BICOL UNIVERSITY
CS-IT Department
A.Y. 2021-2022

8 – Puzzle Solver Documentation: Discussion, Analysis and Performance Comparison

Nestor Rafael Oscillada (Code, Documentation)
Paul Marren Claveron (Code, Documentation)
Bradford Lester Bonto (Code, Documentation)
Jan Vincent Rada (Code, Documentation)
BSCS 3B

Arlene Satuito
Professor

Overview of the Source code

This program is a C++ implementation to the 8-Puzzle problem which uses A* Search and Iterative Deepening Search (IDS). IDS is a blind search strategy whilst A* is an informed search strategy which uses a heuristic function. A given goal state is declared where the following input states should reach the said goal using both search strategies. The program returns the number of nodes expanded, cost of the solution path, the path taken from the initial state to the goal state, and the runtime of the program in milliseconds.

About the 8-Puzzle Problem

The 8-Puzzle problem is being played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. The goal is to rearrange the blocks so that they are in order. In this iteration of the problem, there are different initial states for the puzzle to start from and there is a given goal state for the puzzle to achieve.

A puzzle configuration is considered as solvable, if there exists a sequence of actions, which leads to the goal configuration. This holds true for exactly half of all possible puzzle configurations [1].

Source Code Documentation

Source File	Definition
main.cpp	Includes the main source code for the program. This consist of our implementation for Iterative Deepening Search (IDS), and A* search
class.h	This header file includes all the data structure that is used in the main program. This also includes the functions to be used.
input.h	This header file includes the input and output for the 8-Puzzle.

Class State ("class.h")

Is the prime function of the **"class.h"** header file, that is the header file for the data structure of the main program, **class State** will be our user-defined data structure function declaration. "public:" access specifier is need for them to access and call by the functions in this header file.

- **State *parent** will be the declared root node of the search, it will be used in the main.cpp to indicate as the last node to close the expansion of the search.
- **int s[n][n]** will be indicate the state matrix of the puzzle (to present the user the assigned location of their inputted value: the initial and goal state). Also it will be use as the flag and counter for the manhattanHeuristic() function and give the Manhattan cost of the system.
- **int moves** will be used as a variable to present the action of the used search algorithm.
- **int g** will count the cost of the initial state to the current state of the search, also it will be added to the **int h** to get the total cost of the search.
- **int h** will count the cost of the current state to the goal state.
- **int totalCost** is the variable to be used to present the solution cost of the search.

```
class State
{
public:
    State *parent;         // The parent state
    int s[n][n];           // The state of the puzzle
    int moves;             // The number of moves to reach this state
    int g, h, totalCost; // g: cost from initial state to this state
                           // h: heuristic cost from this state to goal state
                           // totalCost: g + h

    State();         // Default constructor for State
    bool is_goal(); // Check if this state is the goal state
    bool operator==(const State &) const;
    bool operator<(const State &) const;
    void manhattanHeuristic(); // Calculate the heuristic cost
};
```

- **State()** will be Default constructor function in initializing the value of **int g, h, totalCost** and the parent node (**parent**)

```
State::State()
{
    g = h = totalCost = 0; // Initialize g, h, and totalCost
    parent = NULL;         // Initialize parent
}
```

- **bool is_goal()** will be the function to check if the node is in the goal state already.

```
bool State::is_goal()
{

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            // if any two items not equal - not goal
            if (s[i][j] != Goal[i][j])
                return false;
        }
    }
    //reaching this point means all items are equal - goal state found
    return true;
}
```

- **bool operator==(const State &) const** will check the value of the present state of the node (e.g. **State &r**) if any of the two items are equal or not, if yes the returning value is true if not then the returning value is false.

```
bool State::operator==(const State &r) const
{

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            //if any two items not equal - not goal
            if (s[i][j] != r.s[i][j])
                return false;
        }
    }
    // states are equal
    return true;
}
```

- **bool operator<(const State &) const** will this function is used to determine the overall solution cost of the A* search. If the value of aSearch will be true, then the function will be returning the value of the totalCost (g+h, g as the path cost of the initial state to the current state, and h as the path cost of the current state to the goal state).

```cpp
bool State::operator<(const State &r) const
{

    if (aSearch)
    { // for heuristic based algorithms
        return totalCost < r.totalCost;
    }
    else
    {
        return g < r.g; // for normal search algorithms
    }
}
```

- **void State::manhattanHeuristic()** in this function it will determine the Heuristic path cost of the used Heuristic Search (A*).

```cpp
void State::manhattanHeuristic()
{
    int h_val = 0;              // Initialize temporary holder for heuristic value
    bool found;                 // Flag to check if the value is found
    for (int i = 0; i < n; i++) // Loop through the rows of Goal state
    {
        for (int j = 0; j < n; j++) // Loop through the columnns of Goal state
        {
            found = false;                // Reset the flag
            for (int i2 = 0; i2 < n; i2++) // Loop through the rows of current state
            {
                for (int j2 = 0; j2 < n; j2++) // Loop through the columns of current state
                {

                    if (Goal[i][j] == s[i2][j2]) // If the value is found
                    {
                        //dx + dy
                        h_val += abs(i - i2) + abs(j - j2); // Calculate the heuristic value from the current state to the goal state
                        found = true;
                    }
                    if (found)
                        break;
                }
                if (found)
                    break;
            }
        }
    }
    h = h_val; // Copy the heuristic value from temporary holder to h
}
```

- **list<State> closedList, fringeList,** this list declares the close and open list of the current nodes/fringe inside the main program.

