VIT Vidyalankar Institute of Technology Accredited A+ by NAAC	**DEPARTMENT OF COMPUTER ENGINEERING**

Experiment No. 4C

Semester	T.E. Semester VI
Subject	ARTIFICIAL INTELLIGENCE (CSL 604)
Subject Professor In-charge	Prof. Avinash Shrivas
Assisting Teachers	Prof. Avinash Shrivas

Student Name	Deep Salunkhe
Roll Number	21102A0014
Lab Number	310A

Title:

8 Puzzle Problem using Best First search

Theory:

The 8-puzzle problem is a classic problem in artificial intelligence and computer science, where the goal is to rearrange a scrambled 3x3 grid of tiles numbered from 1 to 8, with one blank space, into a particular goal state. It's a popular problem used for testing various search algorithms, including Best-First Search (BFS). Best-First Search is an informed search algorithm that uses heuristics to guide the search towards the most promising nodes. In the context of the 8-puzzle problem, BFS evaluates each state of the puzzle based on a heuristic function, which estimates the cost or distance from the current state to the goal state. The algorithm expands the nodes with the lowest heuristic values first, hence the name "Best-First" search. Here's how Best-First Search works for solving the 8-puzzle problem:

1. Initialization:
    - Start with the initial state of the puzzle and define a heuristic function to estimate the cost or distance to the goal state. Common heuristics for the 8-puzzle problem include the Manhattan distance or the number of misplaced tiles.
2. Priority Queue:
    - Maintain a priority queue (such as a min-heap) to store the states of the puzzle, prioritized based on their heuristic values. The state with the lowest heuristic value (i.e., the most promising state) is dequeued first for expansion.
3. Expansion:

- Expand the current state by generating all possible successor states (i.e., possible moves) from the current state. This step involves moving the blank tile (if possible) in all four directions: up, down, left, and right.

4. Evaluation:
   - For each successor state generated, calculate its heuristic value using the heuristic function. Enqueue the successor states into the priority queue based on their heuristic values.
5. Goal Test:
   - After expanding a state, check if it is the goal state. If the current state matches the goal state, terminate the search and return the solution path.
6. Repeat:
   - Repeat steps 3 to 5 until the goal state is reached or the priority queue becomes empty (indicating that no solution exists).
7. Solution Path:
   - If a solution is found, backtrack from the goal state to the initial state to reconstruct the solution path. The solution path represents the sequence of moves required to reach the goal state from the initial state.

Note: My solution does not contain implementation using heap as here which state to chose is decided by the user .

Program Code:

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


bool moveUp(vector<vector<int> >&IS,vector<vector<int> >&GS){

    int xb;
    int yb;

    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(IS[i][j]==0){
                xb=i;
                yb=j;
            }
        }
    }

    if(xb==0){
     return false;
    }
    else{
```

```
        swap(IS[xb][yb],IS[xb-1][yb]);
        return true;
    }

}

bool moveRight(vector<vector<int> >&IS,vector<vector<int> >&GS){

    int xb;
    int yb;

    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(IS[i][j]==0){
                xb=i;
                yb=j;
            }
        }
    }

    if(yb==2)
    return false;
    else{
        swap(IS[xb][yb],IS[xb][yb+1]);
        return true;
    }

}
bool moveDown(vector<vector<int> >&IS,vector<vector<int> >&GS){

    int xb;
    int yb;

