8-Puzzle Solver Documentation

8-Puzzle Solver with an Implementation of Breadth-First Search and Depth-First Search

Student:

Michael Ricky - 05111840000078

Teacher:

M.M. Irfan Subakti

Course:

Design and Analysis of Algorithm (D)

Heads Up

8-Puzzle is a game where the player will move (or slide) a piece of puzzle on a game board until it reaches the goal state of the game. It's a game which came straight from the n-puzzle problem, in which the n is 8. The 8 puzzle means there are a total of 8 tiles and 1 blank tile, summed up to 9 tile, a 3 x 3 game board. This applies to it's higher level game, such as the 15-Puzzle, which means there are a total of 15 tiles and 1 blank tile.

It is always interesting to solve this kind of problem using AI (Artificial Intelligence). AI can do the search of the solution within the game and even count how many steps needed to reach the goal state from a given initial state. In this program, the writer is trying to solve the problem using the most popular search, the Breadth-First Search and the Depth-First Search. These searches is as known as the brute force method, where the program will keep searching for all possible result from all different move and combination.

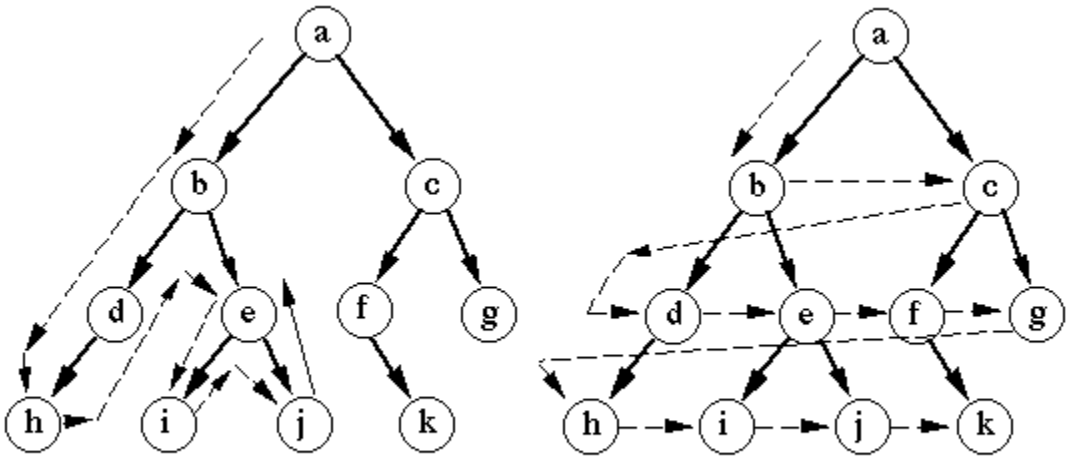
The Brute-Force Method

Brute force is a way of searching by actually trying all the combinations possible for a problem. For example, if we have to search which shoes have the best fit on us, we would try all the pair and choose which one is the best. It is exactly the way brute force works.

Breadth-First Search and Depth-First Search are some of the brute force method. It does the job by looking at all the possible sets and choose which way will lead us to the goal state. They are some of the way we can do the tree traversing.

A Tree is a way we represent the path of the state we will be looking after. A tree has nodes and something called the "depth". In this particular definition, the depth is a tree level, where the node(s) are located.

The main difference between the Breadth-First Search and the Depth-First Search is that the Breadth-First Search will look at one level and all of their neighbors, while Depth-First Search will look until the deepest node and coming back up to the neighbors.



(Source : https://vivadifferences.com/difference-between-dfs-and-bfs-in-artificial-intelligence/)

The Main Idea

We'll start from the node.

Assume that the gameboard and the states are represented by a node.

What's the next move?

We'll need to see which move is possible. If the blank tile is on the bottom, we can't move down (relative to the blank tile). If the blank tile is on the top, we can't move up. If the blank tile is on the leftmost, we can't move left. If the blank tiles is on the rightmost, we can't move right.

Generating Nodes...

From the list of the possible move, we make some new nodes with the moved state and set these nodes as a child of the former node. We then move to the next node based on the algorithm and doing the same thing from the start. We will do this repeatedly until we find the goal state and then backtrack to see the path from the initial state to the goal state.