- **State** declares the possible state of the node such as the start state, current state, temporary value (to store temporary value for switching values), and the goal state.

```
list<State> closedList, fringeList; // closed list and active list for fringe
State startState, currentState, tempState;
```

- **extern int Goal[][]** and **extern bool aSearch** is declared as an external variable to be used inside the **"class.h"** and also for the main program (by such to indicate the value of **aSearch** if it is true then proceed to determine the heuristic of the search algorithm)

```
#define n 3
using namespace std;

// External variables from main file
extern int Goal[n][n];
extern bool aSearch;
```

Input and Output ("input.h")

This header consists of functions: inputState() and printMatrix(). Its purposes are to prompt and get input from the user for state configuration, and to display the current game board.

- int i, j will serve as the variables for the indexes of the state's rows and columns. These two variables act as indicators for the exact location of a node in a state.
- int holdCheck[9] serves as a storage for the variables already entered by the user.
- int holder will temporarily hold the variables entered by the user; such that, it will only pass the value to the current state, if and only if it satisfies the given conditions.

```
void inputState()
{
    // Initial configuration
    // Value 0 is used for empty space
    /* 2D array declaration*/
    /*Counter variables for the loop*/
    int i, j;
    int holdCheck[9] = {0};

    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            int holder;
            cout << "Board [" << i << "][" << j << "]:";
            cin >> holder;
```

```cpp
// Function to print N x N matrix
void printMatrix(int const s[n][n])
{
    cout << "\n";
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
            if (s[i][j] == '0')
                cout << "|  ";
            else
                cout << "| " << s[i][j] << " ";
        cout << "| \n";
    }
    cout << "\n";
}
```

Main Program ("main.cpp")

The main program consists of C++ implementation for the A* search and Iterative Deepening Search (IDS). It uses the two headers "input.h" and "class.h." The user is asked for an input and goal state, after which they are given the option to choose between IDS and A* Search.

- **bool aSearch** default value is false. It is a flag which is used for a function declared in "class.h" where the said function is also used in A* Search.
- **int Goal[n][n]** stores the goal state.
- **const char *action[4]** stores the value "UP, DOWN, LEFT, RIGHT," which is used in the print function for the solution path.

```cpp
bool aSearch = false;
int Goal[n][n];
const char *action[4] = {"UP", "DOWN", "LEFT", "RIGHT"};
```

- **void Expand()** expands the current node and updates the current state to the next state according to the movement of the blank tile or 0. The variable **"i"** checks if the blank tile is not in the first and last row. If the blank tile is not in the first row, it will swap the blank tile UP, and if the blank tile is not in the last row, it will swap the blank tile DOWN. The variable **"j"** checks if the blank tile is not in the first and last column. If the blank tile is not in the first column, it will swap the blank tile to the LEFT, and if the blank tile is not in the last column, it will swap the blank tile to the RIGHT.

```
void Expand()
{

    closedList.push_back(currentState);

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            //find blank tile in state array
            if (currentState.s[i][j] == 0)
            {
                // if not in the first row
                if (i > 0)
                {
                    //set child
                    tempState = currentState;
                    tempState.parent = &(closedList.back());
                    // shift blank tile UP
                    swap(tempState.s[i][j], tempState.s[i - 1][j]);
                    // search for the child in the closed list
                    // if the child not found in the closed list
                    if (!InClosed(tempState))
                    {
                        tempState.g += 1;
                        tempState.manhattanHeuristic();
                        tempState.totalCost = tempState.g + tempState.h;
                        tempState.moves = 0;
                        fringeList.push_front(tempState);
                    }
                }
                //if not in last row
                if (i < n - 1)
                {
                    tempState = currentState;
                    tempState.parent = &(closedList.back());
                    //shift the blank tile DOWN
                    swap(tempState.s[i][j], tempState.s[i + 1][j]);
                    if (!InClosed(tempState))
                    {
                        tempState.g += 1;
                        tempState.manhattanHeuristic();
                        tempState.totalCost = tempState.g + tempState.h;
                        tempState.moves = 1;
                        fringeList.push_front(tempState);
                    }
                }
```

```
                // if not in the first column
                if (j > 0)
                {
                    tempState = currentState;
                    tempState.parent = &(closedList.back());
                    // shift blank tile LEFT
                    swap(tempState.s[i][j], tempState.s[i][j - 1]);
                    if (!InClosed(tempState))
                    {
                        tempState.g += 1;
                        tempState.manhattanHeuristic();
                        tempState.totalCost = tempState.g + tempState.h;
                        tempState.moves = 2;
                        fringeList.push_front(tempState);
                    }
                } // if not in the last column
                if (j < n - 1)
                {
                    tempState = currentState;
                    tempState.parent = &(closedList.back());
                    // shift blank tile RIGHT
                    swap(tempState.s[i][j], tempState.s[i][j + 1]);
                    if (!InClosed(tempState))
                    {
                        tempState.g += 1;
                        tempState.manhattanHeuristic();
                        tempState.totalCost = tempState.g + tempState.h;
                        tempState.moves = 3;
                        fringeList.push_front(tempState);
                    }
                }
            }
        }
    }
    //remove the current state from the fringe list
    fringeList.remove(currentState);
}
```

