Distinction in Complexities of Heuristic and Non – Heuristic Approach Using 8-Tile Puzzle Problem

A Project Report Submitted

for the Course of

Minor Project - I
In

Third year – Fifth Semester of

Bachelor of Technology

In

Computer Science Engineering

With Specialization

In

Artificial Intelligence and Machine Learning
Under

Mr. Bikram Pratim Bhuyan

Assistant professor

Department of Informatics

By

500068878	**R177218054**	**Maansi Singhal**
500068858	**R177218085**	**Shrey Bansal**
500067783	**R177218086**	**Shreya Joshi**

UPES

UNIVERSITY WITH A PURPOSE

DEPARTMENT OF INFORMATICS

SCHOOL OF COMPUTER SCIENCE

UNIVERSITY OF PETROLEUM AND ENERGY STUDIES, BIDHOLI, DEHRADUN, UTTRAKHAND, INDIA

INDEX

Project Title:

Distinction in complexities of Heuristic and non – Heuristic approach using 8-Tile Puzzle problem

Abstract:

8-tile puzzle problem, normally called 8-puzzle problem is a sliding puzzle problem. It consists of the frame (3x3) in which numbers are placed randomly from (1-8) with a blank tile. The bigger picture of this problem is the N-puzzle problem, which is solved using graph – search technique. 8-puzzle problem is a prominent workbench model for analysing the performance of a heuristic algorithm. To find the optimal solution of an N-puzzle problem is np-Hard. In this investigation, we will be analysing the performance of informed (heuristic) and uninformed (non-heuristic) search algorithm on the 8-puzzle problem to find an optimal solution to this problem. The 8-puzzle problem is converted into a game - search tree and then heuristic/non-heuristic approach is applied to it.

The heuristic search method is an intelligent method, which allows us to take decisions to reach from the current node the goal state in optimal manner. On the other hand, the latter search method looks for all possible solutions without additional information about the search space.

Introduction:

8 puzzle problem uses a 3 by 3 grid board with eight inlaid square sliding pieces marked with values 1 through 8 along with a blank tile. The blank tile can move one step at a time. The aim of this puzzle is to arrange the numbers in the grid according to the desired output with the help of legal moves.

8-puzzle problem is the largest possible N-puzzle problem that could be solved completely. The general extension of this 8-puzzle is the N-puzzle which is an np-hard problem.

Literature Review:

- ❑ Alexander Reinefeld [1] investigated 8-puzzle problem and his results include data on the expected solution lengths, the 'easiest' and 'worst' configurations, and the density and distribution of solution nodes in the search tree.

- ❑ Daniel R. Kunkle [2] applied A* algorithm which guarantees that the best solution (that with the least number of moves) will be found. Heuristics are examined to allow the algorithm to find the optimal solution while examining as few states as possible (maximizing the informedness of the heuristic).

- ❑ Masuma Sultana et. al. [3] provide a complete solution of 8-puzzle problem using BFS in Compute Unified Device Architecture (CUDA) Environment. Objective of their work is to find the complete solution of 8-puzzle problem i.e. examining all the permutations for solvability. Using Breadth First search (BFS) graph traversal, we can reach the solution for a definite goal. The parallel algorithm is capable of providing us with much faster solution using CUDA. HuaShi [4] conducted a thorough analysis of the search solution for the 8-puzzle problem. The Breadth-first search algorithm, the Depth-first search algorithm and A *algorithm are used to solve the problem.

Objective:

- Comparison between the complexities of heuristic and non-heuristic algorithms such as A*, Branch and Bound (heuristic) and Breadth First Search (non-heuristic).
- Understanding C language and Data-structures

Methodology:

- In general, a typical 8-puzzle problem may take about 20 steps to solve, but it highly depends on the initial state of the problem. The movement of blank tile depends on its position: four possible moves if it is in middle, two possible moves if it is in corner and three along the edge, which makes the branching factor approximately 3.
- The Algorithms which we have used are: BFS (non-heuristic), A*(heuristic) and Branch and Bound (heuristic). It is to show the comparison between heuristic and non-heuristic approaches to solve the problem.
- **BFS** algorithm traverse a tree in a breadth wise motion and uses queue to remember to get the next vertex to start a search. This algorithm selects a single node (initial or source point) and then visits all the nodes adjacent to the selected node. Once the algorithm visits and marks the starting node, then it moves towards the nearest unvisited nodes and analyses them.
- **A*** Search Algorithm at each step picks the node according to a value **f,** which is a parameter equal to the sum of two other parameters **g** and **h**. At each step, it picks the node having the lowest **f** and process that node. Here **g** is the movement cost to move from the starting point to a given square on the grid, following the path generated to get there and **h is** the estimated movement cost to move from that given square on the grid to the final state.
- **Branch and Bound** heuristic algorithm depends on upper bound and lower bound regions of search space. There are three types of nodes:
  1. **Live node:** It is a node that has been generated but whose children have not yet been generated.
  2. E-node: It is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
  3. Dead node: It is a generated node, that is not to be expanded or explored any further. All children of a dead node have already been expanded.
- Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as:
- $$c(x) = f(x) + h(x),$$
- where f(x) is the length of the path from root to x (the number of moves so far) and h(x) is the number of non-blank tiles not in their goal position (the number of misplaced tiles). There are at least h(x) moves to transform state x to a goal state.

Algorithm:

BFS:

STEP 1: Create a structure NODE. Declare a matrix and variables x , y and level ,cost and pointer as next.

STEP 2: Create a structure Queue which has pointers as front and rear.

STEP 3: Create a function Create_Queue which returns q node and initialize the front and rear to NULL.

STEP 4: Create a function Push to push the node into the Queue.

STEP 5: Create a function Display_Matrix to display the node of the tree.

STEP 6: Create a function Pop to pop the front node from the Queue.

STEP 7: Create a function Swap to swap two nodes.

STEP 8: Create a function NewNode to create any new node .

