REPORT

PROJECT 1
A* ALGORITHM IMPLEMENTATION
FOR
8 PUZZLE PROBLEM

SUBMITTED BY:
SAKSHI SHRIVASTAVA
(800987926)

1) **The 8-puzzle Formulation:** The problem formulation for the 8-puzzle problem is as follows:
   States: The different locations of the tiles represents the states of the problem
   Operations: The tiles can be moved up, down, left and right.
   Initial State: Initial state can be any arrangement of the tile. They may or may not be ordered.
   Goal State: Goal state can again be any arrangement of the tile. They may or may not be ordered.
   Path Cost: For A* algorithm the path cost f(n) = g(n) + h(n). Where h(n) is the heuristic function.

2) **The program structure and program flow:**
   The program is structured in a way that it takes initial and goal states from the user as input and stores them appropriately for further computation. C++ Structure is used to define a node. It include all its characteristics including the state of the node, its cost function, its depth and its parent and child nodes. Different functions/ methods namely: calculateHeuristic, generateNode, generateSuccessor, addToQueue, isRepeated, printStates, goalTest and AStar.

   AStar method includes a implementation of the A* algorithm and calls all other methods for various calculations and manipulations. The main method takes the input from the user, creates the initial and goal nodes and calls the AStar method for further execution.

3) **Global variables and functions:**
   Global variables:

```
1) struct Node{
   int fn;
   int pathcost;
   int state[3][3];
   Node* next, *parent;
   Node* child[4];
  }*front,*visited;
```

The structure define all the properties of a node. Front points to the start of the priority queue and Visited points to the visited queue.

2) goal- Stores the goal state.
3) nodesGenerated - Stores the count of the generated nodes . It acts as a counter that increases as a new node is generated.
4) nodesExpanded - Stores the count of the expanded nodes. It acts as a counter that increases as a new node is generated.

Functions:

1) calculateHeuristic: The function is used to calculate the heuristic value of a node. It takes as input the current node and returns its heuristic value. We are using Manhattan distance as the heuristic parameter.

2) generateNode: The function is used to create a new node. The function takes as input the current state, its path cost and its parents and returns a node i.e node structure.
3) generateSuccessor: The function is used to generate the successor for a node using the up, down, left and right operations. The function takes as input the current state.
4) addToQueue: The function is used to add a node to the queue based on its path cost. The function takes as input a node.
5) isRepeated: The function is used to check whether a state is repeated or not. The function takes as input a node and returns 1 if the stated is repeated else it return 0.
6) printStates: The function is used to print all the possible states for a node. The function takes as input a node.
7) goalTest: The function is used to check whether the current state is a goal state or not. The function takes as input a node.
8) AStar: The function implements the A* algorithm and calls all other methods for various calculations and manipulations

