**Programming Assignment 1: Implementing 8-puzzle problem using A\* algorithm**

**Name**: Krishna Sreenivas

**Student** **ID**: 800984436

**Problem description:**

The puzzle has 9 tiles, each tile has a number assigned to it, starting from 1-8. However, the 9th tile has 0 assigned to it indicating a blank tile. Given the start state and the end state the blank tile can be moved in 4 directions (up, down, left, right) one tile at a time and can be used to swap with another tile. This project uses A\* algorithm which is a path finding algorithm, used to find a path with least cost to the end state.

In this project, I have used Manhattan distance to be the heuristic function. Based on the f(n) value, the algorithm finds the best possible path to the end state. The advantage of using A\* is that, it has the capability to backtrack to the parent of the generated node. While, maintaining the cost of the path as well.

**Program description:**

Global variables:

I have used a global variable *node\_count* to keep track of the number of nodes generated.

Functions:

Below is the list of functions along with a brief description.

1. Tree(): Constructor to initialize the root node to 0.
2. void input(): Takes the input from user for start and end states.
3. bool if\_empty(): Used to check whether the root node is 0 or not.
4. void construct\_root(vector<int> start\_state): Creates an root node for every state.
5. int heuristic\_dist(node\* temp): Used to obtain an estimate cost from current to end state.
6. void node\_sort(): Custom sorting function.
7. void gen\_nodes(): Creates a tree of nodes. So that A\* can back track.
8. void insert\_node(node\* current): Expands existing nodes.
9. void success\_check(node\* current): Checks if the end state could be reachable from current state.
10. void state\_change(node\* current, node\* temp, int dir): changing states by moving blank tile on basis of Manhattan distance.
11. void print(node\* root): Prints individual nodes(i.e. states) by backtracking.

**Code:**

#include <iostream>

#include <stdlib.h>

#include <vector>

#include <algorithm>

using namespace std;

int node\_count=1; //count of nodes generated

class Tree

{

public:

struct node //structure holding node details such as node movements, state, location of blank in puzzle, branch cost and final cost

{

int branch\_cost;//branch cost i.e. g(n)

int final\_cost;//final cost f(n)

vector<int> state;//vector to store digits of the puzzle

int index;//used to know the location of blank tile

node\* parent;

node\* up;//4 movements of the blank tile

node\* down;

node\* left;

node\* right;

};

vector<int> start\_state, end\_state; //start and end state vectors

int heuristic; //holding the h(n) value

bool check\_path; //to check if a path is valid

node\* root; //to hold root of a state

vector<node\*> node\_list; //vector to hold nodes generated

Tree()

{

root=0; //setting root to 0

}

void input() //taking inputs from the user for start and end state.

{

int input;

cout<<"Enter digits from 1 - 8 for start state. Enter 0 for blank tile\n";

for(int i=0;i<9;i++)

{

cin>>input;

start\_state.push\_back(input);

}

cout<<"Enter digits from 1 - 8 for the end state. Enter 0 for blank tile\n";

for(int i=0;i<9;i++)

{

cin>>input;

end\_state.push\_back(input);

}

}

bool if\_empty()//checking if root is empty or not.

{

return root==0;

}

void construct\_root(vector<int> start\_state)

{

node\* new\_node=new node; //creating a empty root node

for(int i=0;i<9;i++)

new\_node->state.push\_back(start\_state[i]);

new\_node->final\_cost=heuristic\_dist(new\_node);

new\_node->branch\_cost=1; //branch cost i.e. depth

new\_node->parent=0;//parent of the new node set to 0

new\_node->up=0;//setting up the movements to 0 initially

new\_node->down=0;

new\_node->left=0;

new\_node->right=0;

for(int i=0;i<9;i++) //checking for the index of blank tile

{

if (start\_state[i]==0)

new\_node->index=i;

}

if (if\_empty())//need to check if the root of the node doesnt have parent

{

root=new\_node; //set new\_node\_1 to root.

node\_list.push\_back(root);

}

}

int heuristic\_dist(node\* temp) //estimating cost to end state from current state h(n)

{

int heuristic\_cost=0;

for(int i=0;i<temp->state.size();i++)

{

if (end\_state[i]!=temp->state[i])

