Report on Programming Project 1: Solving the 8-puzzle using A* algorithm:

By Abhishek Bhandwaldar

The 8 Puzzle Problem:

Problem formulation:

The 8 puzzle problem is a sliding tile puzzle. The goal of puzzle is to reach a specific board configuration from given board configuration by sliding 1 tile each turn. To solve this problem we need to explore all possible configuration from current state. We explore each configuration until we come up to a board configuration such that it matches the given goal board configuration. We use the State space tree to explore all states from current state. Each time we don't find a goal state we explore that node's successor nodes until we arrive at goal position or node.

Problem Optimization:

In above case we explore each and every node of state space tree. In typical 8 puzzle problem can have number of successor ranging from 2 to 4. Hence total number of successor nodes at each stage will increase exponentially by power of 4. This is huge set of nodes and hence this solution is not optimal. To find optimal solution we need to select only those nodes which have the best chance of finding the goal node. Hence we use the A* algorithm for this purpose. We use a heuristic function and the depth of current node to find cost and select the node of least cost.

Heuristic function:

In this assignment I have used the Manhattan distance as heuristic. Manhattan distance is distance between the tile in current node and goal node. It is the distance between the axes of the tile in two state (current and Goal). This heuristic neither overestimate the cost nor underestimates it hence is admissible.

State duplication:

When we explore nodes of state space it is possible to arrive at a state which has been already explored. Hence it is possible that the program might get stuck in a loop and never converge to goal state. To prevent this I use below methods:

Checking the ancestor node:

It is highly possible that same node as the parent's parent might get generated. This might put the program in loop and prevent it from reaching goal state. Hence I have used a pointer to parent in constructing the structure of node. This will help in finding whether current node is same as its ancestor.

Program structure:

Function used in Program:

void init(node*, node*):

This function initializes all the pointers (queue, tree, and stack). Reads input from input.txt file and sets the root node and goal node values. Assigns depth value of 0 and cost 0 to both of them.

void display(node*)

This function will display the node this is passed as argument to it.

void succ()

Generates all the successor to current nodes, calculates their depth and cost and attaches then to parent node. This function makes call to manhattan(), g_cost(), cpy() functions.

void cpy()

This function copies content of one node to another.

int manhattan(node*,node*)

Gnerates manahttan distance between two nodes which are passed as parameter.

int g_cost()

Calculates the depth (g(n))based on parent's depth.

Global Variables:

FILE *fp: File pointer to output file.

int expanded_node : Holds the number of nodes generated

int generated_node : Holds the number of nodes expanded.

struct queue_p *top : Points towards start of queue.

int q_count: contains number of elements currently present in queue.

Source Code:

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

struct node{
   int a[3][3];
   int g ;
   struct node *nxt[4];
   struct node *anc;
   int count;
   int cost;
};
typedef struct node node;
FILE *fp;
int expanded_node = 0;
int generated_node=0;
struct queue_p{
   struct node *data;
   struct queue_p *nxt;
   };
struct queue_p *top;
int q_count;
struct queue_p *visited_nodes;
void push(node*);
node* pop();

struct stack_p{
   struct node* data;
struct stack_p* nxt;
struct stack_p* bck;
};

struct stack_p *path =NULL;
void initi(node*,node*);
void display(node*);
void succ(node*,node*);
void cpy();
int stcmp(node*,node*);
int manhattan(node*,node*);
int g_cost(node*,node*);

void push_visit(node*);
```

```
int main()
{
   int f=1,k=0;
   node* ptr;
   node* goal;
   node* temp;
   struct stack_p* rear = NULL;
   ptr  = (struct node*) malloc(sizeof(struct node));
   goal  = (struct node*) malloc(sizeof(struct node));
   initi(ptr,goal);
   push(ptr);

  //Start the Game tree

    while(f)
    {
      k=0;

      temp = pop();
      if(temp==NULL)
      {
         printf("\nFatal error\n");
         break;
      }

     //  display(temp);
      printf("\n");
      succ(temp,goal);
      expanded_node++;
      if(goal_cond(goal,temp)==1)
         {
            printf("Found");
            f=0;
         }
         else
         {
            while(k<temp->count)
            {

              if(temp->anc!=NULL)
               {
                 if(stcmp(temp->nxt[k],temp->anc)!=1)
                   {
```

