8-Puzzle Solver Documentation and Report

Francis Anjelo M. Andes and Miles B. Artiaga

Department of Computer Science and Information Technology, College of Science,

Bicol University

CSElec 1: Artificial Intelligence

Arlene A. Satuito

December 10, 2020

8-Puzzle Solver Documentation and Report

The sliding puzzle game is a classic game wherein the player is challenged to slide pieces or tiles along certain routes, usually on a board, to establish a certain end configuration. The sliding puzzle has various implementations, which can differ on what the tiles contain (e.g., simple shapes, patterns, sections of a larger picture, or numbers) or the size of the board or grid.

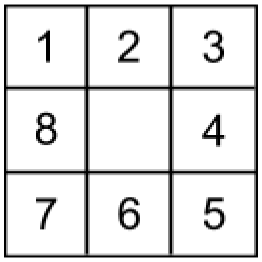
The 8-puzzle is a variation of the sliding puzzle game which consists of a 3 x 3 grid (containing 9 squares), with one of the squares being empty. Similar to the sliding puzzle, the objective of this game is to slide the tiles until it reaches the end-configuration. The end configuration or *goal state* implemented in the program is shown in Figure 1.

Purpose of the Program

The purpose of this program is to find the optimal solution to a given initial *state* or *board configuration* of the 8-puzzle. The *solution* to this game is defined as a series of moves or actions, described as directions, e.g., up or left. that are needed to perform to reach the goal state of the 8-puzzle. These moves or actions refer as to how the blank tile should be moved in the grid—i.e., if the move is down, the blank tile should be swapped with the tile below it—representing the act of sliding of the tiles in the sliding puzzle.

Figure 1

Goal State



Overall Method

The list of the general tasks in the program is:

1. Show the instructions to the user and let them choose their desired initial board configuration.
2. Check if the initial board configuration is solvable. If it is not solvable, alert the user and exit the program.
3. If the initial board configuration is solvable, proceed to ask the user which heuristic measure they want the program to use.
4. Perform the A* Search algorithm and find the solution to the puzzle.
5. Print the solution and the details about the process.

In choosing the initial board configuration, the user can choose from a set of predetermined initial states of varying difficulties, which is shown in Figure 2, or choose to input their desired initial board configuration.

Two heuristics can also be used to perform the A* Search algorithm, being the Hamming Distance and the Manhattan Distance.

Figure 2

Predetermined Initial Board Configurations

1	3	4
8	6	2
7		5

Easy

2	8	1
	4	3
7	6	5

Medium

2	8	1
4	6	3
7		5

Hard

5	6	7
4		8
3	2	1

Worst

Source Code Documentation

Source Files

8-puzzle-solver.c

This C file contains the main program of the 8-Puzzle Solver including the function for A* Search algorithm.

state.h

Included in this header file are the definition for the State data type as well as its operators or functions.

node.h

The data structure representing a game tree node is defined in this header file, along with its operators.

stack.h

This header file contains the definition for the data structure that represents the a node in a linked list which is implemented as a stack. Stack linked list operators, such as push and pop, are also defined in here.

position.h

The data structure representing a position of a tile in an 8-puzzle board is defined in this header file as well as its operator.

io.h

The functions regarding the input and output operations are contained here.

Data Structures and Operators

States

This custom data type represents an instance of the different possible configurations of the 3 x 3 board which is called a game state. This data type is just a two-dimensional array, with both arrays having a size of 3. In a valid game state, each element of the array should be populated with values from 0 to 8 with no value being repeated. The implementation of this custom data type is

```
typedef int State[BOARD_SIZE][BOARD_SIZE];
```

As shown in Table 1, the state data type has several operators. The fValue function takes a state and an integer as its parameter. The integer represents the heuristic measure to be used by the program, and depending on this, the function calls either the HammingDistance or ManhattanDistance functions. The return value of these functions is then returned by the fValue function to the callee.

The isIdentical function takes two states as inputs and compares whether these two states are the same. It returns true if it is, otherwise, it returns false.

The copyState function takes two states as inputs and copies the board configuration of the second state to the first state.