    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(IS[i][j]==0){
                xb=i;
                yb=j;
            }
        }
    }

    if(xb==2)
    return false;
    else{
        swap(IS[xb][yb],IS[xb+1][yb]);
        return true;
```

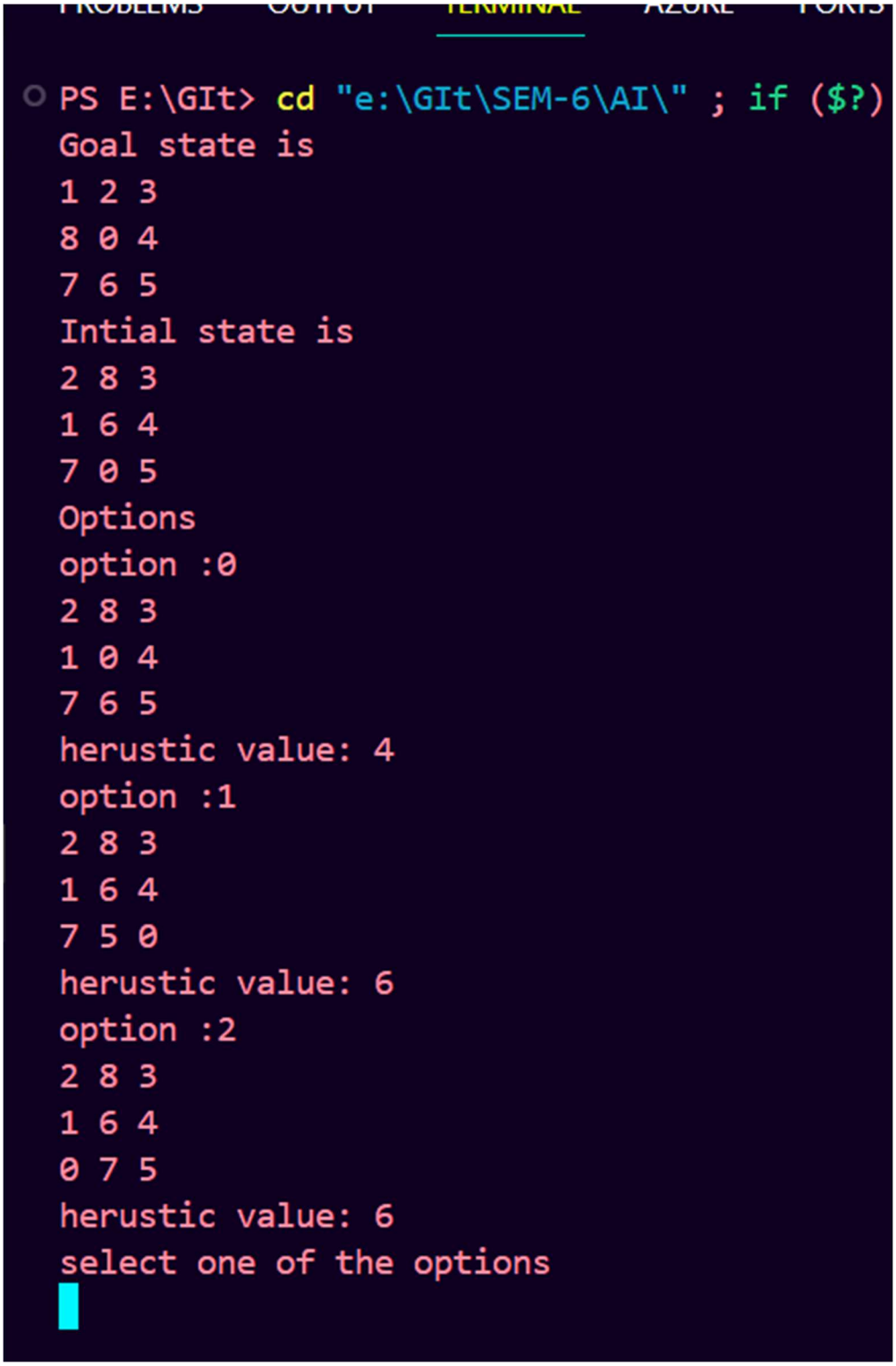
```
    }

}
bool moveLeft(vector<vector<int> >&IS,vector<vector<int> >&GS){

    int xb;
    int yb;

    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            if(IS[i][j]==0){
                xb=i;
                yb=j;
            }
        }
    }

    if(yb==0)
    return false;
    else{
        swap(IS[xb][yb],IS[xb][yb-1]);
        return true;
    }

