NAME: AVINASH SHELKE

ROLL NO: 324048

PRN NO: 22011002

DIVISION: D

BATCH: E9

ASSIGNMENT 1

Problem: - Implement A* algorithm for 8 puzzle.

Theory: -

Rules for solving the puzzle.

Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions viz.,

1. Up
2. Down
3. Right or
4. Left

The empty space cannot move diagonally and can take only one step at a time

BFS Algorithm:

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2

(Waiting state)

[END OF LOOP]

Step 6: EXIT

Code (Screenshot):

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

struct Node *CreateNewNode(int data[3][3]);
void addChild(Node *node, int data[3][3]);
struct Node
{
    int data[3][3];
    vector<Node *> children;
};
int cnt=0;
Node *root;
Node *temp = root;
vector<Node *> AllNodes;
int flag1 = 0;
queue<Node *>Que;

void get_Co_Zero(int mat[3][3], int *zerx, int *zery)
{
    int flag = 0;
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++){
            if (mat[i][j] == 0){
                *zerx = i;
                *zery = j;
                flag = 1;
                break;
            }
        }
        if (flag == 1){
            break;
        }
    }
}
bool isVisited(Node *node)
```

```
{
    int checkflag = 0;
    for (int i = 0; i < AllNodes.size(); i++){
        // cout << "checkflag===" << checkflag;
        if (checkflag == 0){
            checkflag = 1;
            // cout << "checkflag1===" << checkflag;
            for (int j = 0; j < 3; j++)
            {
                for (int k = 0; k < 3; k++)
                {
                    if (AllNodes[i]->data[j][k] != node->data[j][k])
                    {
                        checkflag = 0;
                        break;
                    }
                }
                if (checkflag == 0)
                {
                    break;
                }
            }
        }
        else
        {
            break;
        }
    }
    if (checkflag == 0)
    {
        AllNodes.push_back(node);
        return false;
    }
    return true;
}

// void misplaced_tile(Node * node){
//     int goal[3][3]= {
//         {1,2,3},
//         {4,5,6},
//         {7,8,0}};
//     int cnt=0;
//     for (int i = 0; i < 3; i++){
//         for (int j = 0; j < 3; j++){
//             if (node->data[i][j] != goal[i][j]){
//                 cnt++;
//             }
//         }
```

```
//     }
//     cout<<"Misplaced tiles are: "<<cnt<<endl;
// }

void up(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx - 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "up"<<endl;
        addChild(temp, newm);
        cnt++;
        Que.push(updatedmat);
    }
}
void down(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx + 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
```

```
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "down"<<endl;
        addChild(temp, newm);
        cnt++;
        Que.push(updatedmat);
    }
}
void right(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery + 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);

    if (!isVisited(updatedmat))
    {
        cout << "right"<<endl;
        addChild(temp, newm);
        cnt++;
        Que.push(updatedmat);
    }
}
void left(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
```

```
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery - 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "left"<<endl;
        addChild(temp, newm);
        cnt++;
        Que.push(updatedmat);
    }
}
void decide(Node *temp, int mat[3][3], int zerx, int zery)
{
    switch (zerx)
    {
    case 0:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        default:
            break;
        }
        break;
    case 1:
        switch (zery)
        {
        case 0:
```

```
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:

            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);
            break;
        default:
            break;
        }
        break;

    case 2:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 1:

            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        default:
            break;
        }
        break;
```

```
    default:
        break;
    }
}

void printData(Node *node)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << node->data[i][j]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
    // misplaced_tile(node);
}

bool compareTwoArrays(Node *node)
{
    int goal[3][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++){
        for (int j = 0; j < 3; j++){
            if (node->data[i][j] != goal[i][j]){
                 return false;
            }
        }
    }
    cout << "Reached"<<endl;
    cout << "Total state space= "<<cnt;
    return true;
}

void checkAllPossibilties(Node *temp, int *zerx, int *zery)
{
    while(!Que.empty())
    {
        // cout << "comparing=" << !compareTwoArrays(temp->children[i]);
        if (!compareTwoArrays(Que.front())&&flag1==0)
        {
            get_Co_Zero(Que.front()->data, zerx, zery);
            decide(Que.front(), Que.front()->data, *zerx, *zery);
            for (int j = 0; j < Que.front()->children.size(); j++)
            {
                printData(Que.front()->children[j]);
```

```
                if (compareTwoArrays(Que.front()->children[j]))
                {
                    flag1 = 1;
                    break;
                }
                if(flag1==1){
                    break;
                }
            }
            cout<<"\n";
        }
        Que.pop();
    }


}

int main()
{
    int zerx, zery;
    int mat[3][3] = {{1,2,3},
                     {0,4,6},
                     {7,5,8}};
    // for (int i = 0; i < 3; i++)
    // {
    //     for (int j = 0; j < 3; j++)
    //     {
    //         cin >> mat[i][j];
    //     }
    // }
    root = CreateNewNode(mat);
    temp = root;
    AllNodes.push_back(temp);
    // checkAllPossibilties(temp, &zerx, &zery);

    cout << "root"<<endl;
    for (int j = 0; j < 3; j++)
    {
        for (int k = 0; k < 3; k++)
        {
            cout << root->data[j][k]<<" ";
        }
        cout<<endl;
    }
    cout<<endl;
    get_Co_Zero(mat, &zerx, &zery);
    decide(temp, mat, zerx, zery);
    for (int i = 0; i < root->children.size(); i++)
```

```
    {
        printData(root->children[i]);
    }
    cout<<"\n";
    checkAllPossibilties(temp, &zerx, &zery);

    return 0;
}
// function to create new node
Node *CreateNewNode(int data[3][3])
{
    Node *newNode = new Node();
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            newNode->data[i][j] = data[i][j];
        }
    }
    return newNode;
}

// function to add a child to a specific node
void addChild(Node *node, int data[3][3])
{
    Node *newNode = CreateNewNode(data);
    node->children.push_back(newNode);
}
```

Output (Screenshot) 1:

```
root
1 2 3
0 4 6
7 5 8

right
up
down
1 2 3
4 0 6
7 5 8

0 2 3
1 4 6
7 5 8

1 2 3
7 4 6
0 5 8


right
up
down
1 2 3
4 6 0
7 5 8

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

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


right
2 0 3
1 4 6
7 5 8


right
1 2 3
7 4 6
5 0 8


up
down
1 2 0
4 6 3
7 5 8

1 2 3
4 6 8
7 5 0


right
left
1 3 0
4 2 6
7 5 8
```

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


right
left
1 2 3
4 5 6
7 8 0

Reached
Total state space= 14
Reached
Total state space= 14
```