```
                    push(temp->nxt[k]);
                  }
               }
               else{
                        push(temp->nxt[k]);
               }

               k++;
            }

         }

   }

    fprintf(fp,"Number of nodes generated = %d\nNumber of Nodes Expanded = %d\nBest
Solution Path:\n",generated_node,expanded_node);
    path = (struct stack_p*) malloc(sizeof(struct stack_p));
    path->data = temp;
    path->bck = NULL;
    path->nxt = NULL;
    temp = temp->anc;
   while(temp!=NULL)
   {
       path->nxt = (struct stack_p*) malloc(sizeof(struct stack_p));
       rear = path;
       path = path->nxt;
       path->data = temp;
       path->bck = rear;
      temp = temp->anc;
   }
    while(path!=NULL)
    {
       display(path->data);
       path = path->bck;
    }

   return 0;

}

void initi(node* ptr, node* goal)
{
    int i,j;
    FILE *fo;
    path = NULL;
    top = NULL;
```

```
   q_count = 0;
   fo = fopen("input.txt","r");
   fp = fopen("output.txt","w");
   visited_nodes = NULL;
   //Input for Initial State. Input is take from Input.txt file.
   for(i=0;i<3;i++)
   {
         fscanf(fo,"%d %d %d",&ptr->a[i][0],&ptr->a[i][1],&ptr->a[i][2]);
   }
   ptr->g = 0;
   //Set initial cost and Depth to 0 as this is root node.
   ptr->count = 0;
   ptr->nxt[0]=NULL;
   ptr->nxt[1]=NULL;
   ptr->nxt[2]=NULL;
   ptr->nxt[3]=NULL;
   ptr->anc = NULL;

   //Input for Goal State. Goal State is taken from Input.txt
   goal->g = 0;
   for(i=0;i<3;i++)
   {
         fscanf(fo,"%d %d %d",&goal->a[i][0],&goal->a[i][1],&goal->a[i][2]);
   }
   //Set Initial Cost and Depth as 0 as this is Goal node.
   goal->count = 0;
   goal->nxt[0]=NULL;
   goal->nxt[1]=NULL;
   goal->nxt[2]=NULL;
   goal->nxt[3]=NULL;
   goal->anc = NULL;
   ptr->cost = ptr->g+manhattan(ptr,goal);
}

//This Function Displays the Board, which is passed as an Argument. The Output is written in
Output.txt file.
void display(node* ptr)
{
   int i,j;
   node* temp;
   for(i =0;i<3;i++)
   {
      fprintf(fp,"%d %d %d\n",ptr->a[i][0],ptr->a[i][1],ptr->a[i][2]);
   }
```

```
   fprintf(fp,"\n");
   j=0;
}

//This function is Responsible for Generation of Successor node for a given node.
void succ(node* ptr, node* goal)
{
   int i,j;
   int found = 0;
   node* temp;
   for(i = 0;i<3;i++)
   {
      for(j = 0;j<3;j++)
      {
         if(ptr->a[i][j]==0)
         {
            found = 1;
            break;
         }
      }
      if(found == 1)
         break;
   }

   //Successor 1 Generation and set the Depth to 1 plus that of Parent. Set the Cost = g(n) + h(n)
   temp  = (struct node*) malloc(sizeof(struct node));
   cpy(temp,ptr);
   if(i+1 <= 2)
   {
      temp->a[i][j] = temp->a[i+1][j];
      temp->a[i+1][j] = 0;
      temp->count = 0;
      temp->g = ptr->g+1;
      temp->nxt[0] = NULL;
      temp->nxt[1] = NULL;
      temp->nxt[2] = NULL;
      temp->nxt[3] = NULL;
      temp->cost = g_cost(temp,goal);
      temp->anc = ptr;
      ptr->nxt[ptr->count++] = temp;
      generated_node++;    }
   //Successor 2 Generation and set the Depth to 1 plus that of Parent. Set the Cost = g(n) + h(n)
   temp  = (struct node*) malloc(sizeof(struct node));
   cpy(temp,ptr);
```

```
if(i-1>=0)
{
   temp->a[i][j] = temp->a[i-1][j];
   temp->a[i-1][j] = 0;
   temp->count = 0;
   temp->g = ptr->g+1;
   temp->nxt[0] = NULL;
   temp->nxt[1] = NULL;
   temp->nxt[2] = NULL;
   temp->nxt[3] = NULL;
   temp->anc = ptr;
   temp->cost = g_cost(temp,goal);
   ptr->nxt[ptr->count++] = temp;
   generated_node++;
}