}


void display2d(vector<vector<int> >v){
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            cout<<v[i][j]<<" ";
        }
        cout<<endl;
    }
}

bool goalReached(vector<vector<int> >&IS,vector<vector<int> >&GS){
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
          if(IS[i][j]!=GS[i][j])
          return false;
        }

    }

    return true;
}
```

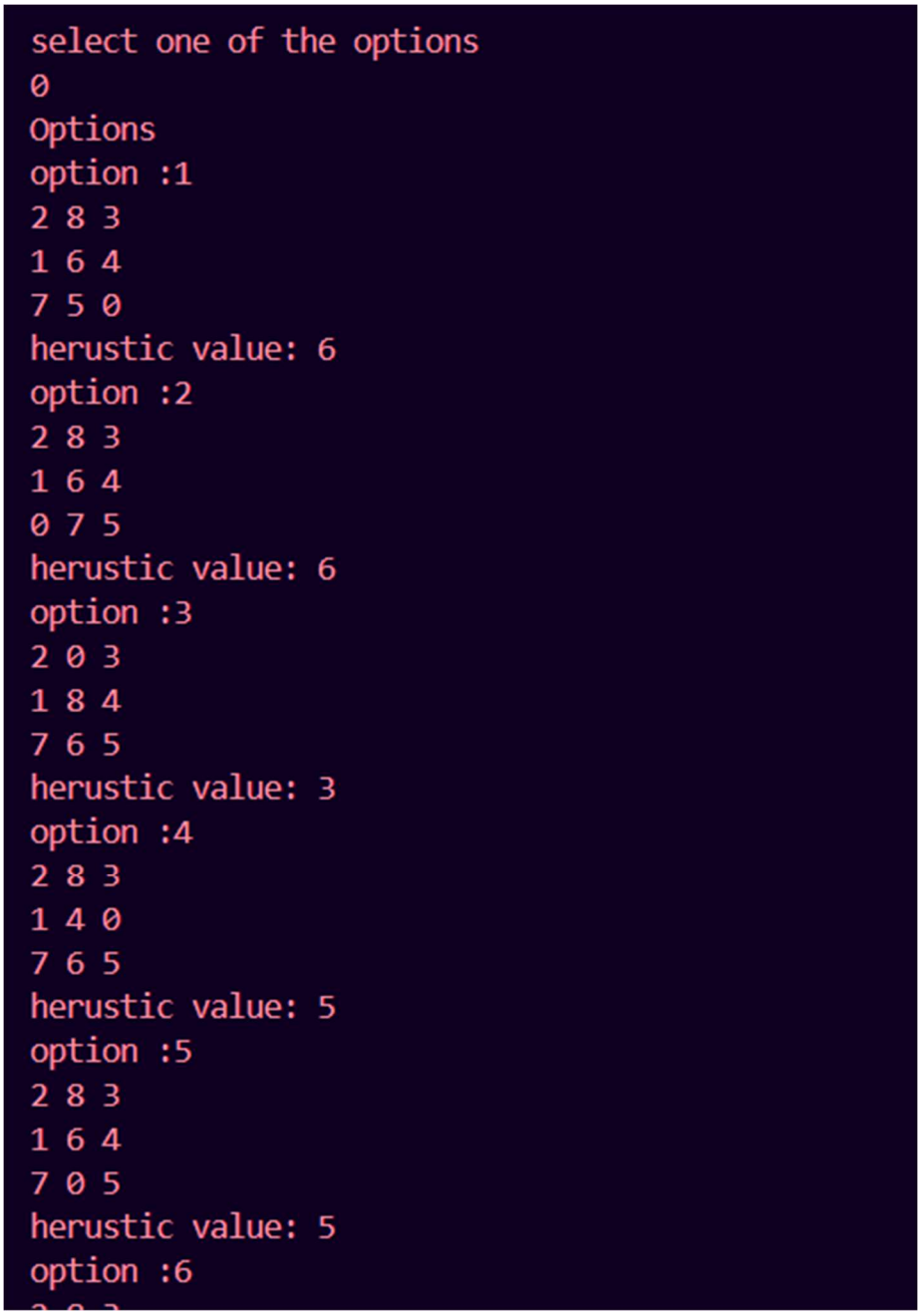
```
void generateOptions(vector<vector<int> >&IS,vector<vector<int> >&GS,   vec-
tor<vector<vector<int> > >&options,  map<vector<vector<int> >,bool>&visited){