4) Program source codes:

```
#include <iostream>
#include <stdlib.h>
#include<stdlib.h>


#define UP 0
#define DOWN 1
#define RIGHT 2
#define LEFT 3


//Defining characterstics of a node for 8 Puzzle problem.
struct Node{
        int fn;                                         // the f(n) value.
        int pathcost;                           // level or depth of the node.
        int state[3][3];                        // state of the 8 puzzle at a particular time.
        Node* next, *parent;            // parent of current node.
        Node* child[4];                         // Any node can have at maximum 4 children.

}*front,*visited;

//front: start of the priority queue.
// visited: start of visited nodes.

int goal[3][3];                                 // The finial state entered by the user.
int nodesGenerated = 0;                 // Stores the number of nodes generated.
```

```
int nodesExpanded = 0;                    // Stores the number of nodes expanded.


// Method to calculate the heuristic function h(n). The method calculates the manhattan distance which we take as the heuristic value.
int calculateHeuristic(int s[3][3])
{
   int hn = 0,i,j,k,l;

   for(i=0;i<3;i++)
   {
      for(j=0;j<3;j++)
      {
         if(s[i][j]!= goal[i][j] && s[i][j]!= 0)
         {
            for(k=0;k<3;k++)
            {
               for(l=0;l<3;l++)
               {
                  if(s[i][j] == goal[k][l])
                     hn += (abs(k-i) + abs(l-j));
               }
            }

         }
      }
   }

   return hn;
}

// Methode to generate or create a node.
struct Node* generateNode(int n[3][3],int gn, Node* p)
{
            nodesGenerated++;
            Node *temp = new Node;
            temp->pathcost = gn;
            temp->fn = calculateHeuristic(n)+gn;                    //Since for A* f(n) = g(n) + h(n).
            temp->next = NULL;
            temp->parent = p;
            for(int i=0;i<4;i++)
                        temp->child[i]=NULL;
            for(int i=0;i<3;i++)
```

```
	{
		for(int j=0;j<3;j++)
		{
			temp->state[i][j]= n[i][j];
		}
	}

	return temp;
}



// Methode to generate all possible states for a node using operator: UP, DOWN, LEFT & RIGHT.
void generateSuccessor(Node* n)
{
	int temp[3][3];
	int x,y;				// x is co-ordinate of blank tile , y is the y co-ordinate of
blank tile
	for(int i=0;i<3;i++)
	{
		for(int j=0;j<3;j++)
		{
			if(n->state[i][j] == 0)
			{
				x = i;
				y = j;
				break;
			}
		}
	}

	for(int k=0;k<4;k++)
	{
		for(int i=0;i<3;i++)
		{
			for(int j=0;j<3;j++)
			{
				temp[i][j]=n->state[i][j];
			}
		}
		switch(k)
		{
```

```
                        case 0:
//Moving blank tile up
                                if(x != 2)
                                {
                                        temp[x][y] = temp[x+1][y];
                                        temp[x+1][y] = 0;
                                        n->child[k]=generateNode(temp,n->pathcost+1,n);
                                }
                                break;

                        case 1:
//Moving blank tile down
                                if(x != 0)
                                {
                                        temp[x][y] = temp[x-1][y];
                                        temp[x-1][y] = 0;
                                        n->child[k]=generateNode(temp,n->pathcost+1,n);
                                }
                                break;

        case 2:                                 //Moving blank tile right
           if(y != 2)
           {
              temp[x][y] = temp[x][y+1];
              temp[x][y+1] = 0;
              n->child[k]=generateNode(temp,n->pathcost+1,n);
           }
           break;

                        case 3:
//Moving blank tile left
                                if(y != 0)
                                {
                                        temp[x][y] = temp[x][y-1];
                                        temp[x][y-1] = 0;
                                        n->child[k]=generateNode(temp,n->pathcost+1,n);
                                }
                                break;

                }
        }
}
```

```
//Method to add a node to the priority queue depending on its fn value.
void addToQueue(Node* n)
{

	Node* traverser1 = front, *traverser2 = front;
	if(front==NULL)
	{
		front = n;
	}
	else
	{
		if(traverser1->fn >= n->fn)
		{
			n->next=front;
			front=n;
		}
		else
		{
			while((traverser1->fn < n->fn) && (traverser1->next != NULL))
			{
				traverser2=traverser1;