Program Analysis

There are 2 program, but the difference of the source code is very few so that the DFS Source Code is not included here, but uploaded on the github page. On the program, the main idea is as stated before. In the implementation, the writer use the queue data structure for the BFS technique and the stack data structure for the DFS technique. The rest of the program is the same.

The program works by inputting the numbers needed for the gameboard, representing the initial state. The initial state then inputted into the BFS Search Function. It is where the next possible nodes based on the possible moves are then generated. In the breadth-first search, the list of the possible node is inserted in a *queue*, which means it's FIFO (First In First Out). It will then search the next node based on the order in the queue. The depth-first search utilize a different data structure. It use the stack data structure, which means it's LIFO (Last In First Out). It will keep generating a new node until it reach a point where it can't generate any nodes left.

Because of the nature of DFS, it will take a very long time (a.k.a. infinite) because the node will always be generateable. It can only solve the problem if the move needed to reach the goal is below than 2, otherwise it will stuck on an infinite loop. The BFS, on the other hand, will reach the goal almost faster and mostly will find a way through, though some worst cases will make the computer ran out of memory before even finding a solution.

Each nodes generated will then evaluated for the goal state. If the node is the goal state, it will then backtrack the way from the initial node to reach this node. If it's not, it will then retrying the search again by generating another set of possible nodes.

Git Hub Link : https://github.com/djtyranix/DAA_D_Quiz2

Source Code

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

#define blank '0' // defining a blank tile

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)
```

```
typedef struct ListNode ListNode;
typedef struct NodeList NodeList;
typedef struct Node Node;

typedef enum moves
{
      UP, DOWN, LEFT, RIGHT, NA // the moves applicable to blank tile
} moves;

typedef struct State
{
      moves action;
      char gameboard[3][3]; // The game board is 3 x 3
} State;

struct ListNode
{
    Node *currNode;
    struct ListNode *prevNode; //the node before `this` instance
    struct ListNode *nextNode; //the next node in the linked list
};

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
};

struct Node
{
    unsigned int depth; //depth of the node from the root.
    State *state;       //state designated to a node
    Node *parent;       //parent node
    NodeList *children; //list of child nodes
};

Node* createNode(unsigned int d, State *s, Node *p)
{
      Node *newNode = malloc(sizeof(Node));

      if(newNode)
      {
            newNode->depth = d;
            newNode->state = s;
            newNode->parent = p;
            newNode->children = NULL;

            ++nodesGenerated;
      }

      return newNode;
}

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);
}

State* createState(State *state, moves move)
{
      State *newState = malloc(sizeof(State));

      char i, j;
      char row, column;

      for(i = 0; i < 3; i++)
      {
            for(j = 0; j < 3; j++)
            {
                  if(state->gameboard[i][j] == blank) // getting coordinates
of the blank tile
                  {
                        row = i;
                        column = j;
                  }

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

      if(move == UP && (row - 1) >= 0) // MOVING UP. min row is 0, if lower
than 0 then it's not applicable.
      {
            char temp = newState->gameboard[row - 1][column];
            newState->gameboard[row - 1][column] = blank;
            newState->gameboard[row][column] = temp;
            newState->action = UP;
            return newState;
      }
      else if(move == DOWN && (row + 1) < 3) // MOVING DOWN. max row is 2, if
higher than 2 then it's not applicable.
      {
            char temp = newState->gameboard[row + 1][column];
            newState->gameboard[row + 1][column] = blank;
            newState->gameboard[row][column] = temp;
            newState->action = DOWN;
            return newState;
      }
```

```
      else if(move == LEFT && (column - 1) >= 0) // MOVING LEFT. min column is
0, if lower then it's not applicable.
      {
            char temp = newState->gameboard[row][column - 1];
            newState->gameboard[row][column - 1] = blank;
            newState->gameboard[row][column] = temp;
            newState->action = LEFT;
            return newState;
      }
      else if(move == RIGHT && (column + 1) < 3) // MOVING RIGHT. max column
is 2, if lower then it's not applicable.
      {
            char temp = newState->gameboard[row][column + 1];
            newState->gameboard[row][column + 1] = blank;
            newState->gameboard[row][column] = temp;
            newState->action = RIGHT;
            return newState;
      }

      free(newState);
      return NULL;
}

void destroyState(State **state)
{
      free(*state);
      state = NULL;
}

typedef struct SolutionPath
{
      moves action;
      struct SolutionPath *next;
} solutionPath;

void clearSolution(solutionPath **list)
{
    solutionPath *next;

    while(*list)
    {
        next = (*list)->next;
        free(*list);
        *list = next;
    }