- **void PrintPath(State *r)** prints the solution path from the initial state to the goal state.

```
void PrintPath(State *r)
{
    //print the solution path
    int counter = 0;
    int rg = r->g; //set the current state's g value
    int temp[rg];  //create a temporary array to store the solution path
    int hold;      //hold the value of the current state's g value

    for (r; r; r = r->parent, counter++) //while the parent of the current state is not null
    {
        temp[counter] = r->moves; //add the move to the temp array, this is to reverse the order of the moves.
    }

    for (int i = 1; i < rg + 1; i++)
    {
        hold = temp[rg - i];              //hold the value of the current index
        cout << i << ". " << action[hold]; //print the move
    }
}
```

- **bool InClosed(State &r)** checks whether the state is in the closed list. If the state is in the closed list it will return true, and if not it will return false.

```
bool InClosed(State &r)
{
    for (list<State>::iterator it = closedList.begin(); it != closedList.end(); ++it) //iterate through the closed list
    {
        if ((*it) == r) //if the state is found in the closed list
        {
            return true;
        }
    }
    return false;
}
```

Runtime Statistics

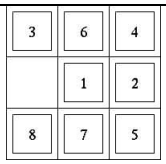
The goal of the following runtime comparison is to find which search method is faster and more optimized in terms of finding the solution path. By trying to run the program on a semi-controlled state, the final output is achieved. Although the built-in antivirus Windows Defender was running during the test, its effect for discrepancy is minimal.

The resulting compiled program is a 64-bit executable file. Using VS Code's built-in runner, the program is executed on a 64-bit Windows 10 OS with an AMD Ryzen 5 3600 processor and 16 GB of RAM set at 3200hz.

Table of Results

The table presents the summary for the runtime of IDS and A* Search on every given difficulty. Solution Path includes the movement taken by the search strategy to achieve the goal state, solution cost includes the number steps taken, Number of nodes expanded include the nodes that were visited and expanded before reaching the goal node, and the running time includes the total time taken by each search strategy to achieve the desired goal state.

Initial State		**IDS**	**A* Search**
Easy	Solution Path	UP – RIGHT – UP – LEFT – DOWN	UP – RIGHT – UP – LEFT – DOWN
	Solution Cost	5	5
	Number of nodes expanded	56	5
	Running Time	0ms	0ms
Medium	Solution Path	UP – RIGHT – RIGHT – DOWN – LEFT – LEFT – UP – – RIGHT – DOWN	UP – RIGHT – RIGHT – DOWN – LEFT – LEFT – UP – – RIGHT – DOWN
	Solution Cost	9	9
	Number of nodes expanded	634	16
	Running Time	3ms	0ms
Hard	Solution Path	LEFT – UP – LEFT – UP – RIGHT – RIGHT – DOWN – LEFT – LEFT – UP – RIGHT DOWN	LEFT – UP – LEFT – UP – RIGHT – RIGHT – DOWN – LEFT – LEFT – UP – RIGHT DOWN
	Solution Cost	12	12
	Number of nodes expanded	6298	25
	Running Time	120ms	0ms
Worst	Solution Path	RIGHT – DOWN – LEFT – LEFT – UP – UP – RIGHT – RIGHT – DOWN – DOWN – LEFT – LEFT – UP – UP – RIGHT – DOWN – DOWN – LEFT – LEFT – UP – UP – RIGHT –	UP – RIGHT – DOWN – DOWN – LEFT – LEFT – UP – UP – RIGHT – DOWN – DOWN – LEFT – LEFT – UP – UP – RIGHT – RIGHT – DOWN – DOWN – LEFT – LEFT – UP – UP – RIGHT –

		DOWN – DOWN – LEFT – UP – LEFT – DOWN – RIGHT – UP	RIGHT – DOWN – DOWN – LEFT – UP
	Solution Cost	32	30
	Number of nodes expanded	424473	1185
	Running Time	264616ms	45ms
 Preferred initial configuration	Solution Path	RIGHT – UP – LEFT – DOWN – DOWN – RIGHT – UP – RIGHT – UP – LEFT – DOWN	RIGHT – UP – LEFT – DOWN – DOWN – RIGHT – UP – RIGHT – UP – LEFT – DOWN
	Solution Cost	11	11
	Number of nodes expanded	1303	13
	Running Time	9ms	0ms

Conclusion

The result proves that A* Search is faster compared to IDS which is true with the stated time complexities, where A* time complexity is exponential to the heuristics given whilst IDS time complexity is $O(b^d)$. On given occasions, A* gives the most optimized solution path compared to IDS. This is evident on the worst case when IDS' solution had a higher cost of 32 compared to A* Search that has a cost of 30.