भारतीय सूचना प्रौद्योगिकी संस्थान भागलपुर
Indian Institute of Information Technology
Bhagalpur

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY BHAGALPUR

AI - PROJECT

CS303 – Artificial Intelligence Lab

P8: 8 Puzzle Problem using IDA*

Submission date: 20/09/19

Submitted By:

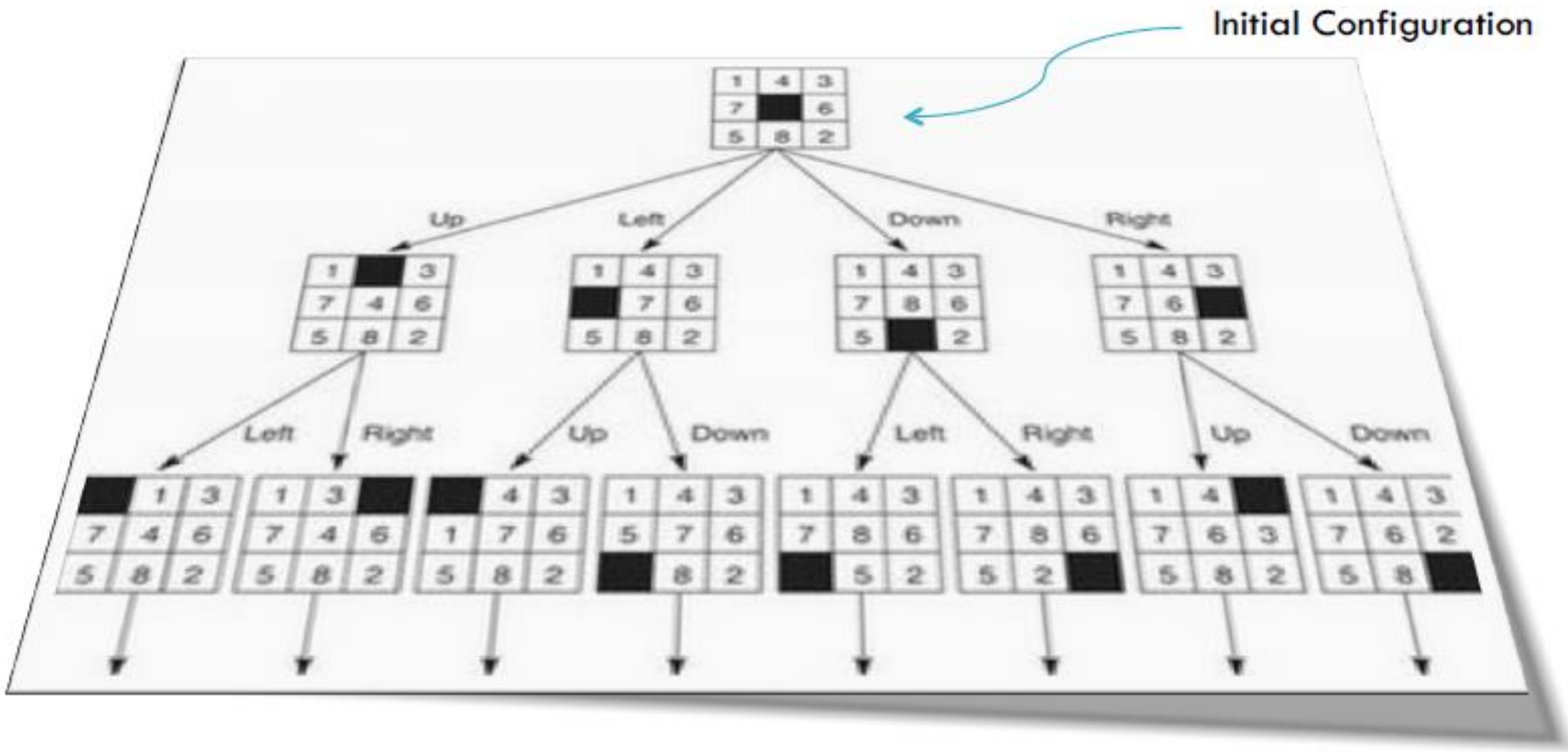
Sunny Kumar (170101051)
Mithlesh Kumar Basfor (170101026)
Susheela (170102051)
Anil Kumar(170102005)
Anand Kumar(170101010)

Submitted To:

Dr. Rupam Bhattacharyya

Problem Description

- The 8 puzzle is a simple game which consists of eight sliding tiles, labeled with pieces of a image, placed in a 3x3 squared board of nine cells.
- One of the cells is always empty, and any adjacent (horizontally and vertically) tile can be moved into the empty cell.
- The objective of the game is to start from an initial configuration and end up in a configuration which the tiles are placed in ascending number order.