STEP 9: Create a function Cal_Cost to count no of misplaced tiles from the **goal node.**

STEP 10: Create a function is_safe to check whether the node is safe or not . if the node is safe .

STEP 11: Create a function Display path to print the path from the root node to goal node.

STEP 12: Create a function Solve to solve the puzzle.

STEP 13: In the solve function we would push root node to queue.

STEP 14: pop the node from QUEUE and check whether it is equal to final node . If it is equal to final node then go to step 16 else make it live node to explore its children.

STEP 15: check whether the child node is safe and calculate its cost . If safe , then push the node to queue .

STEP 16: Display the path and matrix from root to goal node.

A-STAR:

STEP 1: Create a structure node_array. Declare a matrix and variables depth , text and cost and pointers as right, left ,top, down, parent and next.

STEP 2: Intialize the front and rear of the Queue to NULL and Front_list and Rear_list to NULL.

STEP3: Input goal and intial matrix .

STEP 4: Calculate heuristic function using Manhattan distance function (mod) and return its value to f variable.

STEP 5: Call the A_star function by passing the root node.

STEP 6: In A_star Function insert the root to list and queue using functions Insert_list and Insert_Queue respectively.

STEP 7: If the current node is equal to the goal node then print the path from root to goal state else go to step 8.

STEP 8: Explore the children of the current node with the help of function next_move .

STEP 9: Push the child nodes into queue and list.

STEP 10: Now prioritize the nodes in the queue according to their minimum costs with the help of the function Arrange.

STEP 11: Go to Step 4

STEP 12: Now print the number of steps taken to reach goal state

BRANCH & BOUND

```
struct list_node
{
  list_node *next;
  // Helps in tracing path when answer is found
  list_node *parent;
  float cost;
}
algorithm LCSearch(list_node *t)
{
  // Search t for an answer node
  // Input: Root node of tree t
  // Output: Path from answer node to root
```

```
  if (*t is an answer node)
  {
     print(*t);
     return;
  }
  E = t; // E-node
  Initialize the list of live nodes to be empty;
  while (true)
  {
    for each child x of E
    {
       if x is an answer node
       {
         print the path from x to t;
         return;
       }
       Add (x); // Add x to list of live nodes;
       x->parent = E; // Pointer for path to root
    }
    if there are no more live nodes
    {
      print ("No answer node");
      return;
    }
// Find a live node with least estimated cost
    E = Least();
    // The found node is deleted from the list of
    // live nodes
  }
}
```

Result:

Final

1	2	3
8	0	4
7	6	5

1	3	4
8	6	2
7	0	5

Easy

2	8	1
0	4	3
7	6	5

` **Medium**

5	6	7
4	0	8
3	2	1

Worst

Algorithm	Difficulty level	Solution Length	Executed or not	Execution time
BFS(non-heuristic)	Easy	5	executed	0
	Medium	12	executed	0.001s
	Worst	-	TLE	-
A-Star(heuristic)	Easy	5	executed	0
	Medium	9	executed	0
	Worst	30	executed	0.88s
Branch&Bound	Easy	5	executed	0
	Medium	14	executed	0.003s
	Worst	-	TLE	-

Result Analysis:

The aim of the project was to show distinction between the complexities for solving the 8 puzzle problem.

Both the heuristic and non heuristic algorithms are implemented to prove the efficiency of the algorithms.

For the verification of the puzzle solvability A*, BFS and Branch and Bound are implemented and verified.

From the above result we can analyze the following points:

- A* algorithm which is heuristic works for all kinds of difficulty level of puzzle.
- For the medium difficulty level three of the algorithms works fine but branch and bound and BFS took more time.
- Also the length of the solution in A* algorithm is much smaller than BFS.

So we can finally assess that A* which is pure heuristic algorithm does a great job in solving this 8 puzzle problem. It provided us optimal solution for every test case with great efficiency.

Conclusion:

As we can see from the results, A* saves more time and space than BFS and Branch and Bound search, which reflects the theoretical comparison between the three algorithms' time complexities. In fact, while A* solved all three test cases, BFS and Branch and Bound didn't due to the memory requirement not being met by our computer. Therefore, we conclude that A* search algorithm is more efficient than BFS and Branch and Bound.

References:

[1]Alexander Reinfeld, “Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA*”, Paderborn center for parallel computing, wax burger str.100, D-33095 Paderborn, Germany.

[2] Daniel R. Kunkle, “Solving the 8 Puzzle in a Minimum Number of Moves: An Application of the A* Algorithm”, College of Computing and Information Sciences, Rochester Institute of Technology, assessed in October 08, 2001.

[3] Masuma Sultana, Rathindra Nath Dutta & S. K. Setua “Complete Solution of Eight Puzzle Problem using BFS in CUDA Environment”, IEEE International WIE Conference on Electrical and Computer Engineering, 19-20 December, 2015

[4]Shi & Hua.“Searching Algorithms Implementation and Comparison of Eight-Puzzle Problem”, International Conference in Computer Science and Network Technology (ICCSNT), Proceedings, IEEE Xplore, Vol. 2, (pp.1203-1206), 2011.

Appendix:

A*

```
#include<stdio.h>
#include<malloc.h>
#include<string.h>

typedef struct node_array
{
    int a[3][3];
    int depth;
    int f;
    char text[11];
    struct node_array *shift_top;
    struct node_array *shift_down;
    struct node_array *shift_left;
    struct node_array *shift_right;
    struct node_array *parent;
    struct node_array *next;
} node;

node *front=NULL,*rear=NULL,*front_list=NULL,*rear_list=NULL;

int goal[3][3],depth_flag=0,f_count=0;

void set_zero(int a[3][3])
{
    int i,j;
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            a[i][j]=0;
}

void copy_array(int a[3][3],int b[3][3])//copy b to a
{
    int i,j;
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            a[i][j]=b[i][j];
}

int is_equal(int a[3][3],int b[3][3])
{
    int i,j,flag=1;
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            if(a[i][j]!=b[i][j])
                flag=0;
    return flag;
}

void swap(int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}
```

```c
void print_array(int a[3][3])
{
    int i,j;
    for(i=0;i<3;i++)
    {                 for(j=0;j<3;j++)
            printf("%d ",a[i][j]);
        printf("\n");
    }
}

int check_zero(int a[3][3])//returns 1 if 0
{
    int i,j,flag=1;
    for(i=0;i<3;i++)
    {                 for(j=0;j<3;j++)
            if(a[i][j]!=0)
                flag=0;
    }
    return flag;
}

int check_list(node *nd)//returns 0 if it doesnt match the list
{
    node *new_node=NULL;
    if(front_list==NULL)
        return 0;
    else
    {
        new_node=front_list;
        while(new_node!=NULL)
        {
            if(is_equal(new_node->a,nd->a)==1)
                return 1;
            new_node=new_node->next;
        }
        return 0;
    }
}


void insert_queue(node *nd)
{
    nd->next=NULL;
    if(front==NULL)
    {
        front=nd;
        rear=nd;
    }
    else
    {
        rear->next=nd;
        rear=nd;
        rear->next=NULL;
    }
}

void insert_list(node *nd)
{
    nd->next=NULL;
    if(front_list==NULL)
    {
```

```
          front_list=nd;
          rear_list=nd;
     }
     else
     {
          rear_list->next=nd;
          rear_list=nd;
          rear_list->next=NULL;
     }
}


int mod(int a)
{
     if(a>=0)
          return a;
     else
          return (-1*a); // To make cost positive
}

int calc_heuristic(node *nd)
{
     int i,j,k,l,h=0;
     for(i=0;i<=2;i++)
     {
          for(j=0;j<=2;j++)
          {
               if(nd->a[i][j]!=0)
               {for(k=0;k<=2;k++)
                    {
                         for(l=0;l<=2;l++)
                         {
                              if(nd->a[i][j]==goal[k][l])
                                   goto xy;
                         }
                    }
                    xy:
                    h+=(mod(i-k)+mod(j-l)); //Manhatton Distance
                    // Those tiles which are matching with the goal state tiles
               }
          }
     }
     return h;
}


void next_move(node *nd)
{
     int i,j,x,y;
     for(i=0;i<3;i++)
     {
          for(j=0;j<3;j++)
          {
               if(nd->a[i][j]==0)   //searching for blank tile coordinates
                    goto label;
          }
     }
     label:
     // Blank tile coordinaes
     x=i;
     y=j;
```

```
    if(y+1 >2)
        nd->shift_right=NULL;  //This move is not possible
    else
    {
        nd->shift_right=(node*)malloc(sizeof(node));
        copy_array(nd->shift_right->a,nd->a);
        swap(&nd->shift_right->a[x][y],&nd->shift_right->a[x][y+1]);
        if(check_list(nd->shift_right)==1)
        {
            nd->shift_right=NULL;
            free(nd->shift_right);
        }
        else
        {
            nd->shift_right->parent=nd;
            nd->shift_right->depth=nd->depth+1;
            strcpy(nd->shift_right->text,"move right");
            nd->shift_right->f=nd->depth+calc_heuristic(nd->shift_right)+1;
// f(n) =g(n) +h(n)
            insert_list(nd->shift_right);
            insert_queue(nd->shift_right);
        }
    }
    if(y-1 <0)
        nd->shift_left=NULL; //This move is not possible
    else
    {
        nd->shift_left=(node*)malloc(sizeof(node));
        copy_array(nd->shift_left->a,nd->a);
        swap(&nd->shift_left->a[x][y],&nd->shift_left->a[x][y-1]);
        if(check_list(nd->shift_left)==1)
        {
            nd->shift_left=NULL;
            free(nd->shift_left);
        }
        else
        {
            nd->shift_left->parent=nd;
            strcpy(nd->shift_left->text,"move left");
            nd->shift_left->depth=nd->depth+1;
            nd->shift_left->f=nd->depth+calc_heuristic(nd->shift_left)+1;//
f(n) =g(n) +h(n)
            insert_list(nd->shift_left);
            insert_queue(nd->shift_left);
        }
    }
    if(x+1 >2)
        nd->shift_down=NULL; //This move is not possible
    else
    {
        nd->shift_down=(node*)malloc(sizeof(node));
        copy_array(nd->shift_down->a,nd->a);
        swap(&nd->shift_down->a[x][y],&nd->shift_down->a[x+1][y]);
        if(check_list(nd->shift_down)==1)
        {
            nd->shift_down=NULL;
            free(nd->shift_down);
        }
        else
        {
            nd->shift_down->parent=nd;
```

```
            strcpy(nd->shift_down->text,"move down");
            nd->shift_down->depth=nd->depth+1;
            nd->shift_down->f=nd->depth+calc_heuristic(nd->shift_down)+1;//
f(n) =g(n) +h(n)
            insert_list(nd->shift_down);
            insert_queue(nd->shift_down);
        }
    }
    if(x-1 <0)
        nd->shift_top=NULL; //This move is not possible
    else
    {
        nd->shift_top=(node*)malloc(sizeof(node));
        copy_array(nd->shift_top->a,nd->a);
        swap(&nd->shift_top->a[x][y],&nd->shift_top->a[x-1][y]);

        if(check_list(nd->shift_top)==1)
        {
            nd->shift_top=NULL;
            free(nd->shift_top);
        }
        else
        {
            nd->shift_top->parent=nd;
            strcpy(nd->shift_top->text,"move up");
            nd->shift_top->depth=nd->depth+1;
            nd->shift_top->f=nd->depth+calc_heuristic(nd->shift_top)+1;//
f(n) =g(n) +h(n)
            insert_list(nd->shift_top);
            insert_queue(nd->shift_top);
        }
    }
}
// Function to check if initial state is the goal state
int is_goal(node *nd) //return 1 if goal
{
    return is_equal(nd->a,goal);
}

node *pop_queue()
{
    node *nd;
    nd=front;
    front=front->next;
    if(front==NULL)
        rear=NULL;
    return nd;
}

void final_print(node *nd)
{
    while(nd!=NULL)
    {
        printf("\n");
        print_array(nd->a);
        printf("\n\n%s\n",nd->text);
        nd=nd->parent; // for path

        f_count++;
    }
    printf("\nNo of steps %d",f_count-1);
```

```
}

int count_queue()
{
    int count=0;
    node *temp;
    temp=front;
    while(temp!=NULL)
    {
        temp=temp->next;
        count++;
    }
    return count;
}
void arrange()
{
    node *t1,*t2,*t3;
    int i,j;
    i=count_queue();
    j=i;
    for(int k=0;k<i;k++)
    {
        t1=front; //initial
        t2=front->next; //left

        for(int l=1;l<j;l++)
        {
            if(t1->next->f > t2->next->f)  //t2->left is middle
            {
                t3=t2; //t3 is left
                t1->next=t2->next; // put middle value to left
                t3->next=t3->next->next;
                t1->next->next=t2;
            }
            j--;
            t1=t1->next;
            t2=t2->next;
        }
    }
    t1=front;
    while(t1->next!=NULL)
        t1=t1->next;
    rear=t1;
    rear_list=t1;
}


void A_star(node *root)
{

    {
        node *nd;
        insert_queue(root);
        insert_list(root);
        nd=root;
        while(1)
        {
            nd=front;
            if(is_goal(nd)==1)
            {
                final_print(nd);
```

```
                break;
            }
            next_move(nd);
            arrange(); // function acting as  priority queue
            nd=pop_queue(); //popping live node
        }
    }
}




int main()
{
    node *n;
    int i,j;
    n=(node*)malloc(sizeof(node));

    printf("Enter the goal state\n");
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)
            scanf("%d",&n->a[i][j]);

    n->parent=NULL;
    n->next=NULL;
    n->shift_top=NULL;
    n->shift_down=NULL;
    n->shift_left=NULL;
    n->shift_right=NULL;
    n->depth=0;


    printf("Enter the current state\n");
    for(i=0;i<3;i++)
    for(j=0;j<3;j++)
        scanf("%d",&goal[i][j]);

    n->f=calc_heuristic(n);
    printf("\nThe seq of steps are\n",f_count);
    A_star(n);

    return 0;
}
```