-> Total state space generated are **14.**

Output (Screenshot) 2:
For input:

```
int zerx, zery;
int mat[3][3] = {{3,1,0},
                 {4,2,6},
                 {7,8,5}};
```

```
root
3 1 0
4 2 6
7 8 5

left
down
3 0 1
4 2 6
7 8 5

3 1 6
4 2 0
7 8 5


left
down
0 3 1
4 2 6
7 8 5

3 2 1
4 0 6
7 8 5


left
down
3 1 6
4 0 2
7 8 5
```

```
down
4 3 1
0 2 6
7 8 5


right
left
down
3 2 1
4 6 0
7 8 5

3 2 1
0 4 6
7 8 5

3 2 1
4 8 6
7 0 5


left
up
down
3 1 6
0 4 2
7 8 5

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

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


left
3 1 6
4 2 5
7 0 8


right
down
4 3 1
2 0 6
7 8 5

4 3 1
7 2 6
0 8 5


up
down
3 2 0
4 6 1
7 8 5

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

-> The problem has solution at depth of 18. Therefore, BFS got stuck after exploring few depths.

Time complexity: O(b^d)

b= branch factor, d=depth of tree.

Space Complexity: O(b^d)

b= branch factor, d=depth of tree.

DFS Algorithm:

It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.
6. Repeat steps 2, 3, and 4 until the stack is empty.

Code (Screenshot):

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

struct Node *CreateNewNode(int data[3][3]);
void addChild(Node *node, int data[3][3]);
struct Node
{
    int data[3][3];
    vector<Node *> children;
};
Node *root;
Node *temp = root;
int cnt=0;
vector<Node *> allNodes;
int flag1 = 0;

void get_Co_Zero(int mat[3][3], int *zerx, int *zery)
{
    int flag = 0;
    for (int i = 0; i < 3; i++)
```

```
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] == 0)
            {
                *zerx = i;
                *zery = j;
                flag = 1;
                break;
            }
        }
        if (flag == 1)
        {
            break;
        }
    }
}
bool isVisited(Node *node)
{
    int flagCheck = 0;
    for (int i = 0; i < allNodes.size(); i++)
    {
        // cout << "flagCheck===" << flagCheck;

        if (flagCheck == 0)
        {
            flagCheck = 1;
            // cout << "flagCheck1===" << flagCheck;
            for (int j = 0; j < 3; j++)
            {
                for (int k = 0; k < 3; k++)
                {
                    if (allNodes[i]->data[j][k] != node->data[j][k])
                    {
                        // cout << "ak";
                        flagCheck = 0;
                        break;
                    }
                    // else
                    // {
                    //     cout << "flagCheck2===" << flagCheck;
                    //     cout << "\nalli=" << allNodes[i]->data[j][k];
                    //     cout << "\nnode=" << node->data[j][k];
                    // }
                }
                if (flagCheck == 0)
                {
                    break;
```

```
                }
            }
        }
        else
        {
            break;
        }
    }
    if (flagCheck == 0)
    {
        allNodes.push_back(node);
        return false;
    }
    return true;
}

void up(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx - 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "up"<<endl;
        cnt++;
        addChild(temp, newm);
    }
}
void down(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 2)
    {
```

```
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx + 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "down"<<endl;
        cnt++;
        addChild(temp, newm);
    }
}
void right(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery + 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);

    if (!isVisited(updatedmat))
    {
        cout << "right"<<endl;
        cnt++;
        addChild(temp, newm);
    }
}
void left(Node *temp, int mat[3][3], int zerx, int zery)
```

```
{
    int newm[3][3];
    if (zery != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery - 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "left"<<endl;
        cnt++;
        addChild(temp, newm);
    }
}
void decide(Node *temp, int mat[3][3], int zerx, int zery)
{
    switch (zerx)
    {
    case 0:
        switch (zery)
        {
        case 0:
            down(temp, mat, zerx, zery);
            right(temp, mat, zerx, zery);

            break;
        case 1:
            left(temp, mat, zerx, zery);
            right(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
```

```
        default:
            break;
        }
        break;
    case 1:
        switch (zery)
        {
        case 0:

            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 1:
            up(temp, mat, zerx, zery);
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:

            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);
            break;
        default:
            break;
        }
        break;

    case 2:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 1:
            up(temp, mat, zerx, zery);
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
```

```
            up(temp, mat, zerx, zery);

            break;
        default:
            break;
        }
        break;

    default:
        break;
    }
}
void printData(Node *node)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << node->data[i][j];
        }
        cout<<endl;
    }
    cout << endl;
}
bool compareTwoArrays(Node *node)
{
    int goal[3][3] = {{1, 2, 3},
                      {4, 5, 6},
                      {7, 8, 0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            // cout << "\nnodeij=" << node->data[i][j];
            // cout << "\ngoalij=" << goal[i][j];
            if (node->data[i][j] != goal[i][j])
            {

                return false;
            }
        }
    }
    cout << "Reached"<<endl;
    cout<<"total state space= "<<cnt;
    return true;
}

void checkAllPossibilties(Node *temp, int *zerx, int *zery)
```

```
{
    for (int i = 0; i < temp->children.size(); i++)
    {
        // cout << "comparing=" << !compareTwoArrays(temp->children[i]);
        if (!compareTwoArrays(temp->children[i]))
        {
            get_Co_Zero(temp->children[i]->data, zerx, zery);
            decide(temp->children[i], temp->children[i]->data, *zerx, *zery);
            for (int j = 0; j < temp->children[i]->children.size(); j++)
            {

                printData(temp->children[i]->children[j]);
                if (compareTwoArrays(temp->children[i]->children[j]))
                {
                    flag1 = 1;
                    break;
                }
            }
        }
        else if (flag1 == 1)
        {
            // exit(0);
            // flag1 = 1;
            break;
        }
    }
    if (flag1 == 0)
    {
        for (int i = 0; i < temp->children.size(); i++)
        {
            temp = temp->children[i];
            checkAllPossibilties(temp, zerx, zery);
        }
    }
}

int main()
{
    int zerx, zery;
    int mat[3][3]=  {{1,2,3},
                     {0,4,6},
                     {7,5,8}};
    // for (int i = 0; i < 3; i++)
    // {
    //     for (int j = 0; j < 3; j++)
    //     {
    //         cin >> mat[i][j];
    //     }
```

```
    // }
    root = CreateNewNode(mat);
    temp = root;
    allNodes.push_back(temp);
    checkAllPossibilties(temp, &zerx, &zery);

    for (int i = 0; i < allNodes.size(); i++)
    {
        cout << "\nallnodes"<<endl;
        for (int j = 0; j < 3; j++)
        {
            for (int k = 0; k < 3; k++)
            {
                cout << allNodes[i]->data[j][k];
            }
            cout<<endl;
        }
    }
    get_Co_Zero(mat, &zerx, &zery);
    decide(temp, mat, zerx, zery);
    // cout << root->children.size();
    for (int i = 0; i < root->children.size(); i++)
    {
        printData(root->children[i]);
    }
    checkAllPossibilties(temp, &zerx, &zery);

    // get_Co_Zero(root->children[0]->data,&zerx,&zery);
    // decide(root->children[0]->data,zerx,zery);
    // for(int i=0;i<root->children[0]->children.size();i++){
    //     printData(root->children[0]->children[i]);
    // }

    return 0;
}
// function to create new node
Node *CreateNewNode(int data[3][3])
{
    Node *newNode = new Node();
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            newNode->data[i][j] = data[i][j];
        }
    }
    return newNode;
}
```

```
// function to add a child to a specific node
void addChild(Node *node, int data[3][3])
{
    Node *newNode = CreateNewNode(data);
    node->children.push_back(newNode);
}
```