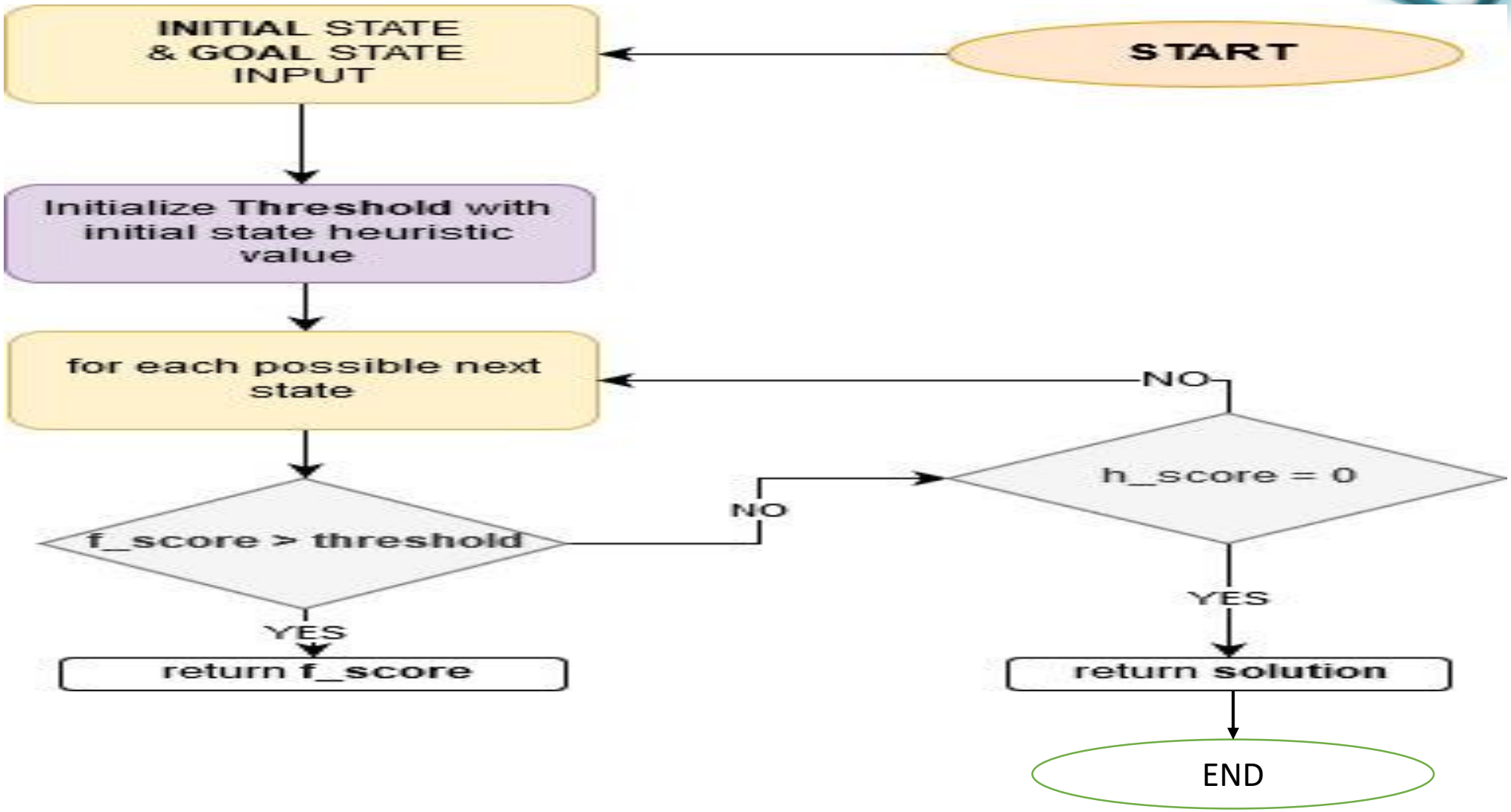

Initial Configuration
Up
Left
Down
Right
Left
Right
Up
Down
Left
Right
Up
Down

IDA* algorithm

- Many 8-puzzles cannot be solved efficiently with the A* algorithm, since it generates too many new states and consumes a lot of memory maintaining these lists. To solve such puzzles, Iterative-Deepening-A* (IDA*) can be used. Like the A* algorithm, it finds optimal solutions when paired with an admissible heuristic but is much more efficient with respect to space. IDA* is described as follows:
  - Set *threshold* equal to the heuristic evaluation of the initial state.
  - Conduct a depth-first search, pruning a branch when the cost of the latest node exceeds *threshold*. If a solution is found during the search, return it.
  - If no solution is found during the current iteration, increment *threshold* by the minimum amount it was exceeded, and go back to the previous step.

Flow Chart


1
2
3
4
5
6
7
8


START
INITIAL STATE & GOAL STATE INPUT
Initialize Threshold with initial state heuristic value
for each possible next state
f_score > threshold
YES
return f_score
NO
h_score = 0
NO
YES
return solution
END

CODES :


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>

#define NUM_OF_POSSIBLE_MOVES 4 /* Up, down, left, right */
#define MAX_SOLUTION_LENGTH 1000
#define MAX_F_VALUE 100

int print_array(const int *arr, int N);

int** new_board(int N);

int print_board(const int** board, int N);

int free_board(int** board, int N);

int valid_moves(int N, int* result, int x_row, int x_col);

int print_solution(const int **start, int N, const char *desc);

int search(int **board, const int **goal, int N, int depth, int threshold, int *found, char *desc,
    char **solution, int *current_rows, int *current_cols, const int *goal_rows, const int * goal_cols, int N_squar
    int x_row, int x_col, int h_score);

int run(int **start, const int **goal, int N);

```

```c
int main()
{

    FILE *fid;
    int N, i, j;
    int **start, **goal;

    /* Read start position */
    fid = fopen("start.txt", "rt");
     N=3;
    start = new_board(N);
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            fscanf(fid, "%d ", &start[i][j]);
        }
    }
    fclose(fid);

    /* Read goal position */
    fid = fopen("goal.txt", "rt");
    goal = new_board(N);
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            fscanf(fid, "%d ", &goal[i][j]);
        }
    }
    fclose(fid);


    run(start, (const int **) goal, N);

    free_board(start, N);
