BICOL UNIVERSITY
CS-IT Department
2nd Semester 2017 - 2018

8-Puzzle Solver Documentation,

Statistics, and Performance Comparison

Angelo Dina

Christian A. Collamar

BSCS 3-A Students

Arlene Satuito
Professor

1. Source Code Documentation

1.1. Overview

SOURCE FILES	DEFINITION
`main.c`	Defines the main program procedure. This includes our breadth-first and A* implantation.
`list.h`	Defines the data structure for holding a list of nodes as well as its operations.
`node.h`	Defines the data structure for representing a game tree node, which contains the game state and other information. Node operations are also defined here.
`state.h`	Defines the data structure for representing a game state. Operations on states are also defined here.
`io.h`	Contains functions related to input/output operations.

1.2. Data Structures and Operations

1.2.1. States (state.h: `struct State`)

A game state is an instance of all the possible 3x3 board configurations, with numbers ranging from 0 to 8 assigned to each of the nine blocks. Since a state is viewed as a physical representation of the game, we decide to include information about the action that resulted to the corresponding board configuration, instead of placing that information to the node.

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

The number of total states in this definition would be *9!*, although only half of which, are solvable[1].

STATE OPERATIONS	
FUNCTION PROTOTYPE	DESCRIPTION
`State* createState(State *s, Move m);`	Creates and returns anew state derived from the arguments, if `m` is a valid action from `s`. Otherwise, NULL is returned.
`void destroyState(State **s);`	Simply deallocates the state created by `createState()`. And sets `*s` to NULL.
`int manhattanDist(State *c, State *g);`	Evaluates the manhattan distance (the heuristic value) of the given state configuration `c` based on the goal state `g`.

[1]https://en.wikipedia.org/wiki/15_puzzle#Solvability

1.2.2. *Operators* (state.h: `struct Move`)

Operators are all actions available in accordance with the game mechanics. In the case of any normal n-puzzle game, possible actions are up, down, left, and right.

```
3   //this enumerates available movements in the game relative to the blank character
4   typedef enum Move {
5       UP, DOWN, LEFT, RIGHT, //values for moving up, down, left, right, respectively
6       NOT_APPLICABLE         //value assigned for initial and goal input states
7   } Move;
```

TECHNICAL NOTE: `Move::NOT_APPLICABLE` is not an operator. It is only used to be assigned to the input initial and goal states, since there is no action that resulted to the respective board configurations.

1.2.3. *Nodes* (node.h: `struct Node`)

A node constitutes our search tree. It holds information about its parent, children, states, path cost, heuristic cost, and depth, not to be confused with `ListNode`.

```
1   /**
2    * DESCRIPTION: Defines the node structure used to create a search tree
3    **/
4   typedef struct Node Node;
5   struct Node {
6       unsigned int depth; //depth of the node from the root. For A* search,
7                           //this will also represent the node's path cost
8       unsigned int hCost; //heuristic cost of the node
9       State *state;       //state designated to a node
10      Node *parent;       //parent node
11      NodeList *children; //list of child nodes
12  };
```

TECHNICAL NOTE: In the context of this game, the path cost of a node will be how far it is from the root of the tree; that is, the deeper the node is, the more expensive it gets. Since the basis on assigning path costs to a node is correlated to its depth, adding another variable for holding the path cost would be unnecessarily redundant. Therefore, we decide to reuse the node's depth as its path cost to save space.

NODE OPERATIONS	
FUNCTION PROTOTYPE	DESCRIPTION
`Node* createNode(int d, int h, State *s, Node *p);`	Creates and returns anew node with depth `d`, heuristic value `h`, state `s`, and parent node `p`.
`void destroyTree(Node *n);`	Deallocates the whole subtree of nodes from a root node `n`.
`NodeList* getChildren(Node *p, State *g);`	Expands the node `p` and returns a list of child nodes with their computed h-cost based on the goal state `g`.
`int totalCost(Node *n);`	Evaluates and returns the node `n`'s total cost by adding its heuristic value to its path cost.