//Successor 3 Generation and set the Depth to 1 plus that of Parent. Set the Cost = g(n) + h(n)
temp  = (struct node*) malloc(sizeof(struct node));
cpy(temp,ptr);
if(j+1<=2)
{
   temp->a[i][j] = temp->a[i][j+1];
   temp->a[i][j+1] = 0;
   temp->count = 0;
   temp->g = ptr->g+1;
   temp->nxt[0] = NULL;
   temp->nxt[1] = NULL;
   temp->nxt[2] = NULL;
   temp->nxt[3] = NULL;
   temp->anc = ptr;
   temp->cost = g_cost(temp,goal);
   ptr->nxt[ptr->count++] = temp;
   generated_node++;
}

//Successor 4 Generation and set the Depth to 1 plus that of Parent. Set the Cost = g(n) + h(n)
temp  = (struct node*) malloc(sizeof(struct node));
cpy(temp,ptr);
if(j-1>=0)
{
   temp->a[i][j] = temp->a[i][j-1];
   temp->a[i][j-1] = 0;
   temp->count = 0;
   temp->g = ptr->g+1;
```

```
      temp->nxt[0] = NULL;
      temp->nxt[1] = NULL;
      temp->nxt[2] = NULL;
      temp->nxt[3] = NULL;
      temp->anc = ptr;
      temp->cost = g_cost(temp,goal);
      ptr->nxt[ptr->count++] = temp;
      generated_node++;
   }
}

int g_cost(node* temp, node* goal)
{
   return(temp->g + manhattan(temp,goal));
}

//This function is Used for copying the value of 1 node to Other
void cpy(node* temp, node* ptr)
{
   int i,j;
   for(i = 0;i<3;i++)
   {
      for(j = 0;j<3;j++)
      {
         temp->a[i][j] = ptr->a[i][j];
      }
   }
}

//This function is used for Comparison of One Node to another.
int stcmp(node* ptr, node* temp)
{
   int i,j;
   for(i=0;i<3;i++)
   {
      for(j=0;j<3;j++)
      {
         if(ptr->a[i][j]!=temp->a[i][j])
         {
            return 0;
         }
      }
   }
   return 1;
```

```
}

//This Function is Responsible for Cal calculation of Heuristic function h(n)
int manhattan(node *ptr, node *temp)
{
   int i=0,j=0,found=0,sum=0,k=0,l=0;
   for(i = 0;i<3;i++)
   {
      for(j=0;j<3;j++)
      {
         if(ptr->a[i][j]!=0)
         {
         for(k = 0;k<3;k++)
         {

            for(l = 0;l<3;l++)
            {
               if(ptr->a[i][j]==temp->a[k][l])
               {
                  found = 1;
                  break;
               }
            }
            if(found==1)
               {
                  found=0;
                  break;
               }
         }
         sum = sum + (abs(k-i) + abs(l-j));
         }
      }
   }
   return sum;
}

//This is the Queue Push function. Based on the Cost the push() function will inert the node such
that all nodes with greater cost are below that node.
void push(node* temp)
{
  struct queue_p *ptr , *ptr_bck, *t;

   if(top==NULL)
   {
```

```
   top = (struct queue_p*) malloc(sizeof(struct queue_p));
   top->data = temp;
   top->nxt=NULL;
}
else
{
   ptr = top;
   while(ptr!=NULL)
   {
      if(stcmp(ptr,temp)==1)
      {
         if(ptr->data->cost > temp->cost)
         {
            ptr->data = temp;
            return;
         }
         else
            return;
      }
      ptr = ptr->nxt;
   }
      if(temp->cost < top->data->cost)
      {
         ptr = (struct queue_p*) malloc(sizeof(struct queue_p));
         ptr->nxt = top;
         ptr->data = temp;
         top = ptr;
      }
      else
      {
      //Check which node cost is greater than temp node
         ptr = top;
         while(ptr!=NULL)
         {
            if(temp->cost < ptr->data->cost)
         {
            break;
         }
         ptr_bck = ptr;
         ptr = ptr->nxt;
         }
      //if ptr==NULL than we are at end of list and hence proceed with insertion at end of list
         if(ptr==NULL)
         {
```

```
            ptr_bck->nxt = (struct queue_p*) malloc(sizeof(struct queue_p));
            ptr_bck = ptr_bck->nxt;
            ptr_bck->data = temp;
            ptr_bck->nxt = NULL;
            }
            else{
               //Inert the node in between two elements
               ptr_bck->nxt = (struct queue_p*) malloc(sizeof(struct queue_p));
               ptr_bck = ptr_bck->nxt;
               ptr_bck->nxt = ptr;
               ptr_bck->data = temp;
            }
         }
   }
   q_count++;
}