    free_board(goal, N);
    return 0;
}

```

```c
int print_array(const int *arr, int N){
    int i;
    for (i = 0; i < N; i++){
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
int** new_board(int N){
   int **board, i;
   board = (int **) malloc(sizeof(int *) * N);
   for (i = 0; i < N; i++){
      board[i] = (int *) malloc(sizeof(int) * N);
   }
   return board;
}

int print_board(const int** board, int N){
    int i, j;
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            if (board[i][j] == 0) {
                printf("%2c ",'x');
            } else {
                printf("%2d ", board[i][j]);
            }
        }
        printf("\n");
    }
    return 0;
}
```

```
int free_board(int** board, int N){
    int i;
    for (i = 0; i < N; i++){
        free(board[i]);
    }
    free(board);
    return 0;
}

int print_solution(const int **start, int N, const char *desc){
    int **board, i, j, temp, x_row, x_col, new_x_row, new_x_col;
    char move;
    board = new_board(N);
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            board[i][j] = start[i][j];
            if (board[i][j] == 0) {
                x_row = i;
                x_col = j;
            }
        }
    }
    printf("Moves to get to the solution: %s \n", desc);

    printf("The solution: \n");
    i = 1;int count=0;
    while (1) {
        print_board((const int **) board, N);
        printf("\n\n");
        move = desc[i];
        if (move == '\0'){
```

```
122     while (1) {
123         print_board((const int **) board, N);
124         printf("\n\n");
125         move = desc[i];
126         if (move == '\0'){
127             printf("NUMBER OF STEPS REQUIRED TO REACH GOAL STATE: %d\n",count);
128             printf("\nGOT IT.....:):)\n");
129             for (i = 0; i < N; i++){
130                 free(board[i]);
131             }
132             free(board);
133             return 0;
134         }
135         count++;
136         switch (move){
137             case 'u':
138                 /* Up */
139                 new_x_row = x_row - 1;
140                 new_x_col = x_col;
141                 break;
142             case 'd':
143                 /* Down */
144                 new_x_row = x_row + 1;
145                 new_x_col = x_col;
146                 break;
147             case 'l':
148                 /* Left */
149                 new_x_row = x_row;
150                 new_x_col = x_col - 1;
151                 break;
152             case 'r':
```