The parameter of the isSolvable function is a state. This function checks whether the board configuration of the state is solvable by checking the numbers of inversions. If the number of inversions are odd, then it is solvable. An integer value is returned by this function which represents a boolean value.

Moves

This enum data type holds the different actions that can be performed on a certain game state according to the game mechanics. In the case of the 8-puzzle game and any normal n-puzzle games the possible actions are up, down, left, and right.

```
typedef enum Move
{
    UP, DOWN, LEFT, RIGHT,  // values for each of the move
    NOT_APPLICABLE          // value for the initial state
}
Move;
```

Table 1

State Operators

Function	Parameters				Return Value		
	Name	Type	Value/ Reference	Description	Name	Type	Description
fValue	state	State	reference	the game state	fValue	int	the fValue of the state
	heuristic	int	value	heuristic being used			
HammingDistance	state	State	reference	the game state	counter	int	number of misplaced tiles
ManhattanDistance	state	State	reference	the game state	sum	int	sum of the distances of each tile from their goal position
isIdentical	state1	State	reference	first game state	<none>	int	boolean representation
	state2	State	reference	second game state			
copyState	destination	State	reference	destination game state	<none>	<none>	<none>
	source	State	reference	source game state			
isSolvable	state	State	reference	the game state	<none>	int	boolean representation

Nodes

This data structure represents the game tree node for when performing the A* Search and holds all the necessary information about a node, such as the state, the action that led to the state, its parent node, f-value and depth. Note that this data structure is not to be confused with StackNode. The implementation of this data structure is

```c
typedef struct Node
{
    State state;            // state (current board configuration)
    Move action;            // action that led to this state
    struct Node *parent;    // pointer to its parent node
    unsigned int depth;     // the depth of this node in the search tree
    unsigned int fValue;    // the f-value of this state
}
Node;
```

Shown in Table 2 are the operators on the node data structure. The createNode function is a function used to create a new node. With the parameters being the state, parent node, depth, f-value, and move, a new node is created with the values set equal to the input values. This newly created node is then returned to the callee.

The createSuccessorNode function takes a node, heuristic, and move as inputs and creates a new child node that is made after performing the move to the parent node. This newly created node, complete with all its value, is returned to the callee. However, if the move cannot be performed to the parent node, then there is no child node to be generated, therefore this function will return a NULL.

The parameters of the generateSuccesors function are a node and the heuristic. This function, as the name implies, generates all the child nodes that can be generated from the parent node by performing all the possible moves. The linked list of successors is the return value of this function.

Table 2

Node Operators

Function	Parameters				Return Value		
	Name	Type	Value/ Reference	Description	Name	Type	Description
createNode	state	State	reference	the game state	newNode	Node*	the new node created
	parent	Node	reference	pointer to its parent			
	depth	int	value	the depth of the node in the game tree			
	fValue	int	value	the fValue of the game state			
	action	Move	value	the move that led to this state			
createSuccessor Node	parent	Node	reference	the parent node	newNode	Node*	the new node created
	action	Move	value	move performed on the parent node			
	heuristic	int	value	heuristic being used			
generateSuccesors	parent	Node	reference	pointer to its parent	successors	StackNode*	linked list of the successor nodes
	heuristic	int	value	heuristic being used			

StackNodes

This data structure represents a node in a linked list, which contains the data that it holds and a pointer to the next node. The linked list in this program is implemented as a stack. The implementation of this data structure is

```
typedef struct StackNode
{
    Node *node;                     // pointer to a node that represents a state in the puzzle
    struct StackNode *next;         // pointer to the next node in the list
}
StackNode;
```

The different operators of the stack node data structure are shown in Table 3. The push function takes two inputs as its parameters—an address of the top element of a stack and a node. By calling this function, the node is pushed to the stop of the stack.

The pop function on the other hand, has one parameter being an address of the top element of a stack. This function pops or removes the top element of the stack and sets the second stack node as the new top element of the stack. The node data held by the popped stack node is returned to the callee.