1.2.4. *Linked List Container* (list.h: `struct NodeList`)

This structure serves as a wrapper for a doubly linked list of `ListNode`s. It is used to carry a list of type `Node`. It contains the number of nodes in the list, as well as a pointer to the head and the tail nodes.

```c
/**
 * 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
};
```

1.2.5. Linked List Nodes (list.h: `struct ListNode`)

This structure is a part of a `NodeList`. It has a pointer to a previous and next `ListNode` in the list, and can hold at most one `Node`.

```c
/**
 * 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;
```

NODELIST & LISTNODE OPERATIONS	
FUNCTION PROTOTYPE	DESCRIPTION
`char pushNode(Node *n, NodeList** l);`	Appends the game tree node `n` to the head of list `l`. Returns 1 on success; 0 on failure.
`Node* popNode(NodeList** l);`	Removes the node at the tail of the list `l` and returns a pointer to it.
`void pushList(NodeList **t, NodeList *l);`	Adds a list of nodes `t`, to the list `l` at the latter's tail.
`void pushListInOrder(NodeList **t, NodeList *l);`	Same as `pushList()` except that the nodes to be added to l are in order based on the nodes' f-cost.

1.2.6. Solution Path (list.h: `struct SolutionPath`)

This structure creates a list of actions to execute to reach the goal, that is, the solution path. Implementation uses a singly linked list.

```c
/**
 * DESCRIPTION:
 *    A structure for holding the solution.
**/
typedef struct SolutionPath {
    Move action;
    struct SolutionPath *next;
} SolutionPath;
```

SOLUTIONPATH OPERATION	
FUNCTION PROTOTYPE	DESCRIPTION
`void destroySolution(SolutionPath **l);`	Deallocates the list `l`'s allocated memory.

1.3. Global Counters

As instructed, the program should be able to collect and print the sequence of moves corresponding to the solution and the following statistics:

- total number of nodes expanded
- total number of nodes generated
- length of the solution path (number of moves)
- sequence of moves corresponding to the solution (e.g. left, right, up, down)

Therefore, aside from the sequence of moves, we have the following global variables with the following purposes:

VARIABLE	DESCRIPTION
`unsigned int nodesExpanded`	Records the current number of nodes expanded
`unsigned int nodesGenerated`	Records the current number of nodes generated
`unsigned int solutionLength`	Counts the number of edges in the solution path

Additionally, to measure performance of the BFS and A* search algorithms for comparison, an additional global variable `runtime` of type `double` is set to record the algorithms' execution time in milliseconds. On the other hand, to obtain the space consumed by each algorithm for the given inputs, `nodesExpanded` is multiplied to the size of the `struct Node`.