```
168 int comp( int *current_rows, int *current_cols, const int *goal_rows, const int * goal_cols,int N_squared){
169     int count=0;
170     for(int i=1;i<N_squared;i++){
171         if(current_rows[i]==goal_rows[i] && current_cols[i]==goal_cols[i])
172             continue;
173         else
174             count++;
175     }
176     return count;
177 }
178
179 int search(int **board, const int **goal, int N, int depth, int threshold, int *found, char *desc,
180     char **solution, int *current_rows, int *current_cols, const int *goal_rows, const int * goal_cols, int N_squared,
181     int x_row, int x_col, int h_score){
182
183     int f_score;
184     int min, temp;
185     int solution_length;
186     int i;
187     char move, go_back_move, last_move_by_current;
188     int old_x_row, old_x_col, new_x_row, new_x_col;
189     int new_h_score;
190     int temp_1, goal_row_temp, goal_col_temp;
191     int N_minus_one = N - 1;
192
193     /* printf("in search \n");
194     */
195
196     f_score = depth + h_score;
197     if (f_score > threshold) {
198         return f_score;
199     }
```

```
196     f_score = depth + h_score;
197     if (f_score > threshold) {
198         return f_score;
199     }
200     if (h_score == 0){
201         *found = 1;
202         solution_length = 0;
203         while (desc[solution_length] != '\0'){
204             solution_length++;
205         }
206         *solution = (char *) malloc(sizeof(char) * solution_length);
207         for (i = 0; i < solution_length; i++){
208             (*solution)[i] = desc[i];
209         }
210         (*solution)[solution_length] = '\0';
211         return f_score;
212     }
213     min = INT_MAX;
214     last_move_by_current = desc[depth]; /* desc always starts with 'B' so this is okay */
215     old_x_row = x_row;
216     old_x_col = x_col;
217     for (i = 0; i < NUM_OF_POSSIBLE_MOVES; i++){
218         switch (i) {
219             case 0:
220                 move = 'u';
221                 go_back_move = 'd';
222                 new_x_row = x_row - 1;
223                 new_x_col = x_col;
224                 break;
225             case 1:
226                 move = 'd';
```

```
235                 new_x_col = x_col - 1;
236                 break;
237             case 3:
238                 move = 'r';
239                 go_back_move = 'l';
240                 new_x_row = x_row;
241                 new_x_col = x_col + 1;
242                 break;
243         }
244         if (move == 'u'){
245             if (x_row == 0) {
246                 continue;
247             }
248         }
249         if (move == 'd'){
250             if (x_row == N_minus_one){
251                 continue;
252             }
253         }
254         if (move == 'l'){
255             if (x_col == 0){
256                 continue;
257             }
258         }
259         if (move == 'r'){
260             if (x_col == N_minus_one){
261                 continue;
262             }
263         }
264         if (last_move_by_current == go_back_move){
265             /* No need to consider going back to the previous state */
266             continue;
```

```
268
269         /* Update h_score */
270         new_h_score = h_score;
271         temp_1 = board[new_x_row][new_x_col];
272         goal_row_temp = goal_rows[temp_1];
273         goal_col_temp = goal_cols[temp_1];
274         switch (go_back_move){
275             case 'u':
276                 if (goal_row_temp < current_rows[temp_1]){
277                     new_h_score--;
278                 } else {
279                     new_h_score++;
280                 }
281                 break;
282             case 'd':
283                 if (goal_row_temp > current_rows[temp_1]){
284                     new_h_score--;
285                 } else {
286                     new_h_score++;
287                 }
288                 break;
289             case 'l':
290                 if (goal_col_temp < current_cols[temp_1]){
291                     new_h_score--;
292                 } else {
293                     new_h_score++;
294                 }
295                 break;
296             case 'r':
297                 if (goal_col_temp > current_cols[temp_1]){
298                     new_h_score--;
299                 } else {
300                     new_h_score++;
```

```
            }
            break;
        }

        /* Move */
        current_rows[0] = new_x_row;
        current_cols[0] = new_x_col;
        current_rows[temp_1] = old_x_row;
        current_cols[temp_1] = old_x_col;
        board[old_x_row][old_x_col] = temp_1;
        board[new_x_row][new_x_col] = 0;
        desc[depth+1] = move;
        desc[depth+2] = '\0';