{

heuristic\_cost++;

}

}

return heuristic\_cost;

}

void gen\_nodes()//used to generate tree of nodes

{

construct\_root(start\_state); //constructing root node

node\* current;

check\_path=false;

int count=0;

heuristic=1;

while(node\_list.size()>0) //comparing current node with all nodes present in the node list

{

current=node\_list.front(); //choosing the minimum node

node\_list.erase(node\_list.begin());

insert\_node(current);

if(current->branch\_cost>count)//choosing the node which has least g(n) value

{

count=current->branch\_cost;

cout<<"Number of nodes in path= "<<count<<”\n”;

}

}

}

void node\_sort()//sorting nodes in descending order.

{

node\* temp;

for (int i=0;i<node\_list.size()-1;i++)

{

for (int j=0;j<node\_list.size()-1;j++)

{

if (node\_list[j]->final\_cost>node\_list[j+1]->final\_cost)

{

temp=node\_list[j];

node\_list[j]=node\_list[j+1];

node\_list[j+1]=temp;

}

}

}

}

void insert\_node(node\* current)//expanding new nodes of an already existing node

{

node\* new\_node\_1=new node;

state\_change(current,new\_node\_1,1); //swapping the node upwards with blank tile

new\_node\_1->branch\_cost=current->branch\_cost+1; //increasing branch cost by 1

new\_node\_1->final\_cost=heuristic\_dist(new\_node\_1)+new\_node\_1->branch\_cost;//A\* algorithm formula

new\_node\_1->parent=current; //make current node as parent node

//setting all possible movements to zero indicating new node is a leaf node

new\_node\_1->up=0;

new\_node\_1->down=0;

new\_node\_1->left=0;

new\_node\_1->right=0;

node\* new\_node\_2=new node;

state\_change(current,new\_node\_2,4);//swapping the node right with blank tile

new\_node\_2->branch\_cost=current->branch\_cost+1;//increasing branch cost by 1

new\_node\_2->final\_cost=heuristic\_dist(new\_node\_2)+new\_node\_2->branch\_cost;//A\* algorithm formula

new\_node\_2->up=0;//setting all possible movements to zero indicating new node is a leaf node

new\_node\_2->down=0;

new\_node\_2->left=0;

new\_node\_2->right=0;

new\_node\_2->parent=current;//make current node as parent node

node\* new\_node\_3=new node;

state\_change(current,new\_node\_3,2);//swapping the node down with blank tile

new\_node\_3->branch\_cost=current->branch\_cost+1;//increasing branch cost by 1

new\_node\_3->final\_cost=heuristic\_dist(new\_node\_3)+new\_node\_3->branch\_cost;//A\* algorithm formula

new\_node\_3->up=0;//setting all possible movements to zero indicating new node is a leaf node

new\_node\_3->down=0;

new\_node\_3->left=0;

new\_node\_3->right=0;

new\_node\_3->parent=current;//make current node as parent node

node\* new\_node\_4=new node;

state\_change(current,new\_node\_4,3);//swapping the node left with blank tile

new\_node\_4->branch\_cost=current->branch\_cost+1;//increasing branch cost by 1

new\_node\_4->final\_cost=heuristic\_dist(new\_node\_4)+new\_node\_4->branch\_cost;//A\* algorithm formula

new\_node\_4->up=0;//setting all possible movements to zero indicating new node is a leaf node

new\_node\_4->down=0;

new\_node\_4->left=0;

new\_node\_4->right=0;

new\_node\_4->parent=current;//make current node as parent node

if ((current->up==0)&&(new\_node\_1->state.size()>0))

//if up node is not linked to current and has non-zero size,

//then link new\_node\_1 and current by up.

{

node\_count++;

current->up=new\_node\_1;

if (!check\_path)

node\_list.push\_back(current->up);

}

else

delete new\_node\_1;

if ((current->down==0)&&(new\_node\_2->state.size()>0))

{

node\_count++;

current->down=new\_node\_2;

if (!check\_path)

node\_list.push\_back(current->down);

}

else

delete new\_node\_2;

if ((current->left==0)&&(new\_node\_3->state.size()>0))

{

node\_count++;

current->left=new\_node\_3;

if (!check\_path)

node\_list.push\_back(current->left);

}

else

delete new\_node\_3;

if ((current->right==0)&&(new\_node\_4->state.size()>0))

{

node\_count++;

current->right=new\_node\_4;

if (!check\_path)

node\_list.push\_back(current->right);

}

else

delete new\_node\_4;

node\_sort(); //node\_sort nodes with minimum pathcost

success\_check(current); //check if end\_state state is reached or not.