//This Function checks the Goal condition
int goal_cond(node* ptr, node* temp)
{
   if(manhattan(ptr,temp)==0)
      return 1;
   else
      return 0;
}

//This function is used to return the top value of Queue
node* pop()
{
   q_count--;
  // printf("%d  ",q_count);
   node* temp;
   if(top==NULL)
      return NULL;
   temp = top->data;
   top = top->nxt;
   return temp;
}



//This function is used to add visited nodes in Visited nodes list.
void push_visit(node* temp)
{
```

```
   struct queue_p *ptr;
    if(visited_nodes==NULL)
   {
      visited_nodes = (struct queue_p*) malloc(sizeof(struct queue_p));
      visited_nodes->data = temp;
      visited_nodes->nxt=NULL;
   }
   else
   {
      ptr = visited_nodes;
      while(ptr->nxt!=NULL)
         ptr = ptr->nxt;
      ptr->nxt = (struct queue_p*) malloc(sizeof(struct queue_p));
      ptr = ptr->nxt;
      ptr->nxt = NULL;
      ptr->data = temp;
   }
}

// This function checks whether a give node matches the nodes in Visited nodes queue.
int check_visited_nodes(node* temp)
{
   struct queue_p* ptr;
   ptr = visited_nodes;
   if(ptr==NULL)
      return 0;
   while(ptr->nxt!=NULL)
   {
      if(stcmp(temp,ptr->data)==1)
         return 1;

      ptr = ptr->nxt;
   }
   return 0;
}
```

Result:

1. Input 1

Initial state->	2 8 3	Goal State->	1 2 3
	1 6 4		8 6 4
	7 0 5		7 5 0

Output:

Number of nodes generated = 31
Number of Nodes Expanded = 11
Best Solution Path:

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

1 2 3
8 6 4
7 0 5

1 2 3
8 6 4
7 5 0

2. Input 2

Initial state->	5 4 0	Goal->	1 2 3
	6 1 8		0 4 5
	7 3 2		6 7 8

Output:

Number of nodes generated = 1546
Number of Nodes Expanded = 574
Best Solution Path:

5 4 0
6 1 8
7 3 2

5 0 4
6 1 8
7 3 2

5 1 4
6 0 8
7 3 2

5 1 4
6 3 8
7 0 2

5 1 4
6 3 8
7 2 0

5 1 4
6 3 0
7 2 8

5 1 4
6 0 3
7 2 8

5 1 4
6 2 3
7 0 8

5 1 4
6 2 3
0 7 8

5 1 4
0 2 3
6 7 8

0 1 4
5 2 3
6 7 8

1 0 4
5 2 3
6 7 8

1 2 4
5 0 3
6 7 8

1 2 4
0 5 3
6 7 8

0 2 4
1 5 3
6 7 8

2 0 4
1 5 3
6 7 8

2 4 0
1 5 3
6 7 8

2 4 3
1 5 0
6 7 8

2 4 3
1 0 5
6 7 8

2 0 3
1 4 5
6 7 8

0 2 3
1 4 5
6 7 8

1 2 3
0 4 5
6 7 8

3. Input 3

Initial->	7 2 4	Goal->	0 1 2
	5 0 6		3 4 5
	8 3 1		6 7 8

Output:

Number of nodes generated = 17806
Number of Nodes Expanded = 6700
Best Solution Path:
7 2 4
5 0 6
8 3 1

7 2 4
0 5 6
8 3 1

0 2 4
7 5 6
8 3 1

2 0 4
7 5 6
8 3 1

2 5 4
7 0 6
8 3 1

2 5 4
7 3 6
8 0 1

2 5 4
7 3 6
0 8 1

2 5 4
0 3 6
7 8 1

2 5 4
3 0 6
7 8 1

2 5 4
3 6 0
7 8 1

2 5 0
3 6 4
7 8 1

2 0 5
3 6 4
7 8 1

0 2 5
3 6 4
7 8 1

3 2 5
0 6 4
7 8 1

3 2 5
6 0 4
7 8 1

3 2 5
6 4 0
7 8 1

3 2 5
6 4 1
7 8 0

3 2 5
6 4 1
7 0 8

3 2 5
6 4 1
0 7 8

3 2 5
0 4 1
6 7 8

3 2 5
4 0 1
6 7 8

3 2 5
4 1 0
6 7 8

3 2 0
4 1 5
6 7 8

3 0 2
4 1 5
6 7 8

3 1 2
4 0 5
6 7 8

3 1 2
0 4 5
6 7 8

0 1 2
3 4 5
6 7 8