2. Statistics and Performance Comparison

2.1. Target Computer Specification

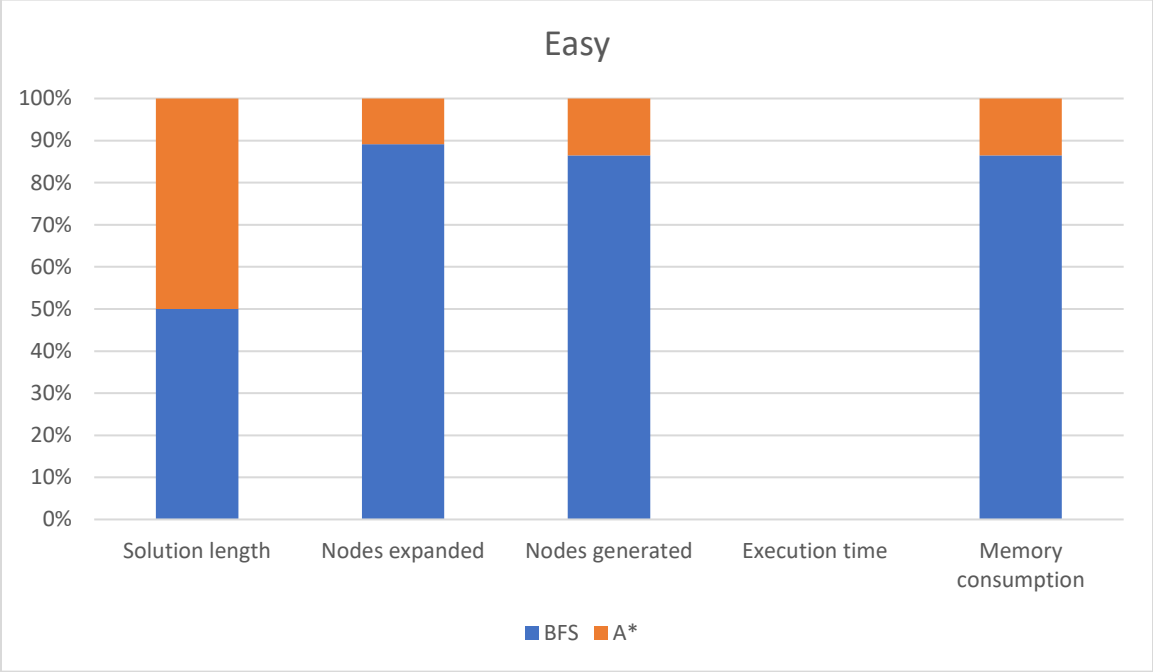
The program is compiled and run as a 32-bit executable on a 64-bit Windows® 7 OS with Intel® Core™ i5 and 8 GB RAM.

2.2. Result

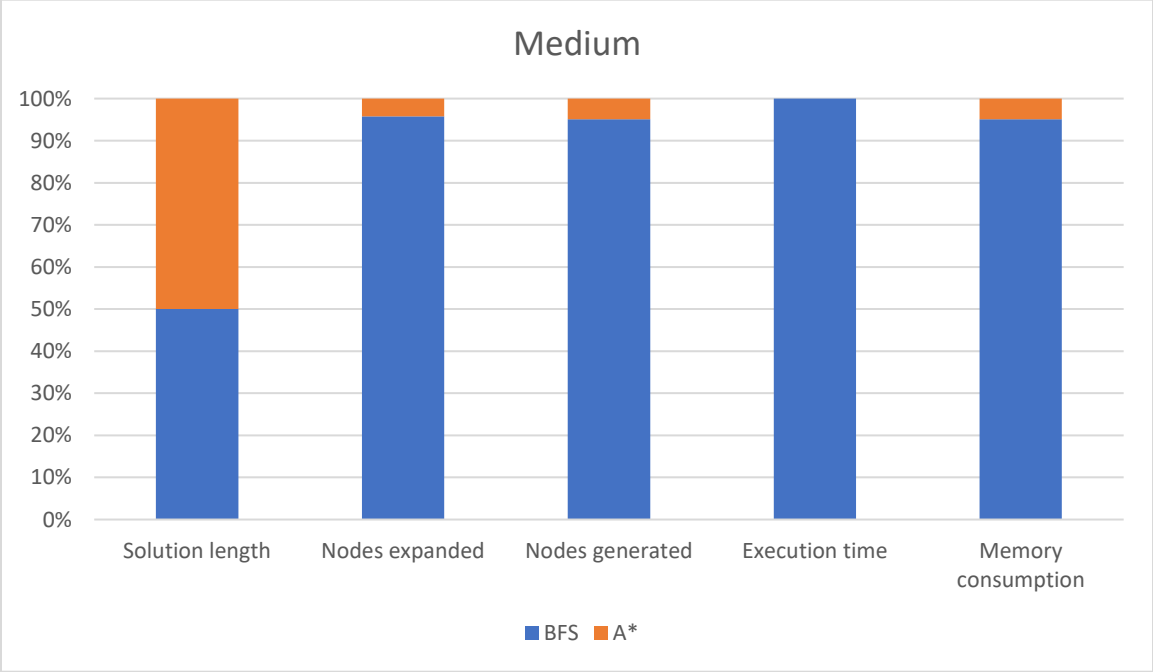
The table shows the summary of the performance of BFS and A* under a series of test cases of varying difficulty. Dashes indicate that the program crashed before obtaining the values.

