```cpp
#include <bits/stdc++.h>
using namespace std;
//Verison Tries to detect and elliminate duplicate nodes to optimize serach memorywise.
void checkValid(int temp[]);
void print(int temp[3][3]);
void solve(int initial[3][3], int fin[3][3], int x, int y,int choice);
int computeMisplaced(int initial[3][3], int fin[3][3]);
int computeManhattan(int initial[3][3], int fin[3][3]);
long long int computeuni(int man, int dis, int initial[3][3]);
long long int numberOfNodes=0;
vector < long long int > a;
int islegal(int x, int y)
{   // To check if the move is legal or not. (otherwise we may get segmentation fault.)
    return ((x >= 0 && x < 3) && (y >= 0 && y < 3));
}
int isNonDuplicate(int x, int y, int initial[3][3])
{   // To check if the move has been seen already. (otherwise we may run outta mem.)
    long long int temp = computeuni(x,y,initial);
    if(std::find(a.begin(), a.end(), temp) != a.end())
      return 0;  //elem exists in the vector
    a.push_back(temp);
    return 1;
}

struct Node
{
    //state at the node
    int state[3][3];
    //the number of misplaced tiles Heuristic
    int misplaced;
    //store manhattan Heuristic
    int manhattan;
    //cost to reach assumed to be 1 for ever parent child edge.
    int uniformCost;
    // position of blank in state
    int x, y;
    int Astar;
};
struct comparator{
    bool operator()(const Node* lhs, const Node* rhs) const{
        return (lhs->Astar) > (rhs->Astar);
    }
};

Node* expand(int state[3][3], int x, int y, int newX,
              int newY, int Cost){
    Node* node = new Node;
    numberOfNodes++;
    // copy data from parent node to current node
    memcpy(node->state, state, sizeof node->state);
    // move tile by 1 postion
    swap(node->state[x][y], node->state[newX][newY]);
    // set no. misplaced tiles
    node->misplaced = INT_MAX;
    // set no. manhattan Heuristic
    node->manhattan = INT_MAX;
    // set cost to reach
    node->uniformCost = Cost;
    node->Astar = INT_MAX;
    // update new blank tile cordinates
    node->x = newX;
    node->y = newY;
    return node;
}

int main()
{

    int initial[3][3] = //if default  chosen this will  be initial state
```

```
70        {   {8, 6, 7},
71            {2, 5, 4},
72            {3, 0, 1}
73        };
74        int Choice;
75        // Input for matrix
76        cout<<"Welcome to the 8-puzzle solver."<<endl<<
77          "Type 1 to use a default puzzle, or 2 to enter your own puzzle"
78          <<endl;
79        cin>>Choice;
80        if(Choice == 2){
81          cout<<"Please enter the numbers for each row and press enter"<<endl;
82          cout<<"Note use 0 for blank."<<endl;
83          for(int i = 0; i < 3 ; i++){
84            cout<<"Enter Space seprated elements of the row "<<i+1<<endl;
85            for(int j = 0; j < 3 ; j++){
86              cin>>initial[i][j];
87            }
88          }
89        }
90        cout<<"The Entered Matrix is..."<<endl;
91        print(initial);
92        cout<<endl;
93        checkValid((int *)initial); //To check if input matrix is valid or not
94        if((Choice != 1) && (Choice != 2)){
95          cout<<"Wrong Choice... Exiting"; //incase of erroneous choice entry
96          exit(0);
97        }
98
99        // Input for Algorithm
100       cout<<"choose the algorithm"<<endl;
101       cout<<"1. Uniform Cost Search"<<endl;
102       cout<<"2. A* with misplaced tile"<<endl;
103       cout<<"3. A* with Manhattan distance"<<endl;
104       cin>>Choice;
105       cout<<endl;
106       Choice = Choice-1;
107
108       // Defining goal state
109       int fin[3][3] =
110       {   {1, 2, 3},
111           {4, 5, 6},
112           {7, 8, 0}
113       };
114
115       //find the postion of blank in initial
116       int x,y;
117       for (int i = 0; i < 3; i++) {
118         for (int j = 0; j < 3; j++) {
119           if(initial[i][j]==0){
120             x = i; y = j;
121           }
122         }
123       }
124       //Calculating the time
125       clock_t start = clock();
126       solve(initial,fin,x,y,Choice);
127       clock_t end = clock();
128       if((double(end - start) / CLOCKS_PER_SEC)==0)
129         cout<<"total clocks taken by solve function: "<< (end - start) ;
130       else cout<<"total time taken by solve function: "<< double(end - start) /
          CLOCKS_PER_SEC;
131
132       return 0;
133   }
134   void checkValid(int temp[]){
135     //check if the given matrix has a solution before building the tree.
136     // This is done by counting number of inversions in the matrix if even solvable
137     // if odd it's impossible.
