
#include <bits/stdc++.h>

using namespace std;

/*
   State information contains the :-
       1. board
       2. empty coordinates
       3. g(state)
       4. h(state)
       5. path to reach the goal state

*/
struct State
{
   vector<vector<int>> board;
   int emptyX, emptyY;    // empty cell coordinates
   int g, h;              // g: cost to reach this state, h: heuristic
value
   vector<State *> path; // path to reach the goal state

   State(vector<vector<int>> b) : board(b), g(0), h(0)
   {
       // Find empty cell position
       for (int i = 0; i < 3; i++)
       {
           for (int j = 0; j < 3; j++)
           {
               if (board[i][j] == 0)
               {
                   emptyX = i;
                   emptyY = j;
               }
```

```
            }
        }
    }

    bool operator<(const State &other) const
    {
        return (g + h) > (other.g + other.h); // For priority queue
    }
};

// Calculating the number of misplaced tiles (Heuristic)
int getMisplacedCount(const vector<vector<int>> &current, const
vector<vector<int>> &goal)
{
    int count = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (current[i][j] != 0 && current[i][j] != goal[i][j])
            {
                count++;
            }
        }
    }
    return count;
}

// Calculating the Manhattan distance (Heuristic)
int getManhattanDistance(const vector<vector<int>> &current, const
vector<vector<int>> &goal)
{
    int distance = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (current[i][j] != 0)
            {
                // Find this number's position in goal state
```

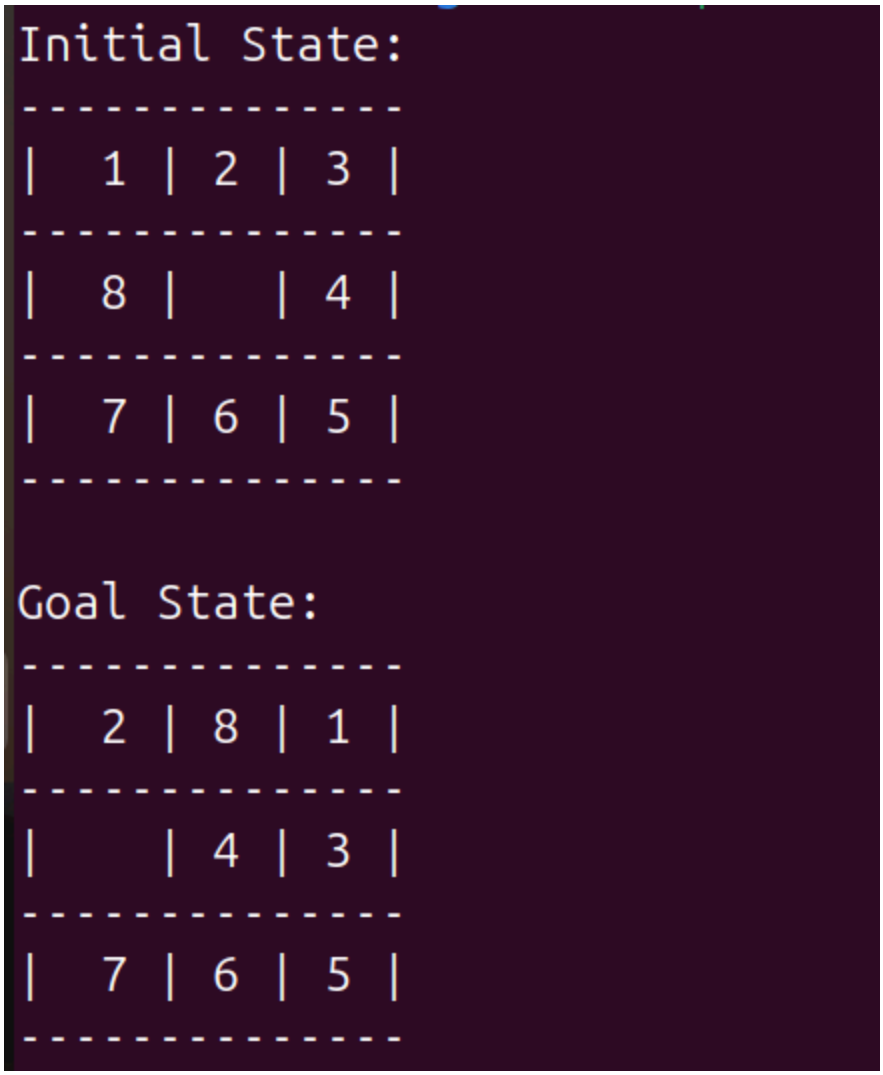
```
                for (int x = 0; x < 3; x++)
                {
                    for (int y = 0; y < 3; y++)
                    {
                        if (goal[x][y] == current[i][j])
                        {
                            distance += abs(x - i) + abs(y - j);
                        }
                    }
                }
            }
        }
    }
    return distance;
}