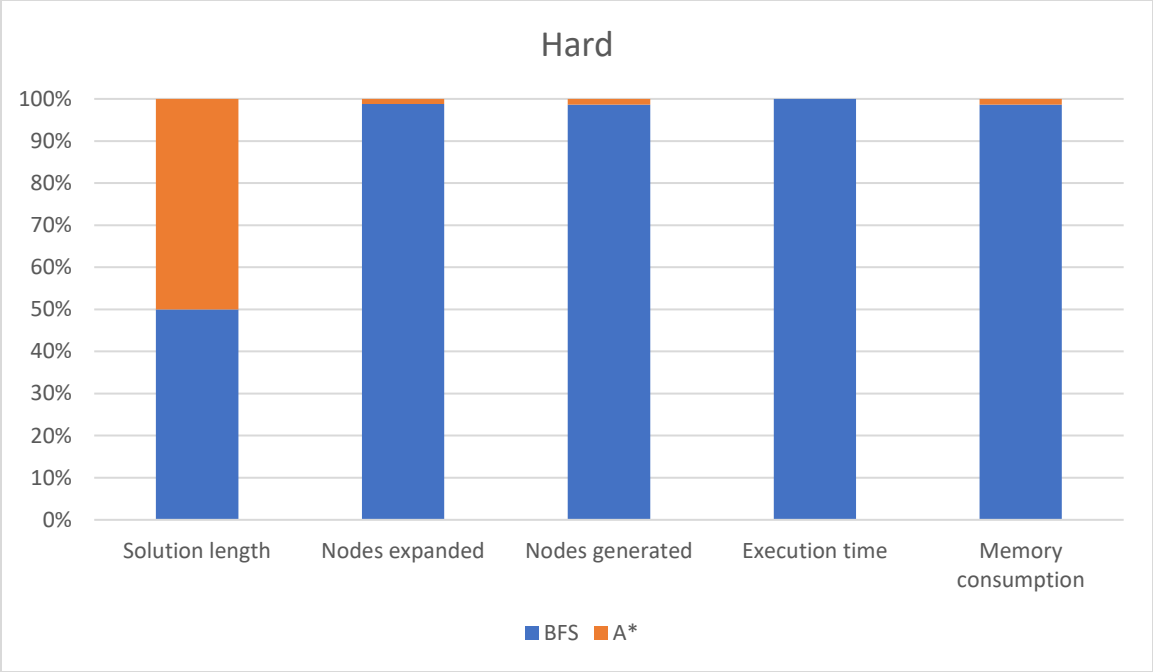
TEST CASE	SEARCH ALGORITHM									
	BFS					A*				
	Solution Length	Nodes Expanded	Nodes Generated	Execution time (ms)	Space used (b)	Solution Length	Nodes Expanded	Nodes Generated	Execution time (ms)	Space used (b)
Easy	5	41	77	0	1540	5	5	12	0	240
Medium	9	385	663	0.001	13260	9	17	34	0	680
Hard	12	2251	3783	0.003	75660	12	27	53	0	1060
Worst	-	-	-	-	-	30	4384	7587	0.087	151740

Below are visualizations of the two algorithms' performance on the four test cases, illustrated using a stacked column; compared against each other by percentage. Smaller area means better performance.



REMARK: Although the chart above implies that both BFS and A* have 0 milliseconds of execution time, this does not mean both programs took the same amount of time; and finished instantaneously just after starting. The reason is because the program can only approximate execution time with precision of a millimeter. The same is true for the next test cases with the same execution time.





REMARK: Since using BFS crashes the program even before finishing, we were not able to collect data and is therefore undefined, so comparing it to the performance of A* would be meaningless.

3. Conclusion

As we can see from the results, A* saves more time and space than BFS search, which reflects the theoretical comparison between the two algorithms' time and space complexities. In fact, while A* solved all four test cases, BFS 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.