For BFS:

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

//#include "state.h"
//#include "list.h"
//#include "node.h"
//#include "io.h"
////////////////////////////
#define BLANK_CHARACTER '0'

//this enumerates available movements in the game relative to the blank
```

```c
character
typedef enum Move {
    UP, DOWN, LEFT, RIGHT, //values for moving up, down, left, right,
respectively
    NOT_APPLICABLE         //value assigned for initial and goal input
states
} Move;

typedef struct State {
    Move action;           //action that resulted to `this` board state
    char board[3][3];      //resulting board configuration after applying
action
} State;

/**
 * DESCRIPTION:
 *    This creates a state if `move` is a valid move of the `state` board
state.
 * PARAMETERS:
 *    state - pointer to the initial state
 *    move  - action to be applied to the given state
 * RETURN:
 *    Returns a pointer to a new `State` after the move is applied. Returns
NULL
 *    upon failure.
**/
State* createState(State *state, Move move) {
    State *newState = malloc(sizeof(State));

    //copy the board configuration of `state` to `newState`
    //while searching for the row and column of the blank character
    char i, j;        //used for traversing the 3x3 arrays
    char row, col;    //coordinates of the blank character

    for(i = 0; i < 3; ++i) {
        for(j = 0; j < 3; ++j) {
            if(state->board[i][j] == BLANK_CHARACTER) {
                row = i;
                col = j;
            }

            newState->board[i][j] = state->board[i][j];
        }
    }

    //test if the coordinates are valid after translation based on the move
    //if it is, swap the concerned board values to reflect the move
    if(move == UP && row - 1 >= 0) {
        char temp = newState->board[row - 1][col];
        newState->board[row - 1][col] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = UP;
        return newState;
    }
    else if(move == DOWN  && row + 1 < 3) {
        char temp = newState->board[row + 1][col];
        newState->board[row + 1][col] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = DOWN;
        return newState;
    }
```

```c
    else if(move == LEFT  && col - 1 >= 0) {
        char temp = newState->board[row][col - 1];
        newState->board[row][col - 1] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = LEFT;
        return newState;
    }
    else if(move == RIGHT && col + 1 < 3) {
        char temp = newState->board[row][col + 1];
        newState->board[row][col + 1] = BLANK_CHARACTER;
        newState->board[row][col] = temp;
        newState->action = RIGHT;
        return newState;
    }

    free(newState);
    return NULL;
}

/**
 * DESCRIPTION: This frees memory of `s` and is reassigned to NULL
**/
void destroyState(State **state) {
    free(*state);
    state = NULL;
}

/**
 * DESCRIPTION:
 *    A heuristic function that assigns h-cost to nodes. Lower values
 *    indicate the more closer to the goal.
 * PARAMETER:
 *    curr - the current board configuration
 *    goal - a pointer to the goal configuration of the board
 * RETURN:
 *    Returns a heuristic value greater than or equal to 0.
**/
int manhattanDist(State * const curr, State * const goal) {
    int x0, y0; //used for indexing each symbol in `curr`
    int x1, y1; //correspoinding row and column of symbol from curr[y0, x0]
at `goal`
    int dx, dy; //change in x0 and x1, and y0 and y1, respectively
    int sum = 0;