}

void success\_check(node\* current)//to check if end state is accessible from current state or not

{

bool check1,check2,check3,check4;

check1=check2=check3=check4=true;

for (int i=0;i<9;i++)

{

if (current->up!=0)

{

if (current->up->state[i]!=end\_state[i])

check1=false;

}

else

check1=false;

if (current->down!=0)

{

if (current->down->state[i]!=end\_state[i])

check2=false;

}

else

check2=false;

if (current->left!=0)

{

if (current->left->state[i]!=end\_state[i])

check3=false;

}

else

check3=false;

if (current->right!=0)

{

if (current->right->state[i]!=end\_state[i])

check4=false;

}

else

check4=false;

}

if(check1)

{

check\_path=true;

cout<<"Done!\n";

print(current->up);

}

else if(check2)

{

check\_path=true;

cout<<"Done!\n";

print(current->down);

}

else if(check3)

{

check\_path = true;

cout<<"Done!\n";

print(current->left);

}

else if(check4)

{

check\_path = true;

cout<<"Done!\n";

print(current->right);

}

if (check\_path)

{

while(!node\_list.empty())

node\_list.erase(node\_list.begin());

}

}

void state\_change(node\* current, node\* temp, int dir)//changing the state on basis of manhattan distance

{

int index;

bool wrong\_move=false;

for (int i=0;i<current->state.size();i++)

{

if (current->state[i]==0)

{

index=i;

}

temp->state.push\_back(current->state[i]);

}

temp->index=index;

if (current->branch\_cost>2)

{

if (index==current->parent->index)

{

wrong\_move=true;

}

}

int temp1;

switch (dir) { //to move tiles in up,down,left,right based on manhattan distance

case 1: // to move blank space in upward dir

if ((index > 2)&&(!wrong\_move)) // to move upwards only if the blank is in 2 and 3 row

{

temp1=temp->state[index];

temp->state[index]=temp->state[index-3];

temp->state[index-3]=temp1;

}

else //delete the node if upward movement is not possible.

temp->state.erase(temp->state.begin(),temp->state.end());

break;

case 4:

if ((index%3!=2)&&(!wrong\_move))

{

temp1=temp->state[index];

temp->state[index]=temp->state[index+1];

temp->state[index+1]=temp1;

}

else

temp->state.erase(temp->state.begin(),temp->state.end());

break;

case 2:

if ((index<6)&&(!wrong\_move))

{

temp1=temp->state[index];

temp->state[index]=temp->state[index+3];

temp->state[index+3]=temp1;

}

else

temp->state.erase(temp->state.begin(),temp->state.end());

break;

case 3:

if ((index%3!=0)&&(!wrong\_move))

{

temp1=temp->state[index];

temp->state[index]=temp->state[index-1];

temp->state[index-1]=temp1;

}

else

temp->state.erase(temp->state.begin(),temp->state.end());

break;

}

}

void print(node\* root) { //prints states from start to end states.

vector<node\*> pnt;

while(root!=0)//backtracking to root node from current node

{

pnt.push\_back(root);

root=root->parent;

}

cout<<"Starting State\n";

for (int j=pnt.size()-1;j>=0;j--)

{

cout<<"Move number "<<pnt.size()-j-1<<":\n";

for (int i=0;i<pnt[j]->state.size();i++)

{

if (pnt[j]->state[i]==0)

cout<<"0 ";

else

cout<<pnt[j]->state[i]<< " ";

if (i % 3 == 2)

cout<<"\n"; //indicates end of row.

}

cout<<"\n";

}

cout<<"End State\n";

cout<<"Total number of nodes generated=" <<node\_count<<"\n";

}

};

int main()

{

Tree b;//tree object

b.input();//taking inputs

b.gen\_nodes();//generating tree

return 0;

}

**Output screenshots:**