Output (Screenshot) 1:

```
allnodes
123
046
758
right
up
down
123
406
758

023
146
758

123
746
058

up
right
down
103
426
758

123
460
758

123
456
708

right
203
146
758
```

```
 right
 123
 746
 508

 left
 right
 013
 426
 758

 130
 426
 758

 up
 down
 120
 463
 758

 123
 468
 750

 right
 left
 123
 456
 780
```

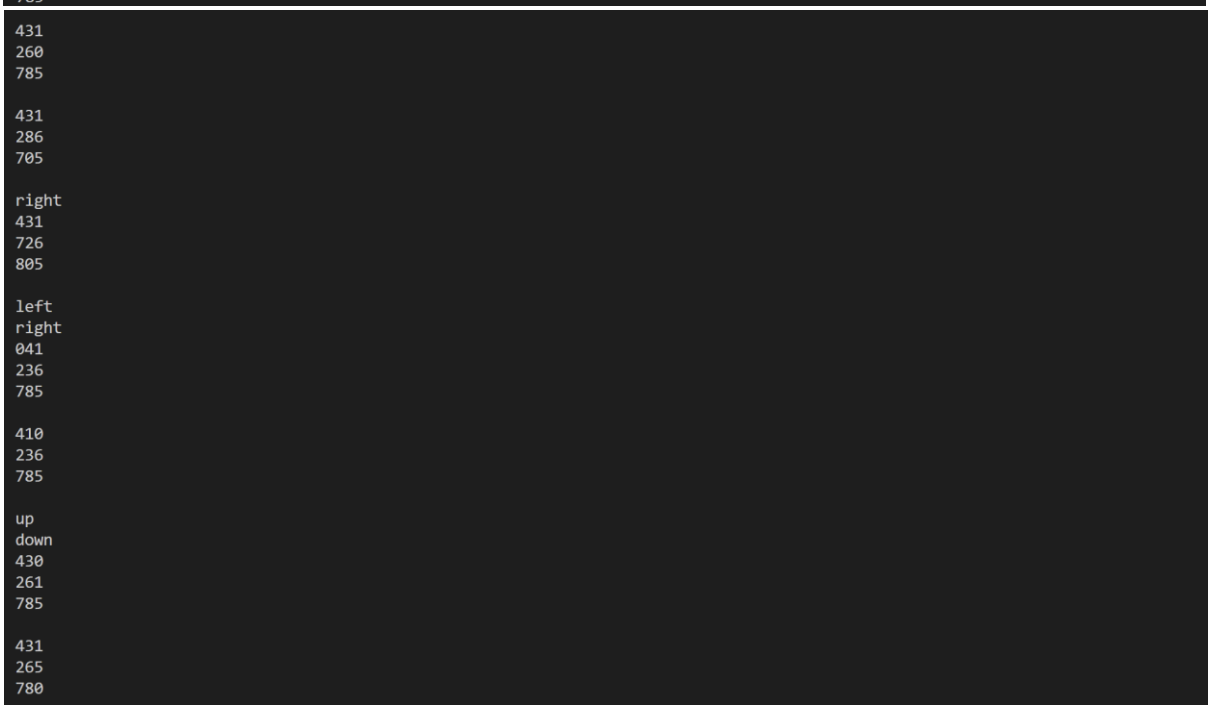
```
Reached
total state space= 14
```

-> Total state space generated are **14.**

Output (Screenshot) 2:
For input:

```
int zerx, zery;
int mat[3][3] = {{3,1,0},
                 {4,2,6},
                 {7,8,5}};
```

```
allnodes
310
426
785
left
down
301
426
785

316
420
785

left
down
031
426
785

321
406
785

left
down
316
402
785

316
425
780
```



-> The problem has solution at depth of 18. Therefore, DFS got stuck after exploring some depths.

Time complexity: O(b^d)

b= branch factor, d=depth of tree.

Space Complexity: O(b^d)

b= branch factor, d=depth of tree.

A* Algorithm:

A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use.

The key feature of the A* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time. It also has a list that holds all the nodes that are left to be explored and it chooses the most optimal node from this list, thus saving time not exploring unnecessary or less optimal nodes.

So we use two lists namely ‘open list’ and ‘closed list‘ the open list contains all the nodes that are being generated and are not existing in the closed list and each node explored after it’s neighboring nodes are discovered is put in the closed list and the neighbors are put in the open list this is how the nodes expand. Each node has a pointer to its parent so that at any given point it can retrace the path to the parent. Initially, the open list holds the start(Initial) node. The next node chosen from the open list is based on its f score, the node with the least f score is picked up and explored.

f-score = h-score + g-score

A* uses a combination of heuristic value (h-score: how far the goal node is) as well as the g-score (i.e., the number of nodes traversed from the start node to current node).

In our 8-Puzzle problem, we can define the h-score as the number of misplaced tiles by comparing the current state and the goal state or summation of the Manhattan distance between misplaced nodes.

g-score will remain as the number of nodes traversed from a start node to get to the current node.

We can calculate the h-score by comparing the initial(current) state and goal state and counting the number of misplaced tiles.

Thus, h-score = 5 and g-score = 0 as the number of nodes traversed from the start node to the current node is 0.

How A* solves the 8-Puzzle problem.

We first move the empty space in all the possible directions in the start state and calculate the f-score for each state. This is called expanding the current state.

After expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list. A state with the least f-score is selected and expanded again. This process continues until the goal state occurs as the current state. Basically, here we are providing the algorithm a measure to choose its actions. The algorithm chooses the best possible action and proceeds in that path.

This solves the issue of generating redundant child states, as the algorithm will expand the node with the least f-score.