    //for each symbol in `curr`
    for(y0 = 0; y0 < 3; ++y0) {
        for(x0 = 0; x0 < 3; ++x0) {
            //find the coordinates of the same symbol in `goal`
            for(y1 = 0; y1 < 3; ++y1) {
                for(x1 = 0; x1 < 3; ++x1) {
                    if(curr->board[y0][x0] == goal->board[y1][x1]) {
                        dx = (x0 - x1 < 0)? x1 - x0 : x0 - x1;
                        dy = (y0 - y1 < 0)? y1 - y0 : y0 - y1;
                        sum += dx + dy;
                    }
                }
            }
        }
    }

    return sum;
```

```c
}

/**
 * DESCRIPTION:
 *    This checks whether the two given states match against each other. An
alternative
 *    way is to check if the manhattan distance between these two states is
zero, but it
 *    would take a more significant amount of time and computational
resource
 * PARAMETERS:
 *    state1 - state to match against `state2`
 *    state2 - state to match against `state1`
 * RETURN:
 *    Returns 1 if states match, 0 otherwise.
**/
char statesMatch(State const *testState, State const *goalState) {
    char row = 3, col;

    while(row--) {
        col = 3;
        while(col--) {
            if(testState->board[row][col] != goalState->board[row][col])
                return 0;
        }
    }

    return 1;
}
///////////////////////////
/////////////////////////////
typedef struct Node Node;
typedef struct NodeList NodeList;

/**
 * DESCRIPTION:
 *    The node in the linked list. Note that as a convention, the previous
 *    node of the list's head is NULL and so is the next node of the list's
 *    tail.
**/
typedef struct ListNode {
    Node *currNode;
    struct ListNode *prevNode; //the node before `this` instance
    struct ListNode *nextNode; //the next node in the linked list
} ListNode;

/**
 * DESCRIPTION:
 *    Contains the linked list of nodes, as well as other list information.
**/
struct NodeList {
    unsigned int nodeCount;    //the number of nodes in the list
    ListNode *head;            //pointer to the first node in the list
    ListNode *tail;            //pointer to the last node in the list
};

/**
 * DESCRIPTION:
 *    A structure for holding the solution.
**/
typedef struct SolutionPath {
```

```c
    Move action;
    struct SolutionPath *next;
} SolutionPath;

/**
 * DESCRIPTION:
 *    This function is used to deallocate a list of type `SolutionPath`.
**/
void destroySolution(SolutionPath **list) {
    SolutionPath *next;
    while(*list) {
        next = (*list)->next;
        free(*list);
        *list = next;
    }
    *list = NULL;
}

/**
 * DESCRIPTION:
 *    This function pushes a node to the list of nodes.
 * PARAMETER:
 *    node - the node to add to the list
 *    list - a pointer to the list pointer to add the node into
 * RETURN:
 *    Returns 1 on success, 0 on failure.
**/
char pushNode(Node *node, NodeList** const list) {
    if(!node)
        return 0;

    ListNode *doublyNode = malloc(sizeof(ListNode));
    if(!doublyNode)
        return 0;

    doublyNode->currNode = node;

    if(*list && !(*list)->nodeCount) {
        (*list)->head = doublyNode;
        (*list)->tail = doublyNode;
        doublyNode->nextNode = NULL;
        doublyNode->prevNode = NULL;
        ++(*list)->nodeCount;
        return 1;
    }

    if(*list == NULL) {
        *list = malloc(sizeof(NodeList));
        if(*list == NULL)
            return 0;

        (*list)->nodeCount = 0;
        (*list)->head = NULL;
        (*list)->tail = doublyNode;
    }
    else {
        (*list)->head->prevNode = doublyNode;
    }

    doublyNode->nextNode = (*list)->head;
    doublyNode->prevNode = NULL;
```

```
    (*list)->head = doublyNode;

    ++(*list)->nodeCount;

    return 1;
}

/**
 * DESCRIPTION:
 *    This detatchs the node at the tail of the list. The previous
 *    node of the next node to detatch will be set to NULL.
 * PARAMETER:
 *    list - the list to pop
 * RETURN:
 *    Returns the address of the dettatched node; NULL if the list
 *    is empty.
**/
Node* popNode(NodeList** const list) {
    if(!*list || (*list)->nodeCount == 0)
        return NULL;

    Node *popped = (*list)->tail->currNode;
    ListNode *prevNode = (*list)->tail->prevNode;

    //free the list node pointing to node to be popped
    free((*list)->tail);

    if((*list)->nodeCount == 1) {
        (*list)->head = NULL;
    }
   else {
      prevNode->nextNode = NULL;
   }

    (*list)->tail = prevNode;
    --(*list)->nodeCount;
    return popped;
}

/**
 * DESCRIPTION:
 *    This function adds a list of nodes to `list`. It connects
 *    the head node of the list to be appended, to the tail of the list
 *    to append to. Connecting a list to itself will be ignored. After
 *    the connection, `toAppend` will be assigned to NULL. Deallocation of
 *    nodes from `toAppend` will be handled from `list`. This is mainly
 *    to avoid dangling pointer, or calling free multiple times to the
 *    same address.
 * PARAMETERS:
 *    toAppend - the list to be appended
 *    list     - the list to append `toAppend` into
**/
void pushList(NodeList **toAppend, NodeList *list) {
    //if either of the list is NULL, the head of the list to be appended is
NULL,
    //or the list points to the same starting node
    if(!*toAppend || !list || !(*toAppend)->head || (*toAppend)->head ==
list->head) {
        return;
    }
```

```
    //if the list to append to has currently no element
    if(!list->nodeCount) {
        list->head = (*toAppend)->head;
        list->tail = (*toAppend)->tail;
    }
    else {
        //connect the lists
        (*toAppend)->tail->nextNode = list->head;
        list->head->prevNode = (*toAppend)->tail;
      list->head = (*toAppend)->head;
    }

    //update list information
    list->nodeCount += (*toAppend)->nodeCount;

    free(*toAppend);
    *toAppend = NULL;
}

int totalCost(Node *); //forward declaration for the next function

/**
 * DESCRIPTION:
 *    This is a special node insertion function used for A*. Unlike
`insertNode()`,
 *    this inserts the node in order based on the sum of the node's
heuristic and
 *    path cost values. Node with the lowest value is sitted to the tail of
the queue.
 * PARAMETERS:
 *    toAppend - the list to be appended
 *    list     - the list to append `toAppend` into
**/
void pushListInOrder(NodeList **toAppend, NodeList *list) {
    //if either of the list is NULL, the head of the list to be appended is
NULL,
    //or the list points to the same starting node
    if(!*toAppend || !list || !(*toAppend)->head || (*toAppend)->head ==
list->head) {
        return;
    }