The sortedInsert function is almost the same as the push function, however, the difference lies in how the node input is inserted into the stack. As opposed to inserting the node to the top of the stack, this function checks the f-value of the node and inserts it into the stack in such a way that the f-values of each node in the stack are in ascending order.

The associate function has an address of the top element of a stack and a node as its parameters. This function checks whether the state of the node is already in the stack and if it is, it compares the f-value of the node and the node that is already in the stack. If it finds that the f-value of the node in the stack is higher, that node is removed from the stack and the function returns a true value. Otherwise, it returns false.

Table 3

StackNode Operators

Function	Parameters				Return Value		
	Name	Type	Value/ Reference	Description	Name	Type	Description
push	top	StackNode	reference	the address of the first element in the stack	<none>	<none>	<none>
	node	Node	reference	new node to be inserted on top of the stack			
pop	top	StackNode	reference	the address of the first element in the stack	popped	Node*	the node popped from the stack
sortedInsert	stack	StackNode	reference	the address of the first element in the stack	<none>	<none>	<none>
	node	Node	reference	new node to be inserted to the stack			
associate	stack	StackNode	reference	the address of the first element in the stack	<none>	int	boolean representation
	node	Node	reference	node to be associated			
destroyStack	top	StackNode	reference	the address of the first element in the stack	<none>	<none>	<none>

The destroyStack function takes a stack as its input and it simply deallocates every node that the stack node contains as well as the stack node itself.

Positions

This data structure represents a position in the board of the 8-puzzle. This contains the row and column coordinates and implemented as

```
typedef struct Position
{
    int row;   // row position
    int col;   // column position
}
Position;
```

Table 4

Position Operator

Function	Parameters				Return Value		
	Name	Type	Value/ Reference	Description	Name	Type	Description
getPosition	state	State	reference	the game state/board configuration	position	Position	the position of the number in the board
	number	int	value	the number whose position is going to be identified			

The position data structure has a single operator, as shown in Table 4, which is getPosition. This function takes a state and a number as its parameters. The position of the number in the board is returned by this function.

Global Variables

The program has several global variables that are used to record the statistics of the process of the A* Search. These global variables are

- *nodesExpanded*, of type unsigned int, which records the numbers of nodes that were expanded;
- *nodesGenerated*, of type unsigned int as well, which counts the number of node that were generated; and
- *runtime*, of type double, that stores the runtime of the A* Search algorithm.

Statistics and Performance Report

Target Computer Specifications

This program was compiled and run on a 64-bit Ubuntu® 20.04.1 LTS with Intel® Core™ i3-6006U CPU @ 2.00GHz. The compiler used was gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04). This program also uses ANSI escape sequences for cursor location control, color, font styling on the terminal, so this program will not run properly on platforms that do not support ANSI escape sequences.

Results and Analysis

The summary of the statistics of the performance of the A* Search algorithm are shown in Table 5. The results shown are the average of five runs of the same combination of test case and heuristic used. Note that these are the results when the program is run on the same machine stated above.

By using the easy test case, the algorithm that used the Manhattan Distance as the heuristic measure performed slightly better than the one which used Hamming Distance in terms of nodes expanded and generated, runtime, and memory used, as illustrated in Figure 3, Panel

A. The difference in the performance was barely noticeable as the algorithm solved the puzzle almost instantly.

The trend in the performance difference, however, continued and gradually became greater as the test cases got higher in level of difficulty. Until in the worst test case, the difference between the performance of the two heuristic measures became so high that it was extremely noticeable during the run of the program. The performance of the algorithm when using the Hamming Distance Heuristic on the worst test case had a runtime of 42.98 minutes while the algorithm that used the Manhattan Distance Heuristic had a runtime of only 0.058 seconds. The amount of memory used and nodes generated and expanded by the former was also 213% more.