A* Algorithm (simple heuristics – misplaced tiles):

Heuristics (misplaced tiles):

```
int findFn(int mat[3][3]){
    int misplaced_tile=0;
    int goal[3][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] != goal[i][j])
            {
                misplaced_tile++;
            }
        }
    }
    return misplaced_tile;
}
```

Code (Screenshot):

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

struct Node *CreateNewNode(int data[3][3]);
void addChild(Node *node, int data[3][3]);
struct Node
{
    int data[3][3];
    // int fn;
    vector<Node *> children;
};
Node *root;
Node *temp = root;
vector<Node *> allNodes;
int flag1 = 0;
queue<Node *>Que;
// queue<Node *>Q;
int cnt=0;
// vector<Node *> childNodes;
map<int, Node *> openList;
```

```
vector<Node *> closedList;
int findFn(int mat[3][3]){
    int misplaced_tile=0;
    int goal[3][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] != goal[i][j])
            {
                misplaced_tile++;
            }
        }
    }
    return misplaced_tile;
}
void get_Co_Zero(int mat[3][3], int *zerx, int *zery)
{
    int flag = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] == 0)
            {
                *zerx = i;
                *zery = j;
                flag = 1;
                break;
            }
        }
        if (flag == 1)
        {
            break;
        }
    }
}
bool isVisited(Node *node)
{
    int flagCheck = 0;
    for (int i = 0; i < allNodes.size(); i++)
    {
        // cout << "flagCheck===" << flagCheck;
        // cout << "\naniket\n";
        if (flagCheck == 0)
        {
```

```
            flagCheck = 1;
            // cout << "flagCheck1===" << flagCheck;
            for (int j = 0; j < 3; j++)
            {
                for (int k = 0; k < 3; k++)
                {
                    if (allNodes[i]->data[j][k] != node->data[j][k])
                    {
                        flagCheck = 0;
                        break;
                    }
                }
                if (flagCheck == 0)
                {
                    break;
                }
            }
        }
        else
        {
            break;
        }
    }
    if (flagCheck == 0)
    {
        allNodes.push_back(node);
        return false;
    }
    return true;
}

void up(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx - 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
```

```
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "up";
        addChild(temp, newm);
        // Q.push(updatedmat);
        cnt++;
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));

    }
    else{
        cout<<"visited!!!";
    }
}
void down(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx + 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "down";
        addChild(temp, newm);
        // Q.push(updatedmat);
        cnt++;
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));

    }
    else{
        cout<<"visited!!!";
    }
}
void right(Node *temp, int mat[3][3], int zerx, int zery)
```

```
{
    int newm[3][3];
    if (zery != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery + 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);

    if (!isVisited(updatedmat))
    {
        cout << "right";
        addChild(temp, newm);
        // Q.push(updatedmat);
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));
        cnt++;
    }
    else{
        cout<<"visited!!!";
    }
}
void left(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery - 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
```

```
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "left";
        addChild(temp, newm);
        // Q.push(updatedmat);
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));
        cnt++;
    }
    else{
        cout<<"visited!!!";
    }
}
void decide(Node *temp, int mat[3][3], int zerx, int zery)
{
    switch (zerx)
    {
    case 0:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        default:
            break;
        }
        break;
    case 1:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);
```

```
            break;
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:

            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);
            break;
        default:
            break;
        }
        break;

    case 2:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 1:

            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        default:
            break;
        }
        break;

    default:
        break;
    }
```

```
}
void printData(Node *node)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << node->data[i][j];
        }
        cout<<endl;
    }
    cout << endl;
}

bool compareTwoArrays(Node *node)
{
    int goal[3][3] = {{1, 2, 3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            // cout << "\nnodeij=" << node->data[i][j];
            // cout << "\ngoalij=" << goal[i][j];
            if (node->data[i][j] != goal[i][j])
            {

                return false;
            }
        }
    }
    cout << "Reached"<<endl;
    cout<<"Total state space="<<cnt;
    return true;
}

void checkAllPossibilties(Node *temp, int *zerx, int *zery)
{
    // while(!Q.empty())
    // {
        // cout << "comparing=" << !compareTwoArrays(temp->children[i]);
        while (!compareTwoArrays((*openList.begin()).second)&&flag1==0)
        {
            Node *minNode=(*openList.begin()).second;
            int fn=(*openList.begin()).first;
            Que.push((*openList.begin()).second);
            openList.erase(fn);
```

```
            while(fn==(*openList.begin()).first){
                Que.push((*openList.begin()).second);
                openList.erase(fn);
            }
            // Q.push(minNode);
            openList.clear();
            while(!Que.empty()){
                minNode=Que.front();
                get_Co_Zero(minNode->data, zerx, zery);
            cout<<"\nminnode=";
            printData(minNode);
            cout<<"\n";
            cout<<"children\n";
            decide(minNode, minNode->data, *zerx, *zery);
            // sort(childNodes.begin(),childNodes.end());

            for (int j = 0; j < minNode->children.size(); j++)
            {
                printData(minNode->children[j]);
                cout<<"fn="<<findFn(minNode->children[j]->data)<<"\n";
                if (compareTwoArrays(minNode->children[j]))
                {
                    flag1 = 1;
                    break;
                }

            }
            cout<<"\n";
            if(flag1==1){
                    break;
                }
            Que.pop();
            }
        }

        // Q.pop();
    // }


}

int main()
{
    int zerx, zery;
    int mat[3][3] = {{1,2,3},
                     {0,4,6},
                     {7,5,8}};
    // for (int i = 0; i < 3; i++)
```

```
    // {
    //     for (int j = 0; j < 3; j++)
    //     {
    //         cin >> mat[i][j];
    //     }
    // }
    root = CreateNewNode(mat);
    temp = root;
    allNodes.push_back(temp);
    // checkAllPossibilties(temp, &zerx, &zery);

        cout << "root";
        for (int j = 0; j < 3; j++)
        {
            for (int k = 0; k < 3; k++)
            {
                cout << root->data[j][k];
            }
        }
        cout<<"\n";
    get_Co_Zero(mat, &zerx, &zery);
    cout<<"\nchildren\n";
    decide(temp, mat, zerx, zery);

    for (int i = 0; i < root->children.size(); i++)
    {
        printData(root->children[i]);
        cout<<"fn="<<findFn(root->children[i]->data)<<"\n";
    }
    cout<<"\n";
    checkAllPossibilties(temp, &zerx, &zery);

    return 0;
}
// function to create new node
Node *CreateNewNode(int data[3][3])
{
    Node *newNode = new Node();
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            newNode->data[i][j] = data[i][j];
        }
    }
    // newNode->fn=findFn(data);
    return newNode;
}
```

```cpp
// function to add a child to a specific node
void addChild(Node *node, int data[3][3])
{
    Node *newNode = CreateNewNode(data);
    node->children.push_back(newNode);
}
```

Output (Screenshot) 1:

```
root
123
046
758


children
right
up
down
123
406
758

fn=3
023
146
758

fn=5
123
746
058

fn=5


minnode=
123
406
758


children
right
Already visited!!!
up
down
123
460
758
```

```
fn=4
103
426
758

fn=4
123
456
708

fn=2


minnode=
123
456
708


children
right
left
Already visited!!!
123
456
780

fn=0
Reached
Total state space=8
Reached
Total state space=8
```

-> Total state space generated are **8.**

Output (Screenshot) 2:

For input:

```
int zerx, zery;
int mat[3][3] = {{3,1,0},
                 {4,2,6},
                 {7,8,5}};
```

```
root
310
426
785


children
left
down
301
426
785

fn=5
316
420
785

fn=6


minnode=
301
426
785
```

```
children
Already visited!!!
left
down
031
426
785

fn=5
321
406
785

fn=4


minnode=
321
406
785


children
right
left
Already visited!!!
down
321
460
785

fn=5
321
046
785
```

```
fn=5
321
486
705

fn=5


minnode=
321
460
785


children
Already visited!!!
up
down
320
461
785

fn=5
321
465
780

fn=4


minnode=
321
465
780
```

…….After a lot exploration……..