    //if the list to append to has currently no element
    if(!list->nodeCount) {
        pushNode(popNode(toAppend), &list);
    }

    ListNode *toAppendNode; //list node to place in `list`
    ListNode *listNode;     //for traversing each node in `list`
    Node *node;

     while((toAppendNode = (*toAppend)->head)) {
        listNode = list->head;

        while(listNode && totalCost(toAppendNode->currNode) <
totalCost(listNode->currNode)) {
            listNode = listNode->nextNode;
        }

        ListNode *temp = toAppendNode->nextNode;
```

```
        //if the head node of `toAppend` is to be inserted after the
current tail of `list`
        if(!listNode) {
            list->tail->nextNode = toAppendNode;
            toAppendNode->prevNode = list->tail;
            toAppendNode->nextNode = NULL;
            list->tail = toAppendNode;
        }
        else {
            //if if the head node of `toAppend` is to be inserted somewhere
in `list` except before its head
            if(listNode->prevNode) {
                toAppendNode->prevNode = listNode->prevNode;
                toAppendNode->nextNode = listNode;
                listNode->prevNode->nextNode = toAppendNode;
                listNode->prevNode = toAppendNode;
            }
            //if the head node of `toAppend` is to be inserted before the
head of `list`
            else {
                toAppendNode->nextNode = list->head;
                toAppendNode->prevNode = NULL;
                list->head->prevNode = toAppendNode;
                list->head = toAppendNode;
            }
        }

        (*toAppend)->head = temp;
        --(*toAppend)->nodeCount;
        ++list->nodeCount;
    }

    free(*toAppend);
    *toAppend = NULL;
}

//////////////////////////////
//////////////////////

/////////////////////
//////////////////////////
extern unsigned int nodesGenerated; //declared from main.c

/**
 * DESCRIPTION: Defines the node structure used to create a search tree
**/
typedef struct Node Node;
struct Node {
    unsigned int depth; //depth of the node from the root. For A* search,
                        //this will also represent the node's path cost
    unsigned int hCost; //heuristic cost of the node
    State *state;       //state designated to a node
    Node *parent;       //parent node
    NodeList *children; //list of child nodes
};

/**
 * DESCRIPTION:
 *    This function creates a node, initializes it with the
 *    following parameters, and sets its children to NULL.
 * PARAMETERS:
```

```c
 *    d - depth of the node
 *    h - heuristic value of the node
 *    s - state assignated to the node
 *    p - parent node
 * RETURN:
 *    Returns a `Node` pointer to the dynamically allocated node,
 *    or NULL on failure.
**/
Node* createNode(unsigned int d, unsigned int h, State *s, Node *p) {
    Node *newNode = malloc(sizeof(Node));
    if(newNode) {
        newNode->depth = d;
        newNode->hCost = h;
        newNode->state = s;
        newNode->parent = p;
        newNode->children = NULL;
        ++nodesGenerated; //update counter
    }
    return newNode;
}

/**
 * DESCRIPTION:
 *    This function is used to deallocate all nodes in a tree, including
 *    the root node.
 * PARAMETER:
 *    node - the root node of the tree to deallocate
**/
void destroyTree(Node *node) {
    if(node->children == NULL) {
        free(node->state);
        free(node);
        return;
    }

    ListNode *listNode = node->children->head;
    ListNode *nextNode;

    while(listNode) {
        nextNode = listNode->nextNode;
        destroyTree(listNode->currNode);
        listNode = nextNode;
    }

    //free(node->state);
    free(node->children);
    free(node);
}

/**
 * DESCRIPTION:
 *    This function 'expands' the node, links it to its children, and
updates the
 *    expansion counter.
 * PARAMETER:
 *    parent    - the node to expand and search children for
 *    goalState - pointer to the goal state where heuristic values of each
child will
 *                be based on
 * RETURN:
 *    Returns a pointer to `NodeList` on success, NULL on failure.
```

```
**/
NodeList* getChildren(Node *parent, State *goalState) {
    NodeList *childrenPtr = NULL;
    State *testState = NULL;
    Node *child = NULL;

    //attempt to create states for each moves, and add to the list of
children if true
    if(parent->state->action != DOWN && (testState = createState(parent-
>state, UP))) {
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != UP && (testState = createState(parent-
>state, DOWN))) {
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != RIGHT && (testState = createState(parent-
>state, LEFT))) {
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != LEFT && (testState = createState(parent-
>state, RIGHT))) {
        child = createNode(parent->depth + 1, manhattanDist(testState,
goalState), testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }

    return childrenPtr;
}

/**
 * DESCRIPTION:
 *    This simply evaluates the node's total cost, i.e. path cost +
heuristic value.
 *    Created to abstract the proccess and reduce code cluttering. Note
that
 *    a node's path cost in a tree depends purely on the node's depth, so
the node's
 *    depth will be representing the path cost (to save space).
 * PARAMETER:
      node - the node to get its total cost
 * RETURN:
 *    Retuns the integer sum of the node's path cost and heuristic value
**/
int totalCost(Node * const node) {
    return node->depth + node->hCost;
}
/////////////////////////

//external variables declared from main.c
extern unsigned int nodesExpanded;
```

```
extern unsigned int nodesGenerated;
extern unsigned int solutionLength;
extern double runtime;