				traverser1=traverser1->next;
			}
			if((traverser1->fn > n->fn))
			{
				n->next=traverser1;
				traverser2->next=n;
			}
			else if(traverser1->next==NULL)
			{
				traverser1->next=n;
				n->next=NULL;
			}
			else
			{
				n->next=traverser1;
				traverser2->next=n;
			}
		}

	}
```

```
}

// Method to chech whether a state is repeated for not.
int isRepeated(int n[3][3])
{
   Node* temp = visited;
   while(temp != NULL)
   {
      for(int i=0;i<3;i++)
      {
         for(int j=0;j<3;j++)
         {
            if(n[i][j] != temp->state[i][j])
               return 1;
         }
      }
      temp=temp->next;
   }

   return 0;
}

// Method to print sll the states of the current node
void printStates(Node* n)
{
   if(n!=NULL)
   {
      std::cout<<"\nf(n)= "<<n->fn;
      std::cout<<"\tLevel= "<<n->pathcost;
      std::cout<<"\n";
      for(int i=0;i<3;i++)
      {
         for(int j=0;j<3;j++)
         {
            std::cout<<n->state[i][j]<<" ";
         }
         std::cout<<"\n";
      }
   }
   else
      std::cout<<"\nNULL";
}
```

```
// Method to check whether the current state is a goal state or not.
int goalTest(int n[3][3])
{
   for(int i=0;i<3;i++)
   {
      for(int j=0;j<3;j++)
      {
         if(n[i][j] != goal[i][j])
            return 0;
      }
   }
   return 1;
}


// Method to implement A* algorithm
void AStar()
{
        Node* currentNode = NULL;

        while(1)
        {
                if(front != NULL)
                {
                        currentNode=front;
                        printStates(currentNode);
                        Node* temp = front;
                        front = front->next;
                        if(visited==NULL)
                                visited=temp;
                        else
                        {
                                temp->next=visited;
                                visited=temp;
                                nodesExpanded++;
                        }
                        if(goalTest(currentNode->state)==0)
                        {
                                generateSuccessor(currentNode);
                                for(int i=0;i<4;i++)
                                {
                                        if(currentNode->child[i] != NULL)
                                        {
```

```
                                        if(currentNode->parent ==NULL)
                                        {
                                                addToQueue(currentNode->child[i]);
                                        }
                                        else if(isRepeated(currentNode->child[i]->state)==1)
                                        {
                                                addToQueue(currentNode->child[i]);
                                        }
                                }
                        }
                }
                else
                {
                        std::cout<<"\n GOAL STATE REACHED!!!";
                        std::cout<<"\n Nodes Generated = "<<nodesGenerated;
                        std::cout<<"\n Nodes Expanded = "<<nodesExpanded+1;
                        break;
                }
        }
        else
        {
                std::cout<<"\nFailure!!!";
                break;
        }
    }
}



int main()
{
    front=NULL;
    visited=NULL;

    int initial[3][3],i,j;

    std::cout<<"\nEnter Initial State (please enter '0' for empty slot):\n";
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
        {
            std::cin>>initial[i][j];
        }
```

```
	}
	std::cout<<"\nEnter Goal State (please enter '0' for empty slot):\n";
	for(i=0;i<3;i++)
	{
		for(j=0;j<3;j++)
		{
			std::cin>>goal[i][j];
		}
	}

	Node* temp = generateNode(initial,0,NULL);
	front = temp;
	AStar();
	std::cin>>i;
	return 0;
}
```

5) Execution results:

Following are the screenshots for few examples that were tested:

1)Initial State: 7 2 4
5 0 6
8 3 1

Goal State: 0 1 2
3 4 5
6 7 8

```
f(n)= 26        Level= 23
1 2 5
3 4 0
6 7 8

f(n)= 26        Level= 24
1 2 0
3 4 5
6 7 8

f(n)= 26        Level= 25
1 0 2
3 4 5
6 7 8

f(n)= 26        Level= 26
0 1 2
3 4 5
6 7 8

 GOAL STATE REACHED!!!
 Nodes Generated = 489755
 Nodes Expanded = 186089
```

2)Initial State: 2 8 3
1 6 4
7 5

Goal State: 1 2 3
8 6 4
7 5

```
f(n)= 7 Level= 4
1 2 3
0 8 4
7 6 5

f(n)= 7 Level= 5
1 2 3
8 0 4
7 6 5

f(n)= 7 Level= 6
1 2 3
8 6 4
7 0 5

f(n)= 7 Level= 7
1 2 3
8 6 4
7 5 0

 GOAL STATE REACHED!!!
 Nodes Generated = 35
 Nodes Expanded = 13
```