```
fn=8


minnode=
174
520
863


children
Already visited!!!
up
down
170
524
863

fn=8
174
523
860

fn=7


minnode=
174
523
860


children
Already visited!!!
Already visited!!!
```

-> It stops finally, after exploring certain depth. Final state **not achieved**.

Time complexity: O(b^d)

b= branch factor, d=depth of tree.

Space Complexity: O(b^d)

b= branch factor, d=depth of tree.

A* Algorithm (Manhattan):

Heuristics (Manhattan):

```
for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if(mat[i][j]!=0){
                 for(int k=0;k<3;k++){
                for(int l=0;l<3;l++){

                    if (mat[i][j] == goal[k][l]){
                    int x=abs(i-k);
                    int y= abs(j-l);
                    dist_manhat=dist_manhat+(x+y);
                }
                }
            }
            }


        }
    }
    // for (int i = 0; i < 3; i++)
    // {
    //     for (int j = 0; j < 3; j++)
    //     {
    //         if (mat[i][j] != goal[i][j])
    //         {
    //             misplace++;
    //         }
    //     }
    // }
    // return dist_manhat+misplace;
    return dist_manhat;
}
```

Inversion Pair:

```
for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            for(int k=i;k<3;k++){
                for(int l=0;l<3;l++){
                    if(i==k&&l>j&&mat[k][l]!=0){
                        if(mat[i][j]>mat[k][l]){
                            // cout<<"inversion pair"<<mat[i][j]<<mat[k][l];
                            inversion++;
```

```
                        }
                    }
                    if(i!=k&&mat[k][l]!=0){
                        if(mat[i][j]>mat[k][l]){
                            // cout<<"inversion pair"<<mat[i][j]<<mat[k][l];
                            inversion++;
                        }
                    }
                }
            }
        }
    }
    if(inversion%2!=0){
        cout<<"Not Solvable";
    }
```

Code (Screenshot):

```
#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include<math.h>
using namespace std;

struct Node *CreateNewNode(int data[3][3]);
void addChild(Node *node, int data[3][3]);
struct Node
{
    int data[3][3];
    int fn;
    vector<Node *> children;
};
Node *root;
Node *temp = root;
vector<Node *> allNodes;
int flag1 = 0;
queue<Node *>Q;
int cnt=0;

// vector<Node *> childNodes;

map<int, Node *> openList;
vector<Node *> closedList;

int findFn(int mat[3][3]){
    int dist_manhat=0;
    int misplace=0;
    int goal[3][3] = {{1,2,3},
```

```
                    {4,5,6},
                    {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if(mat[i][j]!=0){
                 for(int k=0;k<3;k++){
                for(int l=0;l<3;l++){

                    if (mat[i][j] == goal[k][l]){
                    int x=abs(i-k);
                    int y= abs(j-l);
                    dist_manhat=dist_manhat+(x+y);
                }
                }
            }
            }


        }
    }
    // for (int i = 0; i < 3; i++)
    // {
    //     for (int j = 0; j < 3; j++)
    //     {
    //         if (mat[i][j] != goal[i][j])
    //         {
    //             misplace++;
    //         }
    //     }
    // }
    // return dist_manhat+misplace;
    return dist_manhat;
}
void get_Co_Zero(int mat[3][3], int *zerx, int *zery)
{
    int flag = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] == 0)
            {
                *zerx = i;
                *zery = j;
                flag = 1;
                break;
```

```
            }
        }
        if (flag == 1)
        {
            break;
        }
    }
}
bool isVisited(Node *node)
{
    int flagCheck = 0;
    for (int i = 0; i < allNodes.size(); i++)
    {
        // cout << "flagCheck===" << flagCheck;
        // cout << "\naniket\n";
        if (flagCheck == 0)
        {
            flagCheck = 1;
            // cout << "flagCheck1===" << flagCheck;
            for (int j = 0; j < 3; j++)
            {
                for (int k = 0; k < 3; k++)
                {
                    if (allNodes[i]->data[j][k] != node->data[j][k])
                    {
                        flagCheck = 0;
                        break;
                    }
                }
                if (flagCheck == 0)
                {
                    break;
                }
            }
        }
        else
        {
            break;
        }
    }
    if (flagCheck == 0)
    {
        allNodes.push_back(node);
        return false;
    }
    return true;
}
```

```
void up(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx - 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "up";
        addChild(temp, newm);
        Q.push(updatedmat);
        cnt++;
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));

    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
void down(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx + 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
```

```
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "down";
        addChild(temp, newm);
        Q.push(updatedmat);
        cnt++;
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));

    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
void right(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery + 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);

    if (!isVisited(updatedmat))
    {
        cout << "right";
        addChild(temp, newm);
        Q.push(updatedmat);
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));
        cnt++;
    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
```

```
void left(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery - 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "left";
        addChild(temp, newm);
        Q.push(updatedmat);
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));
        cnt++;
    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
void decide(Node *temp, int mat[3][3], int zerx, int zery)
{
    switch (zerx)
    {
    case 0:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);
```

```
            break;
        case 2:
        left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);



            break;
        default:
            break;
        }
        break;
    case 1:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:

            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);
            break;
        default:
            break;
        }
        break;

    case 2:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 1:
```

```
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        default:
            break;
        }
        break;

    default:
        break;
    }
}
void printData(Node *node)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << node->data[i][j];
        }
    }
    cout << "\n";
}
bool compareTwoArrays(Node *node)
{
    int goal[3][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            // cout << "\nnodeij=" << node->data[i][j];
            // cout << "\ngoalij=" << goal[i][j];
            if (node->data[i][j] != goal[i][j])
            {

                return false;
            }
        }
```

```
    }
    cout << "Reached"<<endl;
    cout<<"total state space="<<cnt;
    return true;
}

void checkAllPossibilties(Node *temp, int *zerx, int *zery)
{
    // while(!Q.empty())
    // {
        // cout << "comparing=" << !compareTwoArrays(temp->children[i]);
        while (!compareTwoArrays((*openList.begin()).second)&&flag1==0)
        {
            Node *minNode=(*openList.begin()).second;
            openList.clear();
            get_Co_Zero(minNode->data, zerx, zery);
            cout<<"\nminnode=";
            printData(minNode);
            cout<<"\n";
            cout<<"children\n";
            decide(minNode, minNode->data, *zerx, *zery);
            // sort(childNodes.begin(),childNodes.end());

            for (int j = 0; j < minNode->children.size(); j++)
            {
                printData(minNode->children[j]);
                cout<<"fn="<<findFn(minNode->children[j]->data)<<"\n";
                if (compareTwoArrays(minNode->children[j]))
                {
                    flag1 = 1;
                    break;
                }