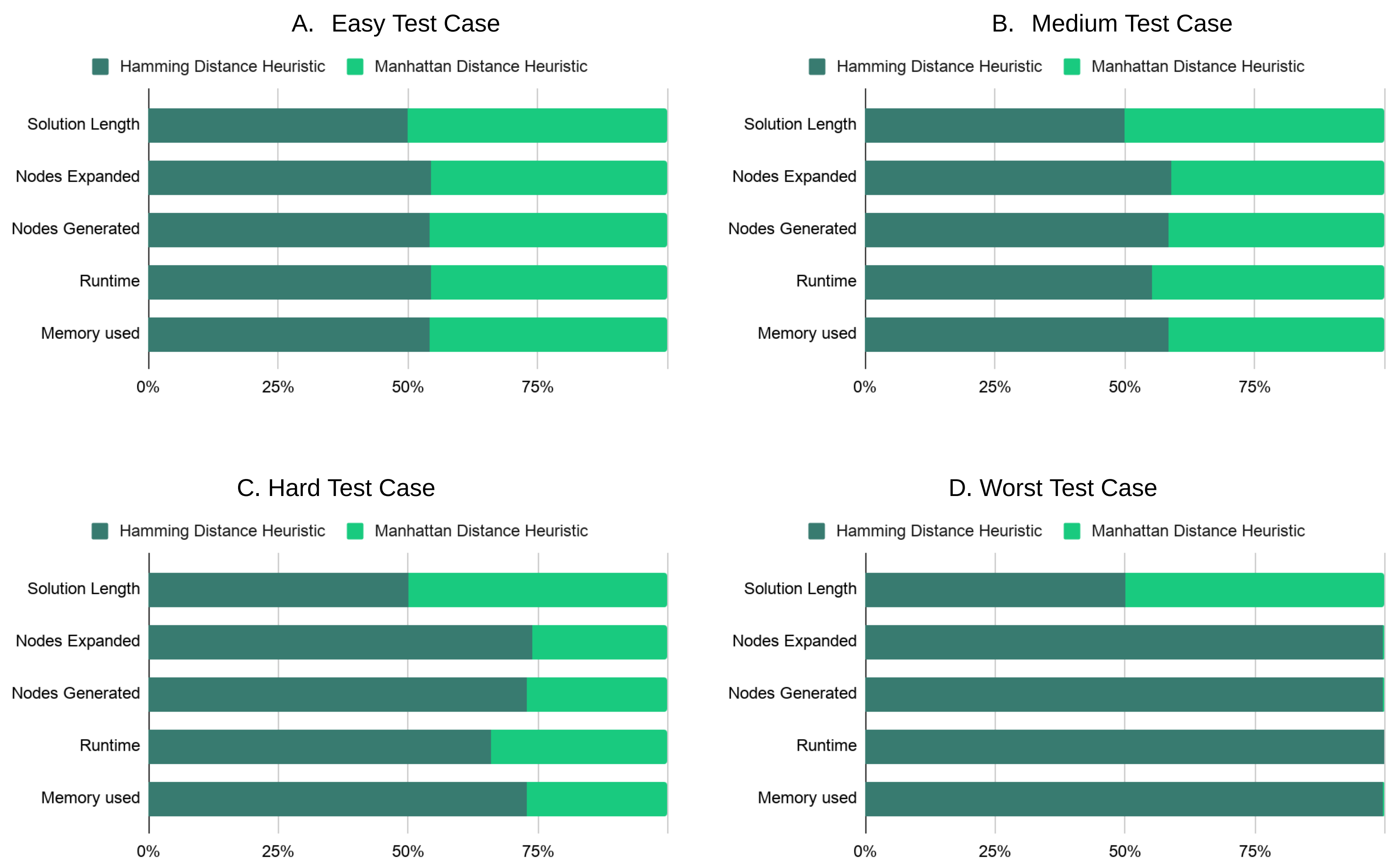
Table 5

Statistics of the Performance of A Search Over Two Different Heuristics*

Test States	Solution Length	Nodes Expanded	Nodes Generated	Runtime (*seconds*)	Memory used (*bytes*)
		Hamming Distance Heuristic			
Easy	5	6	13	0.000214	936
Medium	9	23	45	0.000642	3240
Hard	12	71	134	0.001796	9648
Worst	30	263953	485009	2578.63	34920648
		Manhattan Distance Heuristic			
Easy	5	5	11	0.000179	792
Medium	9	16	32	0.000521	2304
Hard	12	25	50	0.000927	3600
Worst	30	1237	2279	0.058282	164088

Figure 3

Performance of the A Search Over Two Different Heuristics Using Different Test Cases*



Nevertheless, despite the performance differences between the two heuristics, the solution lengths found by the algorithm were still the same.