        vector<vector<int> >currsu=IS;
        vector<vector<int> >currsr=IS;
        vector<vector<int> >currsd=IS;
        vector<vector<int> >currsl=IS;

        if(moveUp(currsu,GS)){

          if(visited.find(currsu)==visited.end())
          {
            options.push_back(currsu);

           visited[currsu]=false;
          }


        }

        if(moveRight(currsr,GS)){
            if(visited.find(currsr)==visited.end())
          {
          options.push_back(currsr);
          visited[currsr]=false;
    }

        }

        if(moveDown(currsd,GS)){
            if(visited.find(currsd)==visited.end())
          {
          options.push_back(currsd);
          visited[currsd]=false;
    }

        }

        if(moveLeft(currsl,GS)){
            if(visited.find(currsl)==visited.end())
          {
          options.push_back(currsl);
          visited[currsl]=false;
    }
```

```
        }




}

int heuristic(vector<vector<int> >&IS,vector<vector<int> >&GS){
    int value=0;
    for(int i=0;i<3;i++){
        for(int j=0;j<3;j++){
            int curr_ele=IS[i][j];
            int cex=i;
            int cey=j;
            int fx;
            int fy;
            if(curr_ele==0)
            continue;
            for(int x=0;x<3;x++){
                for(int y=0;y<3;y++){
                    if(GS[x][y]==curr_ele){
                        value=value+abs(cex-x)+abs(cey-y);
                    }
                }
            }
        }
    }

    return value;
}

void print3d(vector<vector<vector<int> > >options,map<vector<vector<int> >,bool>
&visited,  vector<vector<int> >GS){
    int n=options.size();
    for(int i=0;i<n;i++){
      if(visited[options[i]]==false){
        cout<<"option :"<<i<<endl;
         display2d(options[i]);
         int h=heuristic(options[i],GS);
         cout<<"herustic value: "<<h<<endl;   }

    }
}


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

```
    vector<vector<int> >GS={{1,2,3},{8,0,4},{7,6,5}};
    vector<vector<vector<int> > >options;
    map<vector<vector<int> >,bool>visited;
    cout<<"Goal state is"<<endl;
    display2d(GS);
    cout<<"Intial state is"<<endl;
    display2d(IS);

    while(true){

        generateOptions(IS,GS,options,visited);
        cout<<"Options"<<endl;
        print3d(options,visited,GS);
        cout<<"select one of the options"<<endl;
        int ind;
        cin>>ind;
        IS=options[ind];
        visited[IS]=true;

        if(goalReached(IS,GS)){
            cout<<"Congrats the goal is reached"<<endl;
            break;
        }
    }


}
```

Output:


PROBLEMS OUTPUT TERMINAL AZURE PORTS
PS E:\GIt> cd "e:\GIt\SEM-6\AI\" ; if ($?)
Goal state is
1 2 3
8 0 4
7 6 5
Intial state is
2 8 3
1 6 4
7 0 5
Options
option :0
2 8 3
1 0 4
7 6 5
herustic value: 4
option :1
2 8 3
1 6 4
7 5 0
herustic value: 6
option :2
2 8 3
1 6 4
0 7 5
herustic value: 6
select one of the options



Conclusion:

Best-First Search is effective for solving the 8-puzzle problem because it prioritizes the exploration of states that are more likely to lead to the goal state, based on the heuristic information. However, the choice of heuristic function greatly influences the efficiency and optimality of the search algorithm.