```

```
    //inspired by
    https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html

    int inverted = 0;
    for (int i = 0; i < 8; i++)
        for (int j = i+1; j < 9; j++)
             if (temp[j] && temp[i] &&  temp[i] > temp[j])
                  inverted++;
    /*Commented out to check the robustness of solve function
    if(inv % 2 != 0){
      cout<<"The Matrix entered is impossible to solve";
      exit(0);
    }*/
    // check if entries are among 0-8
    for (int i = 0; i < 9 ; i++) {
      if(temp[i] < 0 && temp[i] > 8 ){
        cout<<"Invalid entry"<< temp[i]<<" in the matrix";
        exit(0);
      }
    }
}

void print(int temp[3][3])
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
            cout<< temp[i][j]<<" ";
        cout<<endl;
    }
}

// Main solving subroutine
void solve(int initial[3][3], int fin[3][3], int x, int y, int Choice){
  int k1,k2,k3,numberOfNodesExpanded;
  numberOfNodesExpanded = 0;
  if(Choice==0){
    k1 = 1; k2 = 0; k3 = 0;
  }
  else if(Choice==1){
    k1 = 1; k2 = 1; k3 = 0;
  }
  else{
      k1 = 1; k2 = 0; k3 = 1;
  }
  //1.make priority queue
  priority_queue <Node*, vector<Node*>, comparator> frontier;
  //2.start building tree with initial state
  Node* initnode = expand(initial,x,y,x,y,0);
  //enter heuristic numbers
  initnode->uniformCost = 0;
  initnode->manhattan = computeManhattan(initial,fin);
  initnode->misplaced = computeMisplaced(initial,fin);
  initnode->Astar = (k1*(initnode->uniformCost)) + (k2*(initnode->misplaced)) +
  (k3*(initnode->manhattan));
  //3.put the only node in pri queue
  frontier.push(initnode);
  //4. take it out from pri queue
  //5. check if this is goal state
  //6. expand the node and put them in pq
  //repeat.3-6 on algorithm given.
  int rowop[] = {-1, 0, 1, 0 };
  int colop[] = { 0, 1, 0, -1 };
  int maxQueue = 0;
  // above are possible operations
  while (!frontier.empty())
  {
      // Find least estimated cost node
       Node* min = frontier.top();
```

```
205          if(frontier.size() > maxQueue) maxQueue = frontier.size();
206          cout<<"About to expand the following state"<<endl;
207          cout<<endl;
208          print(min->state);
209          cout<<endl;
210          cout<<"Cost associated with this state is: "<<min->Astar<<endl<<endl;
211        // delete the node from queue
212        frontier.pop();
213
214        // check if popped element is the goal state
215        if (min->misplaced == 0)
216        {
217            cout<<"The goal state reached"<<endl;
218            print(min->state);
219            cout<<endl;
220            cout<<"Number of states created: "<<numberOfNodes<<endl;
221            cout<<"Number of states expanded: "<<numberOfNodesExpanded<<endl;
222            cout<<"Max entries in the queue: "<<maxQueue<<endl;
223            cout<<"The solution was at level: "<<min->uniformCost<<endl;
224            return;
225        }
226        numberOfNodesExpanded++;
227        // do for each child of min
228        // max 4 children for a node
229        for (int i = 0; i < 4; i++)
230        {
231            if (islegal(min->x + rowop[i], min->y + colop[i]))
232            {
233                // create a child node and calculate costs
234                Node* child = expand(min->state, min->x,
235                             min->y, min->x + rowop[i],
236                             min->y + colop[i],
237                             ((min->uniformCost) + 1));
238                child->misplaced = computeMisplaced(child->state, fin);
239                child->manhattan = computeManhattan(child->state, fin);
240                child->Astar = k1*child->uniformCost + k2*child->misplaced +
                 k3*child->manhattan;
241                // Add child to priority_queue
242                if(isNonDuplicate(child->misplaced,child->manhattan,child->state))
243                  frontier.push(child);
244            }
245        }
246    }
247    cout<<"No Solution "<<endl;
248    cout<<"Number of states created: "<<numberOfNodes<<endl;
249    cout<<"Number of states expanded: "<<numberOfNodesExpanded<<endl;
250    cout<<"Max entries in the queue: "<<maxQueue<<endl;
251  }
252  int computeMisplaced(int initial[3][3], int fin[3][3])
253  {
254      //compute how many tiles are in wrong places
255      int count = 0;
256      for (int i = 0; i < 3; i++)
257        for (int j = 0; j < 3; j++)
258          if (initial[i][j] && initial[i][j] != fin[i][j])// if initial state not equal
          to final and not equal to 0
259              count++;
260      return count;
261  }
262  int computeManhattan(int initial[3][3], int fin[3][3])
263  {
264      //compute how far is each tile from it's destination
265      int count = 0;
266      int tileVal=0, finx=0, finy=0;
267      for (int i = 0; i < 3; i++)
268        for (int j = 0; j < 3; j++){
269            tileVal = initial[i][j];
270            if(tileVal == 0) continue;
271            finx = (tileVal-1) / 3;
```

```
            finy = (tileVal-1) % 3;
            count = count + abs(finx - i) + abs(finy - j); //if tile at final position,
            // count=count
        }
      return count;
}
long long int computeuni(int man, int dis, int initial[3][3]){ //Modified from stack
exchange
  int sum=0;
  for (int i = 0; i < 3; i++) {
    for(int j=0;j<3;j++)
    sum = sum + 300*pow(i+1,3)+10*pow(j+1,3)*(initial[i][j]) ;
  }
  return sum*man*dis;
}

```