/**
 * DESCRIPTION: This displays the '8-Puzzle Solver' ASCII art to the screen
**/
void welcomeUser(void) {
    //style from:
http://patorjk.com/software/taag/#p=display&f=Standard&t=8-Puzzle%20Solver
    printf("\n\
   ___        ____                _        ____         _\n\
  ( _ )      |  _ \\ _   _ _______| | ___   / ___|  ___ | |_   _____ _ __\n\
  / _ \\ _____| |_) | | | |_  /_  / |/ _ \\ \\___ \\ / _ \\| \\ \\ / / _ \\
'__|\n\
 | (_) |_____|  __/| |_| |/ / / /| |  __/  ___) | (_) | |\\ V /  __/ |\n\
  \\___/      |_|    \\__,_/___/___|_|\\___| |____/ \\___/|_| \\_/
\\___|_|\n");
}

/**
 * DESCRIPTION: This displays the input instructions for the user to read
**/
void printInstructions(void) {
    printf(
        "---------------------------------------------------------------------------
------\n"
        "Instructions:\n"
        "    Enter the initial and goal state of the 8-puzzle board.
Input\n"
        "    either integers 0-8, 0 representing the space character, to
assign\n"
        "    symbols toeach board[row][col].\n"
        "---------------------------------------------------------------------------
------\n"
    );
}

/**
 * DESCRIPTION:
 *    This function fills `state` with non-repeating numbers from 0 to 9
**/
void inputState(State * const state) {
    state->action = NOT_APPLICABLE;
    char row, col;
    int symbol;

    // flags for input validation
    char isNumUsed[9] = { 0 };

    for(row = 0; row < 3; ++row) {
        for(col = 0; col < 3; ++col) {
            printf("    board[%i][%i]: ", row, col);

            // to prevent scanning newline from the input stream
            scanf("%i", &symbol);

            // check if input is a blank character or is a number greater
than 0 and less than 9
            if(symbol >= 0 && symbol < 9) {
                // check if input is repeated
```

```
                if(!isNumUsed[symbol]) {
                    state->board[row][col] = symbol + '0';
                    isNumUsed[symbol] = 1;
                }
                else {
                    printf("    ERROR: Number %c is already used. Try again
with different input.\n", symbol);
                    --col;
                }
            }
            else {
                printf("    ERROR: Invalid input. Enter a number from 0 to
8.\n");
                --col;
            }
        }
    }
    printf("\n");
}

/**
 * DESCRIPTION: This displays contents of `board` to the standard output
**/
void printBoard(char const board[][3]) {
    char row, col;

    for(row = 0; row < 3; ++row) {
        printf("+---+---+---+\n");
        for(col = 0; col < 3; ++col) {
            printf("| %c ", board[row][col]);
        }
        printf("|\n");
    }
    printf("+---+---+---+\n");
}

/**
 * DESCRIPTION:
 *    This function interprets numerical instructions of the move to make,
 *    to it's verbal counterpart to be displayed to the screen.
 * PARAMETER:
 *    solution - the solution path consisting a list of nodes from the root
 *               to the goal
**/
void printSolution(struct SolutionPath *path) {
   //check if solution exists
    if(!path) {
        printf("No solution found.\n");
        return;
    }

   //if the initial state is already the goal state
   if(!path->next) {
      printf("No moves needed. The initial state is already the goal
state.\n");
      return;
   }

    printf("SOLUTION: (Relative to the space character)\n");

    //will use hash map to speed up the proccess a bit
```

```c
    char *move[4] = { "UP", "DOWN", "LEFT", "RIGHT" };
    int counter = 1;

    //will be skipping the first node since it represents the initial state
with no action
    for(path = path->next; path; path = path->next, ++counter) {
        printf("%i. Move %s\n", counter, move[path->action]);
    }

    printf(
        "DETAILS:\n"
        " - Solution length : %i\n"
        " - Nodes expanded  : %i\n"
        " - Nodes generated : %i\n"
        " - Runtime         : %g milliseconds\n"
        " - Memory used     : %i bytes\n", //only counting allocated
`Node`s
        solutionLength, nodesExpanded, nodesGenerated, runtime,
nodesGenerated * sizeof(Node));
}


///////////////////

///////////////////////

///////////////////////////

////////////////////////



unsigned int nodesExpanded;  //number of expanded nodes
unsigned int nodesGenerated; //number of generated nodes
unsigned int solutionLength; //number of moves in solution
double runtime;              //elapsed time (in milliseconds)

SolutionPath* BFS_search(State *, State *);
//SolutionPath* AStar_search(State *, State *);

int main(void) {
    welcomeUser();           //display welcome message
    printInstructions();     //display instructions

    State initial;           //initial board state
    State goalState;         //goal board configuration

    //solution path of each search method
    SolutionPath *bfs;
    //SolutionPath *aStar;

    //input initial board state
    printf("INITIAL STATE:\n");
    inputState(&initial);

    //input the goal state
    printf("\nGOAL STATE:\n");
    inputState(&goalState);

    printf("INITIAL BOARD STATE:\n");
    printBoard(initial.board);
```

```c
    printf("GOAL BOARD STATE:\n");
    printBoard(goalState.board);

    //perform A* search
    /*aStar = AStar_search(&initial, &goalState);
    printf("\n-------------------------- USING A* ALGORITHM ---------------
-----------\n");
    printSolution(aStar);

    //reset the counters
    nodesExpanded = 0;
    nodesGenerated = 0;
    solutionLength = 0;
    runtime = 0; */

    //perform breadth-first search
    bfs = BFS_search(&initial, &goalState);
    printf("\n-------------------------- USING BFS ALGORITHM ---------------
------------\n");
    printSolution(bfs);

    //free resources
    destroySolution(&bfs);
    //destroySolution(&aStar);

    return 0;
}

/**
 * DESCRIPTION:
 *    Our breadth-first search implemetation.
 * PARAMETERS:
 *    initial - address to the initial state
 *    goal    - address to the goal state
 * RETURN:
 *    Returns the solution in a linked list; NULL if the solution is not
found.
**/
SolutionPath* BFS_search(State *initial, State *goal) {
    NodeList *queue = NULL;
    NodeList *children = NULL;
    Node *node = NULL;