The result also implies that a direct correlation between the level of difficulty of the test states to time and space complexity of the algorithm is present.

Conclusion

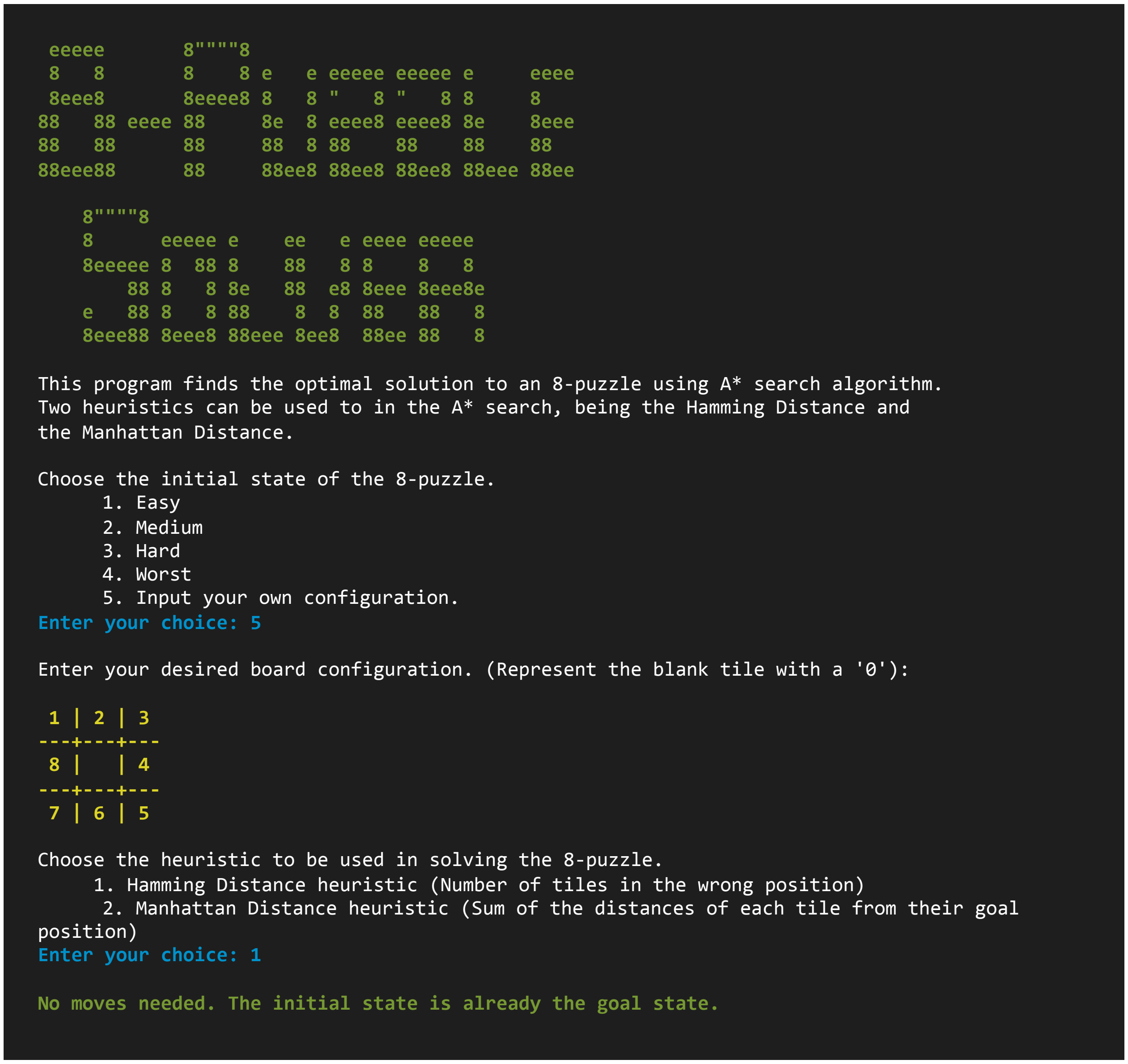
As the results show, the solution length, runtime, amount of memory used, nodes expanded, and nodes generated increases as the test cases get higher in level of difficulty. It can be concluded from the results that with a uniformity of test states, Manhattan Distance heuristic dominates Hamming Distance heuristic in terms of runtime, amount of memory used, nodes expanded, and nodes generated, despite producing the same solution.