// Flatten the board to string for checking visited states
string boardToString(const vector<vector<int>> &board)
{
    string s;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            s += to_string(board[i][j]);
    return s;
}

// MOve validity
bool isValid(int x, int y)
{
    return x >= 0 && x < 3 && y >= 0 && y < 3;
}

// Solve 8-puzzle using A* search algorithm
vector<State *> solvePuzzle(const vector<vector<int>> &initial, const
vector<vector<int>> &goal, bool useManhattan)
{
    // Possible moves (up, right, down, left)
    int dx[] = {-1, 0, 1, 0};
    int dy[] = {0, 1, 0, -1};
```

```
    priority_queue<State> pq;
    set<string> visited;

    State *initialState = new State(initial);
    initialState->h = useManhattan ? getManhattanDistance(initial, goal) :
getMisplacedCount(initial, goal);

    pq.push(*initialState);
    visited.insert(boardToString(initial));

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

        // Check if goal reached
        if (current.board == goal)
        {
            return current.path;
        }

        // All possible moves for the empty cell
        for (int i = 0; i < 4; i++)
        {
            int newX = current.emptyX + dx[i];
            int newY = current.emptyY + dy[i];

            if (isValid(newX, newY))
            {
                vector<vector<int>> newBoard = current.board;
                swap(newBoard[current.emptyX][current.emptyY],
                     newBoard[newX][newY]);

                string boardStr = boardToString(newBoard);
                if (visited.find(boardStr) == visited.end())
                {
                    State *newState = new State(newBoard);
                    newState->g = current.g + 1; // actual cost (edge cost
is 1 uniform)
                    // heuristic cost
```

```
                    newState->h = useManhattan ?
getManhattanDistance(newBoard, goal) : getMisplacedCount(newBoard, goal);
                    newState->path = current.path;
                    newState->path.push_back(newState);

                    pq.push(*newState);
                    visited.insert(boardStr);
                }
            }
        }
    }

    return vector<State *>(); // No solution found
}

// Print board state
void printBoard(const vector<vector<int>> &board)
{
    cout << "--------------\n";
    for (int i = 0; i < 3; i++)
    {
        cout << "| ";
        for (int j = 0; j < 3; j++)
        {
            if (board[i][j] == 0)
                cout << "   |";
            else
                cout << " " << board[i][j] << " |";
        }
        cout << "\n--------------\n";
    }
    cout << "\n";
}

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

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

    cout << "Initial State:\n";
    printBoard(initial);
    cout << "Goal State:\n";
    printBoard(goal);

    cout << "Solving using Misplaced Tiles heuristic...\n";
    vector<State *> solution1 = solvePuzzle(initial, goal, false);
    if (!solution1.empty())
    {
        cout << "Solution found in " << solution1.size() << " moves!\n";
        for (State *state : solution1)
        {
            printBoard(state->board);
        }
    }

    cout << "Solving using Manhattan Distance heuristic...\n";
    vector<State *> solution2 = solvePuzzle(initial, goal, true);
    if (!solution2.empty())
    {
        cout << "Solution found in " << solution2.size() << " moves!\n";
        for (State *state : solution2)
        {
            printBoard(state->board);
        }
    }