    //start timer
    clock_t start = clock();

    //initialize the queue with the root node of the search tree
    pushNode(createNode(0, manhattanDist(initial, goal), initial, NULL),
&queue);
    Node *root = queue->head->currNode; //for deallocating the generated
tree

    //while there is a node in the queue to expand
    while(queue->nodeCount > 0) {
        //pop the last node (tail) of the queue
        node = popNode(&queue);

        //if the state of the node is the goal state
        if(statesMatch(node->state, goal))
            break;
```

```c
        //else, expand the node and update the expanded-nodes counter
        children = getChildren(node, goal);
        ++nodesExpanded;

        //add the node's children to the queue
        pushList(&children, queue);
    }

    //determine the time elapsed
    runtime = (double)(clock() - start) / CLOCKS_PER_SEC;

    //get solution path in order from the root, if it exists
    SolutionPath *pathHead = NULL;
    SolutionPath *newPathNode = NULL;

    while(node) {
        newPathNode = malloc(sizeof(SolutionPath));
        newPathNode->action = node->state->action;
        newPathNode->next = pathHead;
        pathHead = newPathNode;

        //update the solution length and move on the next node
        ++solutionLength;
        node = node->parent;
    }

    --solutionLength; //uncount the root node

    //deallocate the generated tree
    destroyTree(root);

    return pathHead;
}

/**
 * DESCRIPTION:
 *    Our A* implemetation.
 * PARAMETERS:
 *    initial - address to the initial state
 *    goal    - address to the goal state
 * RETURN:
 *    Returns the solution in a linked list; NULL if the solution is not
found.
**/



/*
1  2  3
8  0  4
7  6  5

1  3  4
8  6  2
7  0  5

0*/
```

For Branch and Bound

```
// Program to print path from root node to destination node
// for N*N -1 puzzle algorithm using Branch and Bound
// The solution assumes that instance of puzzle is solvable
#include <bits/stdc++.h>
using namespace std;
#define N 3

// state space tree nodes
struct Node
{
    // stores the parent node of the current node
    // helps in tracing path when the answer is found
    Node* parent;

    // stores matrix
    int mat[N][N];

    // stores blank tile coordinates
    int x, y;

    // stores the number of misplaced tiles
    int cost;

    // stores the number of moves so far
    int level;
};

// Function to print N x N matrix
int printMatrix(int mat[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

// Function to allocate a new node
Node* newNode(int mat[N][N], int x, int y, int newX,
              int newY, int level, Node* parent)
{
    Node* node = new Node;

    // set pointer for path to root
    node->parent = parent;

    // copy data from parent node to current node
    memcpy(node->mat, mat, sizeof node->mat);

    // move tile by 1 position
    swap(node->mat[x][y], node->mat[newX][newY]);

    // set number of misplaced tiles
    node->cost = INT_MAX;

    // set number of moves so far
    node->level = level;
```

```
    // update new blank tile cordinates
    node->x = newX;
    node->y = newY;

    return node;
}

// bottom, left, top, right
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };

// Function to calculate the number of misplaced tiles
// ie. number of non-blank tiles not in their goal position
int calculateCost(int initial[N][N], int final[N][N])
{
    int count = 0;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (initial[i][j] && initial[i][j] != final[i][j])
                count++;
    return count;
}

// Function to check if (x, y) is a valid matrix cordinate
int isSafe(int x, int y)
{
    return (x >= 0 && x < N && y >= 0 && y < N);
}

// print path from root node to destination node
void printPath(Node* root)
{
    if (root == NULL)
        return;
    printPath(root->parent);
    printMatrix(root->mat);

    printf("\n");
}

// Comparison object to be used to order the heap
struct comp
{
    bool operator()(const Node* lhs, const Node* rhs) const
    {
        return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);
    }
};

// Function to solve N*N - 1 puzzle algorithm using
// Branch and Bound. x and y are blank tile coordinates
// in initial state
void solve(int initial[N][N], int x, int y,
           int final[N][N])
{
    // Create a priority queue to store live nodes of
    // search tree;
    priority_queue<Node*, std::vector<Node*>, comp> pq;

    // create a root node and calculate its cost
    Node* root = newNode(initial, x, y, x, y, 0, NULL);
```

```
    root->cost = calculateCost(initial, final);

    // Add root to list of live nodes;
    pq.push(root);

    // Finds a live node with least cost,
    // add its childrens to list of live nodes and
    // finally deletes it from the list.
    while (!pq.empty())
    {
        // Find a live node with least estimated cost
        Node* min = pq.top();

        // The found node is deleted from the list of
        // live nodes
        pq.pop();

        // if min is an answer node
        if (min->cost == 0)
        {
            // print the path from root to destination;
            printPath(min);
            return;
        }

        // do for each child of min
        // max 4 children for a node
        for (int i = 0; i < 4; i++)
        {
            if (isSafe(min->x + row[i], min->y + col[i]))
            {
                // create a child node and calculate
                // its cost
                Node* child = newNode(min->mat, min->x,
                              min->y, min->x + row[i],
                              min->y + col[i],
                              min->level + 1, min);
                child->cost = calculateCost(child->mat, final);

                // Add child to list of live nodes
                pq.push(child);
            }
        }
    }
}

// Driver code
int main()
{
    // Initial configuration
    // Value 0 is used for empty space

    int initial[N][N];
    printf("Enter initial matrix");
    for(int i=0;i<N;i++)
    {
        for(int j=0;j<N;j++)
        {
            cin>>initial[i][j];
        }
    }
```

```
    printf("\n");
    // Solvable Final configuration
    // Value 0 is used for empty space

    int final[N][N];
    printf("Enter final matrix");
    for(int i=0;i<N;i++)
    {
        for(int j=0;j<N;j++)
        {
            cin>>final[i][j];
        }
    }
    printf("\n");

    // Blank tile coordinates in initial
    // configuration
    int x = 1, y = 2;

    solve(initial, x, y, final);

    return 0;
}
```