            }
            cout<<"\n";
            if(flag1==1){
                    break;
                }
        }
        // Q.pop();
    // }


}

int main()
{
    int zerx, zery,inversion=0;
```

```
int mat[3][3] = {{1,2,3},
                 {0,4,6},
                 {7,5,8}};
for (int i = 0; i < 3; i++)
{
    for (int j = 0; j < 3; j++)
    {
        for(int k=i;k<3;k++){
            for(int l=0;l<3;l++){
                if(i==k&&l>j&&mat[k][l]!=0){
                    if(mat[i][j]>mat[k][l]){
                        // cout<<"inversion pair"<<mat[i][j]<<mat[k][l];
                        inversion++;
                    }
                }
                if(i!=k&&mat[k][l]!=0){
                    if(mat[i][j]>mat[k][l]){
                        // cout<<"inversion pair"<<mat[i][j]<<mat[k][l];
                        inversion++;
                    }
                }
            }
        }
    }
}
if(inversion%2!=0){
    cout<<"Not Solvable";
}
else{
    cout<<"inversion="<<inversion;
    root = CreateNewNode(mat);
temp = root;
allNodes.push_back(temp);
// checkAllPossibilties(temp, &zerx, &zery);

    cout << "root";
    for (int j = 0; j < 3; j++)
    {
        for (int k = 0; k < 3; k++)
        {

            cout << root->data[j][k];

        }
    }
    cout<<"\n";
get_Co_Zero(mat, &zerx, &zery);
```

```
    cout<<"\nchildren\n";
    decide(temp, mat, zerx, zery);

    for (int i = 0; i < root->children.size(); i++)
    {
        printData(root->children[i]);
        cout<<"fn="<<findFn(root->children[i]->data)<<"\n";
    }
    cout<<"\n";
    checkAllPossibilties(temp, &zerx, &zery);
    }


    return 0;
}
// function to create new node
Node *CreateNewNode(int data[3][3])
{
    Node *newNode = new Node();
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            newNode->data[i][j] = data[i][j];
        }
    }
    newNode->fn=findFn(data);
    return newNode;
}

// function to add a child to a specific node
void addChild(Node *node, int data[3][3])
{
    Node *newNode = CreateNewNode(data);
    node->children.push_back(newNode);
}
```

Output (Screenshot) 1:

```
inversion=2
root
123
046
758


children
right
up
down
123
406
758

fn=2
023
146
758

fn=4
123
746
058

fn=4


minnode=123
406
758


children
right
Already visited!!!
up
down
123
460
758
```

```
fn=3
103
426
758

fn=3
123
456
708

fn=1


minnode=123
456
708


children
right
left
Already visited!!!
123
456
780

fn=0
Reached
total state space=8
PS B:\Avinash Shelke\VIIT\SEM5\Artificial Intelligence\Assignment 1>
```

-> Total state space generated are **8.**

Output (Screenshot) 2:
For input:

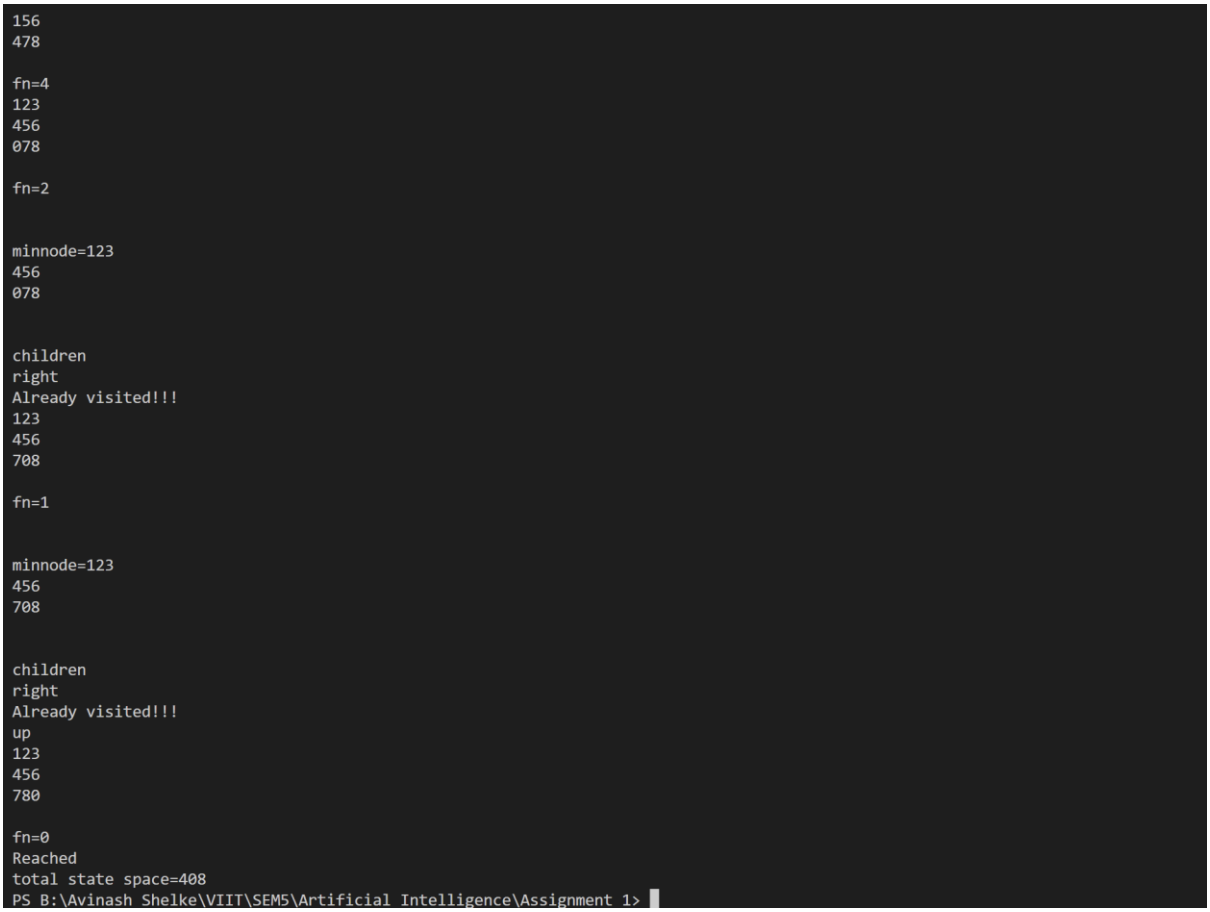
```
int zerx, zery;
int mat[3][3] = {{3,1,0},
                 {4,2,6},
                 {7,8,5}};
```