    *list = NULL;
}

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;
}

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;
}

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;
}

char checkGoal(State const *checkState, State const *goalState)
{
      char i, j;

      for(i = 0; i < 3; i++)
      {
            for(j = 0; j < 3; j++)
            {
                  if(checkState->gameboard[i][j] != goalState-
>gameboard[i][j]) // testing each tile so that it matched the goal state.
                  {
                        return 0; // if any of them not match one another, it
return false.
                  }
            }
      }
```

```
	return 1; // else, return true.
}

void inputState(State * const state)
{
	state->action = NA;
	char row, column;
	int temp;

	for(row = 0; row < 3; row++)
	{
		for(column = 0; column < 3; column++)
		{
			printf("gameboard[%i][%i] : ", row, column);
			scanf("%i", &temp);
			state->gameboard[row][column] = temp + '0';
		}
	}
	printf("\n");
}

void inputGoalState(State * const state)
{
	state->action = NA;
	state->gameboard[0][0] = '1';
	state->gameboard[0][1] = '2';
	state->gameboard[0][2] = '3';
	state->gameboard[1][0] = '8';
	state->gameboard[1][1] = '0';
	state->gameboard[1][2] = '4';
	state->gameboard[2][0] = '7';
	state->gameboard[2][1] = '6';
	state->gameboard[2][2] = '5';

	printf("\n");
}

void printBoard(char const gameboard[3][3])
{
	char row, column;

	for(row = 0; row < 3; row++)
	{
		printf("+---+---+---+\n");
		for(column = 0; column < 3; column++)
		{
			if(gameboard[row][column] == '0')
			{
				printf("|   ");
			}
			else
			{
				printf("| %c ", gameboard[row][column]);
			}
		}
		printf("|\n");
	}
	printf("+---+---+---+\n");
}
```

```
void printSolution(solutionPath *path)
{
      if(!path)
      {
            printf("No solution found.\n");
            return;
      }

      if(!path->next)
      {
            printf("The initial state is the goal state.\n");
            return;
      }

      printf("Solution : (Moves are based on the blank tile moves)\n");

      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("%d. Move %s\n", counter, move[path->action]);
    }

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

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, testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != UP && (testState = createState(parent->state,
DOWN))) {
        child = createNode(parent->depth + 1, testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
```

```
    if(parent->state->action != RIGHT && (testState = createState(parent-
>state, LEFT))) {
        child = createNode(parent->depth + 1, testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }
    if(parent->state->action != LEFT && (testState = createState(parent-
>state, RIGHT))) {
        child = createNode(parent->depth + 1, testState, parent);
        pushNode(child, &parent->children);
        pushNode(child, &childrenPtr);
    }

    return childrenPtr;
}

solutionPath* BFS_search(State *init, State *goalState);

int main()
{
      nodesExpanded = 0;
    nodesGenerated = 0;
    solutionLength = 0;
    runtime = 0;

      printf("Hello! Welcome to the 8-puzzle solver!\n");

      State init;
      State goalState;

      solutionPath *bfs;

      printf("Please input the initial state:\n");
    inputState(&init);

    inputGoalState(&goalState);

    printf("Here's the initial board state:\n");
    printBoard(init.gameboard);

    printf("Here's the board's goal state:\n");
    printBoard(goalState.gameboard);

    printf("\n------------------------ USING BFS ALGORITHM ------------------
--------\n");
    bfs = BFS_search(&init, &goalState);
    printSolution(bfs);

    //free resources
    //clearSolution(&bfs);
}

solutionPath* BFS_search(State *init, State *goalState)
{
      NodeList *queue = NULL;
    NodeList *children = NULL;
    Node *node = NULL;

    clock_t start = clock();
```

```
    pushNode(createNode(0, init, NULL), &queue);
    Node *root = queue->head->currNode;