Appendix

Sample outputs of the test runs of the program are shown Figures A1 and A2. A custom initial board configuration and the Hamming Distance heuristic were used in Figure A1, while the predetermined easy test case and Manhattan Distance heuristic were used in Figure A2.

Figure A1

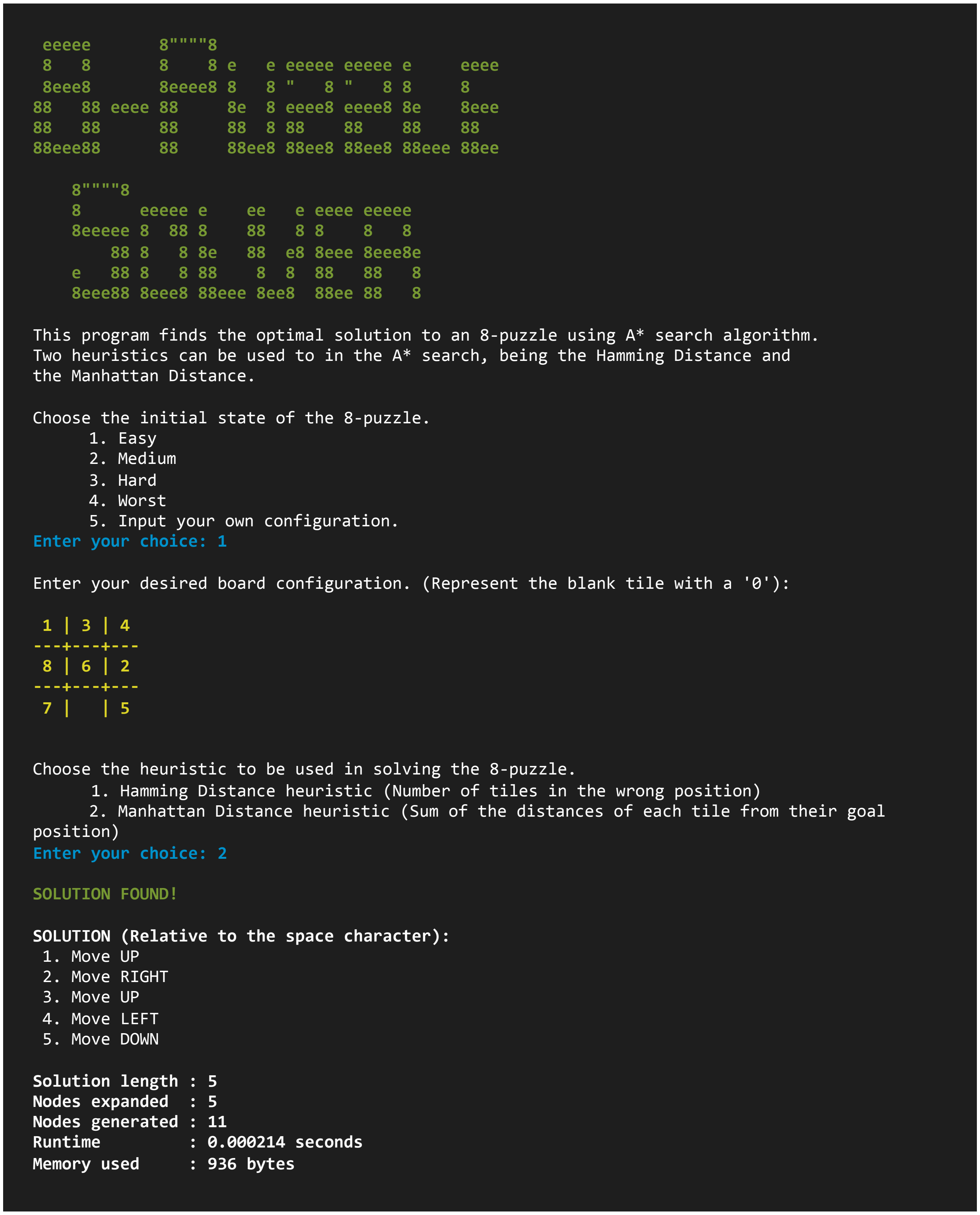
Program Test Run 1



```
 eeeee       8""""8
 8   8       8    8 e   e eeeee eeeee e     eeee
 8eee8       8eeee8 8   8 "   8 "   8 8     8
88   88 eeee 88     8e  8 eeee8 eeee8 8e    8eee
88   88      88     88  8 88    88    88    88
88eee88      88     88ee8 88ee8 88ee8 88eee 88ee

    8""""8
    8      eeeee e    ee   e eeee eeeee
    8eeeee 8  88 8    88   8 8    8   8
        88 8   8 8e   88  e8 8eee 8eee8e
    e   88 8   8 88    8  8  88   88   8
    8eee88 8eee8 88eee 8ee8  88ee 88   8

This program finds the optimal solution to an 8-puzzle using A* search algorithm.
Two heuristics can be used to in the A* search, being the Hamming Distance and
the Manhattan Distance.

Choose the initial state of the 8-puzzle.
      1. Easy
      2. Medium
      3. Hard
      4. Worst
      5. Input your own configuration.
Enter your choice: 5

Enter your desired board configuration. (Represent the blank tile with a '0'):

 1 | 2 | 3
---+---+---
 8 |   | 4
---+---+---
 7 | 6 | 5

Choose the heuristic to be used in solving the 8-puzzle.
     1. Hamming Distance heuristic (Number of tiles in the wrong position)
      2. Manhattan Distance heuristic (Sum of the distances of each tile from their goal
position)
Enter your choice: 1

No moves needed. The initial state is already the goal state.
```