        /* Search further down the game tree */
        temp = search(board, goal, N, depth+1, threshold, found, desc, solution, current_rows, current_cols,
            goal_rows, goal_cols, N_squared, new_x_row, new_x_col, new_h_score);

        /* Move back */
        board[old_x_row][old_x_col] = 0;
        board[new_x_row][new_x_col] = temp_1;
        desc[depth+1] = '\0';
        current_rows[0] = old_x_row;
        current_cols[0] = old_x_col;
        current_rows[temp_1] = new_x_row;
        current_cols[temp_1] = new_x_col;

        if (*found == 1){
            return temp;
        }

```

```
340
341 int run(int **start, const int **goal, int N){
342     int threshold;
343     int found = 0;
344     int *current_cols, *current_rows, *goal_cols, *goal_rows; /* Position of each element */
345     int N_squared;
346     int temp;
347     char *desc = (char *) malloc(sizeof(char) * MAX_SOLUTION_LENGTH);
348     char *solution = NULL;
349     int i,j;
350     int x_row, x_col;
351     int h_score;
352
353     /* Precompute positions for goal */
354     N_squared = N*N;
355     current_cols = (int *) malloc(sizeof(int) * N_squared);
356     current_rows = (int *) malloc(sizeof(int) * N_squared);
357     goal_cols = (int *) malloc(sizeof(int) * N_squared);
358     goal_rows = (int *) malloc(sizeof(int) * N_squared);
359     for (i = 0; i < N; i++){
360         for (j = 0; j < N; j++){
361             goal_rows[goal[i][j]] = i;
362             goal_cols[goal[i][j]] = j;
363         }
364     }
365     for (i = 0; i < N; i++){
366         for (j = 0; j < N; j++){
367             current_rows[start[i][j]] = i;
368             current_cols[start[i][j]] = j;
369         }
370     }
```

```
372     /* Find 'x' position */
373     for (i = 0; i < N; i++){
374         for (j = 0; j < N; j++){
375             if (start[i][j] == 0) {
376                 x_row = i;
377                 x_col = j;
378             }
379         }
380     }
381
382     desc[0] = 'B';
383     desc[1] = '\0';
384     h_score = 0;
385     h_score=comp(current_rows, current_cols, (const int *)goal_rows, (const int *) goal_cols, N_squared);
386     threshold = h_score;
387     while (1){
388         temp = search(start, goal, N, 0, threshold, &found, desc, &solution,
389             current_rows, current_cols, (const int *)goal_rows, (const int *) goal_cols, N_squared,
390             x_row, x_col, h_score);
391         if (found == 1){
392             printf("FOUND SOLUTION!\n");
393             print_solution((const int **) start, N, (const char *) solution);
394             goto CLEANUP;
395         }
396         if (temp > MAX_F_VALUE){
397             /* Threshold larger than maximum possible f value */
398             printf("MAXIMUM F VALUE REACHED! TERMINATING! \n");
399             goto CLEANUP;
400         }
401         threshold = temp;
402     }
```

END of CODE with releasing space used