```
inversion=6
root
310
426
785


children
left
down
301
426
785

fn=7
316
420
785

fn=7


minnode=301
426
785


children
Already visited!!!
left
down
031
426
785
```

```
fn=6
321
406
785

fn=6


minnode=031
426
785


children
Already visited!!!
down
431
026
785

fn=7


minnode=431
026
785


children
right
Already visited!!!
down
431
206
785
```

```
fn=8
431
726
085

fn=8


minnode=431
206
785


children
right
Already visited!!!
up
down
431
260
785

fn=9
401
236
785

fn=9
431
286
705
```

……After a lot exploration……..



-> It stops finally, after exploring certain depth and **408** state spaces. Final state is **achieved**.

Time complexity: O(b^d)

b= branch factor, d=depth of tree.

Space Complexity: O(b^d)

b= branch factor, d=depth of tree.

A* Algorithm (Hamming):

Heuristics (Hamming):

```
int findFn(int mat[3][3]){
    int dist_manhat=0;
    int misplace=0;
    int goal[3][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if(mat[i][j]!=0){
                 for(int k=0;k<3;k++){
                for(int l=0;l<3;l++){

                    if (mat[i][j] == goal[k][l]){
                    int x=abs(i-k);
                    int y= abs(j-l);
                    dist_manhat=dist_manhat+(x+y);
                }
                }
            }
            }


        }
    }
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] != goal[i][j])
            {
                misplace++;
            }
        }
    }
    return dist_manhat+misplace;
}
```

Code (Screenshot):

```
#include <iostream>
#include <vector>
#include <queue>
```

```cpp
#include <map>
#include<math.h>
using namespace std;

struct Node *CreateNewNode(int data[3][3]);
void addChild(Node *node, int data[3][3]);
struct Node
{
    int data[3][3];
    int fn;
    vector<Node *> children;
};
Node *root;
Node *temp = root;
vector<Node *> allNodes;
int flag1 = 0;
queue<Node *>Q;
int cnt=0;

// vector<Node *> childNodes;

map<int, Node *> openList;
vector<Node *> closedList;

int findFn(int mat[3][3]){
    int dist_manhat=0;
    int misplace=0;
    int goal[3][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if(mat[i][j]!=0){
                 for(int k=0;k<3;k++){
                for(int l=0;l<3;l++){

                    if (mat[i][j] == goal[k][l]){
                    int x=abs(i-k);
                    int y= abs(j-l);
                    dist_manhat=dist_manhat+(x+y);
                }
                }
            }
            }
```

```
        }
    }
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] != goal[i][j])
            {
                misplace++;
            }
        }
    }
    return dist_manhat+misplace;
}
void get_Co_Zero(int mat[3][3], int *zerx, int *zery)
{
    int flag = 0;
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            if (mat[i][j] == 0)
            {
                *zerx = i;
                *zery = j;
                flag = 1;
                break;
            }
        }
        if (flag == 1)
        {
            break;
        }
    }
}
bool isVisited(Node *node)
{
    int flagCheck = 0;
    for (int i = 0; i < allNodes.size(); i++)
    {
        // cout << "flagCheck===" << flagCheck;
        // cout << "\naniket\n";
        if (flagCheck == 0)
        {
            flagCheck = 1;
            // cout << "flagCheck1===" << flagCheck;
            for (int j = 0; j < 3; j++)
            {
```

```
                for (int k = 0; k < 3; k++)
                {
                    if (allNodes[i]->data[j][k] != node->data[j][k])
                    {
                        flagCheck = 0;
                        break;
                    }
                }
                if (flagCheck == 0)
                {
                    break;
                }
            }
        }
        else
        {
            break;
        }
    }
    if (flagCheck == 0)
    {
        allNodes.push_back(node);
        return false;
    }
    return true;
}

void up(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx - 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
```

```
        cout << "up"<<endl;
        addChild(temp, newm);
        Q.push(updatedmat);
        cnt++;
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));

    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
void down(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zerx != 2)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zerx + 1;
        int temp = newm[newPos][zery];
        newm[newPos][zery] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
        cout << "down"<<endl;
        addChild(temp, newm);
        Q.push(updatedmat);
        cnt++;
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));

    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
void right(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 2)
    {
```

```
        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery + 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);

    if (!isVisited(updatedmat))
    {
        cout << "right"<<endl;
        addChild(temp, newm);
        Q.push(updatedmat);
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));
        cnt++;
    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
void left(Node *temp, int mat[3][3], int zerx, int zery)
{
    int newm[3][3];
    if (zery != 0)
    {

        for (int i = 0; i < 3; i++)
        {
            for (int j = 0; j < 3; j++)
            {
                newm[i][j] = mat[i][j];
            }
        }
        int newPos = zery - 1;
        int temp = newm[zerx][newPos];
        newm[zerx][newPos] = 0;
        newm[zerx][zery] = temp;
    }
    Node *updatedmat = CreateNewNode(newm);
    if (!isVisited(updatedmat))
    {
```

```
        cout << "left"<<endl;
        addChild(temp, newm);
        Q.push(updatedmat);
        openList.insert(pair<int,Node *>(findFn(newm),updatedmat));
        cnt++;
    }
    else{
        cout<<"Already visited!!!"<<endl;
    }
}
void decide(Node *temp, int mat[3][3], int zerx, int zery)
{
    switch (zerx)
    {
    case 0:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:
        left(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);



            break;
        default:
            break;
        }
        break;
    case 1:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
```

```
        case 1:
            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);

            break;
        case 2:

            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);
            down(temp, mat, zerx, zery);
            break;
        default:
            break;
        }
        break;

    case 2:
        switch (zery)
        {
        case 0:
            right(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 1:

            right(temp, mat, zerx, zery);
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        case 2:
            left(temp, mat, zerx, zery);
            up(temp, mat, zerx, zery);

            break;
        default:
            break;
        }
        break;

    default:
        break;
    }
}
void printData(Node *node)
```

```
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            cout << node->data[i][j];
        }
        cout<<endl;
    }
    cout << endl;
}
bool compareTwoArrays(Node *node)
{
    int goal[3][3] = {{1,2,3},
                      {4,5,6},
                      {7,8,0}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            // cout << "\nnodeij=" << node->data[i][j];
            // cout << "\ngoalij=" << goal[i][j];
            if (node->data[i][j] != goal[i][j])
            {

                return false;
            }
        }
    }
    cout << "Reached"<<endl;
    cout<<"total state space="<<cnt;
    return true;
}

void checkAllPossibilties(Node *temp, int *zerx, int *zery)
{
    // while(!Q.empty())
    // {
        // cout << "comparing=" << !compareTwoArrays(temp->children[i]);
        while (!compareTwoArrays((*openList.begin()).second)&&flag1==0)
        {
            Node *minNode=(*openList.begin()).second;
            openList.clear();
            get_Co_Zero(minNode->data, zerx, zery);
            cout<<"\nminnode=";
            printData(minNode);
            cout<<"\n";
            cout<<"children\n";
```

```
            decide(minNode, minNode->data, *zerx, *zery);
            // sort(childNodes.begin(),childNodes.end());

            for (int j = 0; j < minNode->children.size(); j++)
            {
                printData(minNode->children[j]);
                cout<<"fn="<<findFn(minNode->children[j]->data)<<"\n";
                if (compareTwoArrays(minNode->children[j]))
                {
                    flag1 = 1;
                    break;
                }