    return 0;
}
```

Output :-

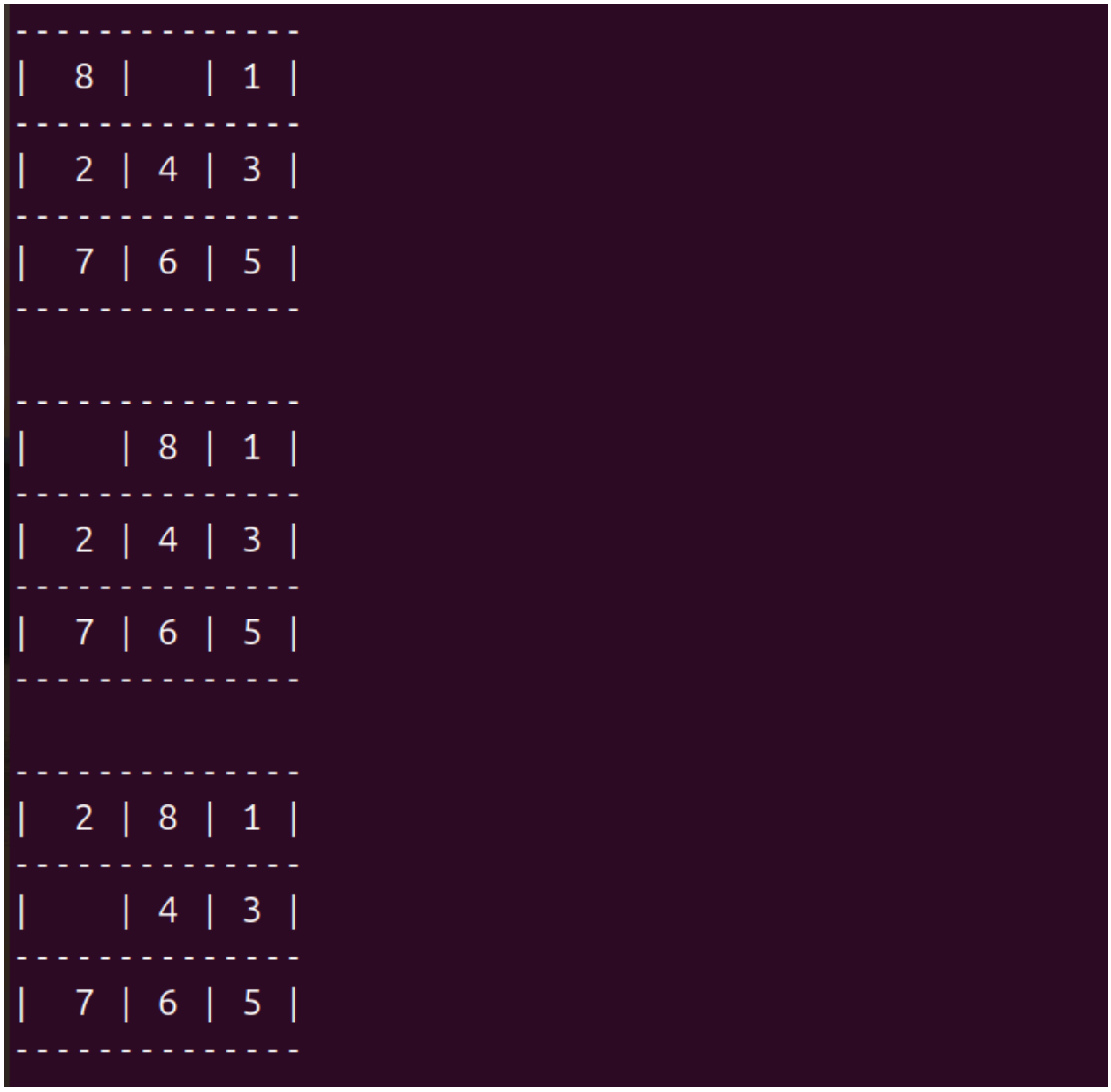


Misplaced Tiles Heuristic

First Few Moves

```
Solving using Misplaced Tiles heuristic...
Solution found in 9 moves!
--------------
|  1 |    | 3 |
--------------
|  8 | 2 | 4 |
--------------
|  7 | 6 | 5 |
--------------

--------------
|    | 1 | 3 |
--------------
|  8 | 2 | 4 |
--------------
|  7 | 6 | 5 |
--------------
```

Last Moves



Manhattan Distance Heuristic :

First Few Moves

```
Solving using Manhattan Distance heuristic...
Solution found in 9 moves!
--------------
|  1 |    | 3 |
--------------
|  8 | 2 | 4 |
--------------
|  7 | 6 | 5 |
--------------

--------------
|    | 1 | 3 |
--------------
|  8 | 2 | 4 |
--------------
|  7 | 6 | 5 |
--------------

--------------
|  8 | 1 | 3 |
--------------
|    | 2 | 4 |
--------------
|  7 | 6 | 5 |
--------------
```

Last Few Moves