Figure A2

Program Test Run 2



```
 eeeee       8""""8
 8   8       8    8 e   e eeeee eeeee e     eeee
 8eee8       8eeee8 8   8 "   8 "   8 8     8
88   88 eeee 88     8e  8 eeee8 eeee8 8e    8eee
88   88      88     88  8 88    88    88    88
88eee88      88     88ee8 88ee8 88ee8 88eee 88ee

    8""""8
    8      eeeee e    ee   e eeee eeeee
    8eeeee 8  88 8    88   8 8    8   8
        88 8   8 8e   88  e8 8eee 8eee8e
    e   88 8   8 88    8  8  88   88   8
    8eee88 8eee8 88eee 8ee8  88ee 88   8

This program finds the optimal solution to an 8-puzzle using A* search algorithm.
Two heuristics can be used to in the A* search, being the Hamming Distance and
the Manhattan Distance.

Choose the initial state of the 8-puzzle.
      1. Easy
      2. Medium
      3. Hard
      4. Worst
      5. Input your own configuration.
Enter your choice: 1

Enter your desired board configuration. (Represent the blank tile with a '0'):

 1 | 3 | 4
---+---+---
 8 | 6 | 2
---+---+---
 7 |   | 5


Choose the heuristic to be used in solving the 8-puzzle.
      1. Hamming Distance heuristic (Number of tiles in the wrong position)
      2. Manhattan Distance heuristic (Sum of the distances of each tile from their goal
position)
Enter your choice: 2

SOLUTION FOUND!

SOLUTION (Relative to the space character):
 1. Move UP
 2. Move RIGHT
 3. Move UP
 4. Move LEFT
 5. Move DOWN

Solution length : 5
Nodes expanded  : 5
Nodes generated : 11
Runtime         : 0.000214 seconds
Memory used     : 936 bytes
```