```
401         threshold = temp;
402     }
403     CLEANUP: {
404         free(solution);
405         free(current_cols);
406         free(current_rows);
407         free(goal_cols);
408         free(goal_rows);
409         free(desc);
410         return 0;
411     }
```

FULL CODE:

https://drive.google.com/open?id=1HmJ7Xm8Jem2jorvRoIgoymDE7-FxDANE

SOME SNAPSHOTS:

Open

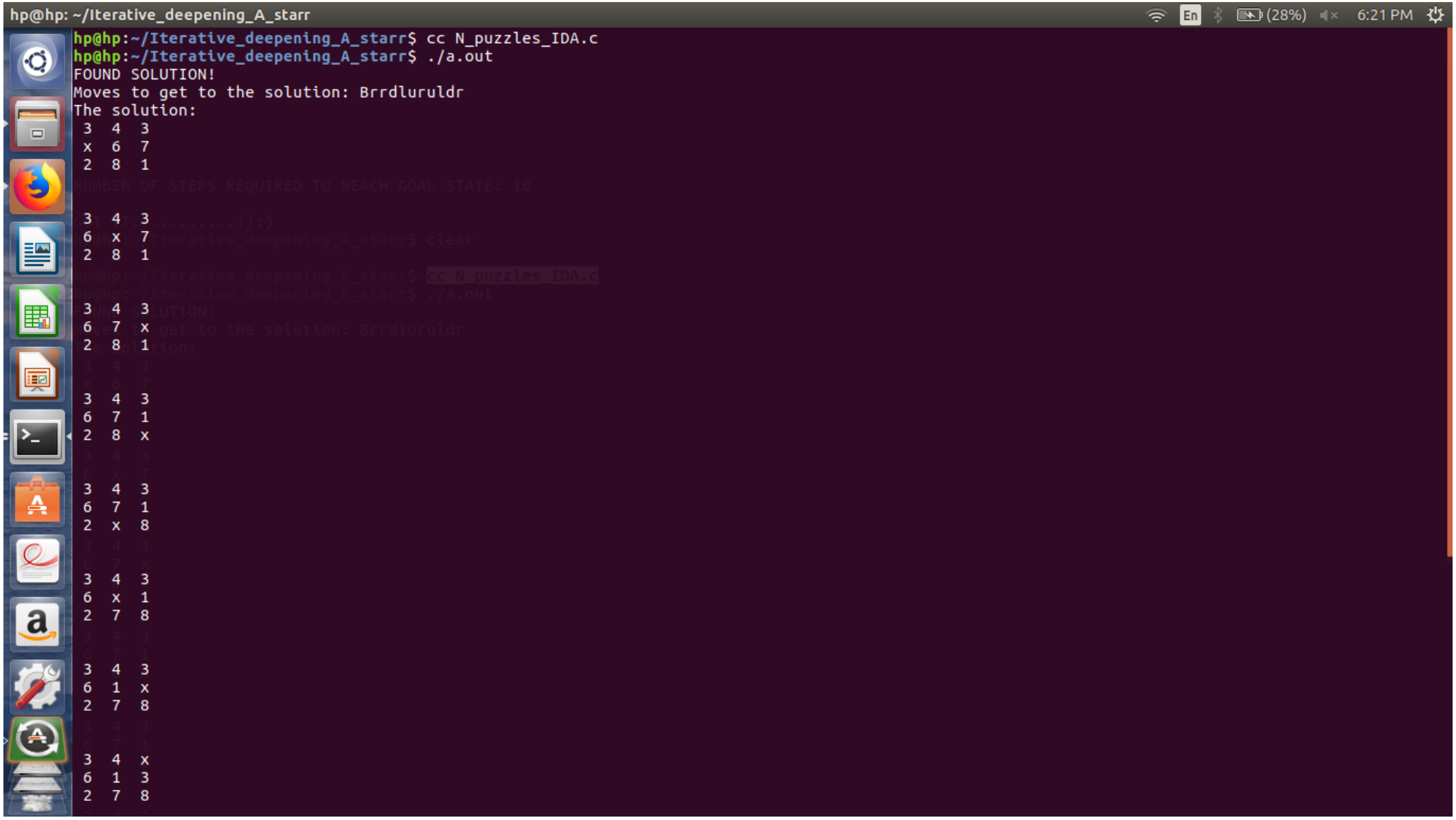
```
4  3  0
6  7  2
8  1  5
```

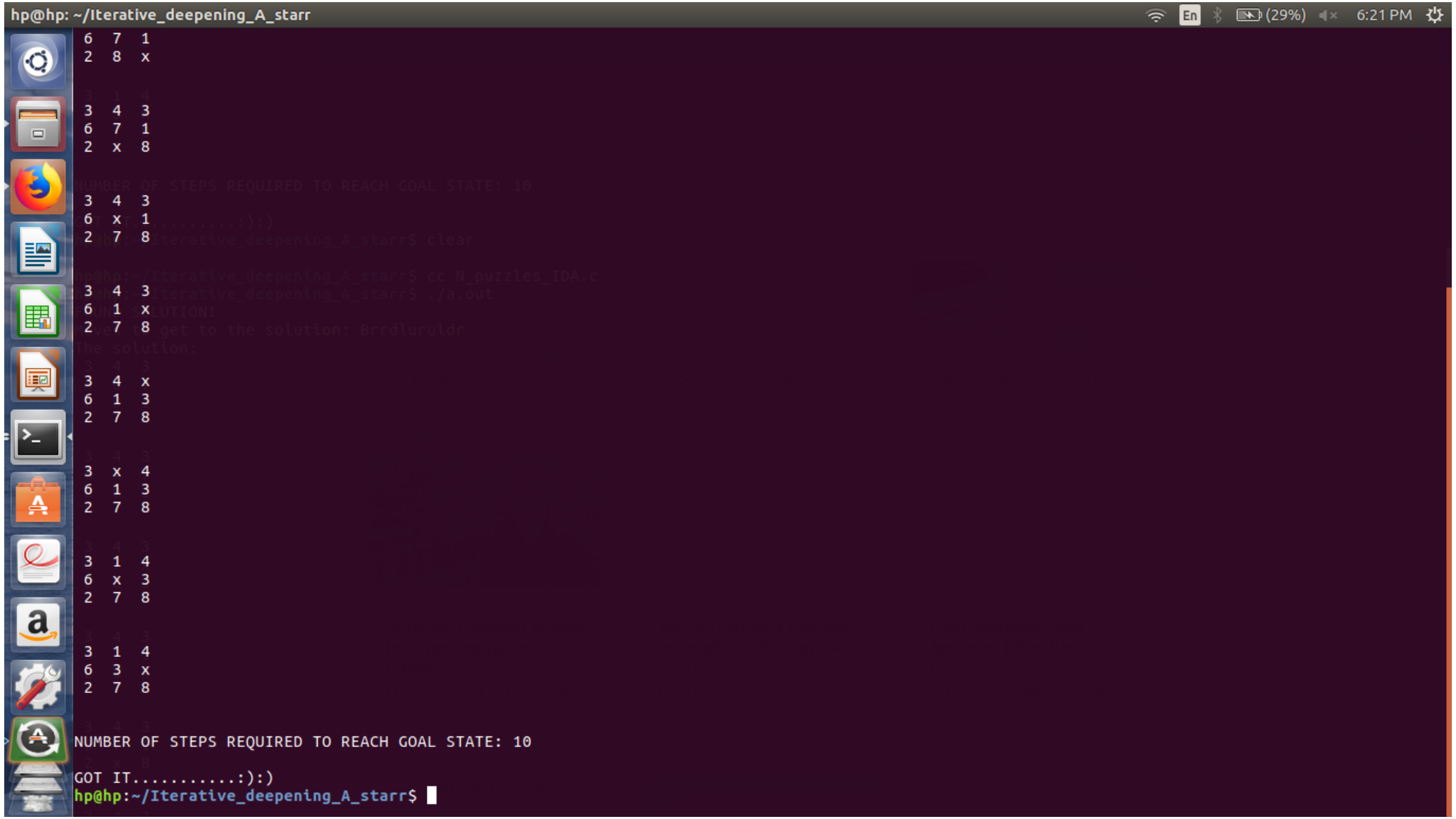
start.txt
file

Open