    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(checkGoal(node->state, goalState))
        {
            break;
        }

        //else, expand the node and update the expanded-nodes counter
        children = getChildren(node, goalState);

        ++nodesExpanded;

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

    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;


}
```

Running Test Case and Result

Here is the goal state of the program :

1	2	3
8		4
7	6	5

Here are some test cases :

1	3	4
8	6	2
7		5

2	8	1
	4	3
7	6	5

2	8	1
4	6	3
7	5	

And here is the result screenshots :

```
Hello! Welcome to the 8-puzzle solver!
Please input the initial state:
gameboard[0][0] : 1
gameboard[0][1] : 3
gameboard[0][2] : 4
gameboard[1][0] : 8
gameboard[1][1] : 6
gameboard[1][2] : 2
gameboard[2][0] : 7
gameboard[2][1] : 0
gameboard[2][2] : 5


Here's the initial board state:
+---+---+---+
| 1 | 3 | 4 |
+---+---+---+
| 8 | 6 | 2 |
+---+---+---+
| 7 |   | 5 |
+---+---+---+
Here's the board's goal state:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+

------------------------- USING BFS ALGORITHM -------------------------
Solution : (Moves are based on the blank tile moves)
1. Move UP
2. Move RIGHT
3. Move UP
4. Move LEFT
5. Move DOWN
DETAILS:
 - Solution length : 5
 - Nodes expanded  : 41
 - Nodes generated : 77
 - Runtime         : 0.000000 milliseconds
 - Memory used     : 2464 bytes

--------------------------------
Process exited after 11.11 seconds with return value 155
Press any key to continue . . .
```

```
Hello! Welcome to the 8-puzzle solver!
Please input the initial state:
gameboard[0][0] : 2
gameboard[0][1] : 8
gameboard[0][2] : 1
gameboard[1][0] : 0
gameboard[1][1] : 4
gameboard[1][2] : 3
gameboard[2][0] : 7
gameboard[2][1] : 6
gameboard[2][2] : 5


Here's the initial board state:
+---+---+---+
| 2 | 8 | 1 |
+---+---+---+
|   | 4 | 3 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Here's the board's goal state:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+

------------------------- USING BFS ALGORITHM -------------------------
Solution : (Moves are based on the blank tile moves)
1. Move UP
2. Move RIGHT
3. Move RIGHT
4. Move DOWN
5. Move LEFT
6. Move LEFT
7. Move UP
8. Move RIGHT
9. Move DOWN
DETAILS:
 - Solution length : 9
 - Nodes expanded  : 385
 - Nodes generated : 663
 - Runtime         : 0.002000 milliseconds
 - Memory used     : 21216 bytes

--------------------------------
Process exited after 15.6 seconds with return value 158
Press any key to continue . . .
```

```
Hello! Welcome to the 8-puzzle solver!
Please input the initial state:
gameboard[0][0] : 2
gameboard[0][1] : 8
gameboard[0][2] : 1
gameboard[1][0] : 4
gameboard[1][1] : 6
gameboard[1][2] : 3
gameboard[2][0] : 7
gameboard[2][1] : 5
gameboard[2][2] : 0


Here's the initial board state:
+---+---+---+
| 2 | 8 | 1 |
+---+---+---+
| 4 | 6 | 3 |
+---+---+---+
| 7 | 5 |   |
+---+---+---+
Here's the board's goal state:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+

------------------------- USING BFS ALGORITHM -------------------------
Solution : (Moves are based on the blank tile moves)
1. Move LEFT
2. Move UP
3. Move LEFT
4. Move UP
5. Move RIGHT
6. Move RIGHT
7. Move DOWN
8. Move LEFT
9. Move LEFT
10. Move UP
11. Move RIGHT
12. Move DOWN
DETAILS:
 - Solution length : 12
 - Nodes expanded  : 2251
 - Nodes generated : 3783
 - Runtime         : 0.008000 milliseconds
 - Memory used     : 121056 bytes

--------------------------------
Process exited after 15.89 seconds with return value 162
Press any key to continue . . .
```