            }
            cout<<"\n";
            if(flag1==1){
                    break;
                }
        }
        // Q.pop();
    // }


}

int main()
{
    int zerx, zery,inversion=0;

    int mat[3][3] = {{1,2,3},
                     {0,4,6},
                     {7,5,8}};
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            for(int k=i;k<3;k++){
                for(int l=0;l<3;l++){
                    if(i==k&&l>j&&mat[k][l]!=0){
                        if(mat[i][j]>mat[k][l]){
                            // cout<<"inversion pair"<<mat[i][j]<<mat[k][l];
                            inversion++;
                        }
                    }
                    if(i!=k&&mat[k][l]!=0){
                        if(mat[i][j]>mat[k][l]){
                            // cout<<"inversion pair"<<mat[i][j]<<mat[k][l];
                            inversion++;
```

```
                        }
                    }
                }
            }
        }
    }
    if(inversion%2!=0){
        cout<<"Not Solvable";
    }
    else{
        cout<<"inversion="<<inversion <<endl;
        root = CreateNewNode(mat);
    temp = root;
    allNodes.push_back(temp);
    // checkAllPossibilties(temp, &zerx, &zery);

        cout << "root"<<endl;
        for (int j = 0; j < 3; j++)
        {
            for (int k = 0; k < 3; k++)
            {
                cout << root->data[j][k];
            }
            cout<<endl;
        }
        cout<<endl;
    get_Co_Zero(mat, &zerx, &zery);
    cout<<"\nchildren\n";
    decide(temp, mat, zerx, zery);

    for (int i = 0; i < root->children.size(); i++)
    {
        printData(root->children[i]);
        cout<<"fn="<<findFn(root->children[i]->data)<<"\n";
    }
    cout<<"\n";
    checkAllPossibilties(temp, &zerx, &zery);
    }


    return 0;
}
// function to create new node
Node *CreateNewNode(int data[3][3])
{
    Node *newNode = new Node();
    for (int i = 0; i < 3; i++)
    {
```

```
        for (int j = 0; j < 3; j++)
        {
            newNode->data[i][j] = data[i][j];
        }
    }
    newNode->fn=findFn(data);
    return newNode;
}

// function to add a child to a specific node
void addChild(Node *node, int data[3][3])
{
    Node *newNode = CreateNewNode(data);
    node->children.push_back(newNode);
}
```

Output (Screenshot) 1:

```
PS B:\Avinash Shelke\VIIT\SEM5\Artificial Intelligence\Assignment 1> cd "b:\Avinash Shelke\VIIT\SEM5\Artificia
ce.cpp -o astar_man_plus_misplace } ; if ($?) { .\astar_man_plus_misplace }
inversion=2
root
123
046
758


children
right
up
down
123
406
758

fn=5
023
146
758

fn=9
123
746
058

fn=9


minnode=123
406
758
```

```
children
right
Already visited!!!
up
down
123
460
758

fn=7
103
426
758

fn=7
123
456
708

fn=3


minnode=123
456
708


children
right
left
Already visited!!!
123
456
789
fn=0
Reached
total state space=8
PS B:\Avinash Shelke\VIII\SEM5\Artificial Intelligence\Assignment 1>
```

-> Total state space generated are **8.**

Output (Screenshot) 2:

For input:

```
int zerx, zery;
int mat[3][3] = {{3,1,0},
                 {4,2,6},
                 {7,8,5}};
```

```
PS D:\Avinash Sheike\VIIT\SEM5\Artificial Intelligence\Assignment 1> cd "D:\Avinash Sheike\VIIT\SEM5\Artificial Int
ce.cpp -o astar_man_plus_misplace } ; if ($?) { .\astar_man_plus_misplace }
inversion=6
root
310
426
785


children
left
down
301
426
785

fn=12
316
420
785

fn=13


minnode=301
426
785


children
Already visited!!!
left
down
```

```
031
426
785

fn=11
321
406
785

fn=10


minnode=321
406
785


children
right
left
Already visited!!!
down
321
460
785

fn=12
321
046
785

fn=12
321
486
fn=12


minnode=321
460
785


children
Already visited!!!
up
down
320
461
785

fn=13
321
465
780

fn=10


minnode=321
465
780


children
left
Already visited!!!
321
465
708
```

……After a lot exploration……..

```
minnode=120
453
786


children
Already visited!!!
down
123
450
786

fn=3


minnode=123
450
786


children
left
Already visited!!!
down
123
405
786

fn=5
123
456
780

fn=0
Reached
total state space=349
PS B:\Avinash Shelke\VIIT\SEM5\Artificial Intelligence\Assignment 1>
```

-> It stops finally, after exploring certain depth and **349** state spaces. Final state is **achieved**.

Output (Screenshot) 3:
For input:

```
int mat[3][3] = {{1,2,3},
                 {4,0,6},
                 {7,8,5}};
```

```
PS C:\Users\avina> cd "b:\Avinash Shelke\VIIT\SEM5\Artificial Intelligence\Assignment 1\" ; if ($?) { g++ astar_man_
astar_man_plus_misplace }
Not Solvable
PS B:\Avinash Shelke\VIIT\SEM5\Artificial Intelligence\Assignment 1>
```

-> It is not solvable type of puzzle (using **Inversion pair**).

Time complexity: O(b^d)

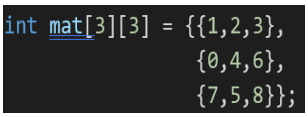
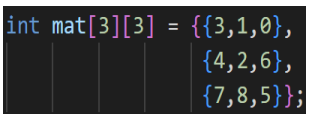
b= branch factor, d=depth of tree.

Space Complexity: O(b^d)

b= branch factor, d=depth of tree.

Conclusion:

Comparative complexity analysis of 8 puzzle w.r.t applied methodology i.e., DFS, BFS, A* (simple heuristic, Manhattan, Hamming (if applied))

	Easy Puzzle: 	**Hard Puzzle:** 
BFS	Puzzle solution explored at depth =3. Total state space generated are **14.**	The problem has solution at depth of 18. Therefore, BFS got stuck after exploring few depths.
DFS	Puzzle solution explored at depth =3. Total state space generated are **14.**	The problem has solution at depth of 18. Therefore, DFS got stuck after exploring some depths.
A*(misplaced tiles)	Puzzle solution explored at depth =3. Total state space generated are **8.**	It stops finally, after exploring certain depth. Final state **not achieved**.
A* (Manhattan)	Puzzle solution explored at depth =3. Total state space generated are **8.**	It stops finally, after exploring certain depth and **408** state spaces. Final state is **achieved**.
A* (Hamming)	Puzzle solution explored at depth =3. Total state space generated are **8.**	It stops finally, after exploring certain depth and **349** state spaces. Final state is **achieved**.