```
0  1  2
3  4  5
6  7  8
```

goal.txt
file


hp@hp: ~/Iterative_deepening_A_starr
En
(28%)
6:21 PM
hp@hp:~/Iterative_deepening_A_starr$ cc N_puzzles_IDA.c
hp@hp:~/Iterative_deepening_A_starr$ ./a.out
FOUND SOLUTION!
Moves to get to the solution: Brrdluruldr
The solution:
3 4 3
x 6 7
2 8 1
3 4 3
6 x 7
2 8 1
3 4 3
6 7 x
2 8 1
3 4 3
6 7 1
2 8 x
3 4 3
6 7 1
2 x 8
3 4 3
6 x 1
2 7 8
3 4 3
6 1 x
2 7 8
3 4 x
6 1 3
2 7 8


hp@hp: ~/Iterative_deepening_A_starr
En
(29%)
6:21 PM
6 7 1
2 8 x
3 4 3
6 7 1
2 x 8
3 4 3
6 x 1
2 7 8
3 4 3
6 1 x
2 7 8
3 4 x
6 1 3
2 7 8
3 x 4
6 1 3
2 7 8
3 1 4
6 x 3
2 7 8
3 1 4
6 3 x
2 7 8
NUMBER OF STEPS REQUIRED TO REACH GOAL STATE: 10
GOT IT.....:):)
hp@hp:~/Iterative_deepening_A_starr$

CONCLUSION CUM OBSERVATION

There are various searching algorithm for solving 8 puzzle but IDA* algorithm fetches the optimal solution among most of them.

The heuristic we used –" the number of misplaced tiles (excluding 0 tile) gives the optimal solution as compare to Manhattan heuristic function.

It also eliminates the memory constraints of A* algorithm without sacrificing optimality.