A Study of Search Algorithms and Heuristics Functions: Iterative Deepening A Star and Recursive Best First Search, Manhattan Distance and Walking Distance

Abstract

In this experiment, we implemented and compared two memory-bounded heuristic search algorithms, IDA* and RBFS for the 15-puzzle problem. In addition, we also implemented the Manhattan distance and the walking distance heuristic functions to see the performance of heuristics regarding to node expansion and their effects of the algorithm performance. As the result, we found that the node search and CPU-time can be the important factors to determine the performance of heuristic function and algorithm.

I. Introduction

In this paper, we implemented two memory bounded heuristic search algorithms, IDA* and RBFS, applying heuristic functions in order to solve a 15 puzzle game. The goal of this puzzle is to locate each numbered tiles in the four by four array in order. We applied Manhattan Distance heuristic function which is well-known admissible function. In addition, we applied Walking Distance heuristic function to improve the efficiency of the search. Firstly, the description of each search algorithm and heuristic function will be described, and the experimental setup will be explained. The comparison of the performance measured by the length of the solution found, the number of nodes searched, and the total CPU time spent on evaluating the heuristic and on solving the whole problem will be introduced. Lastly, we will discuss questions asked in the instruction of this assignment.

II. Description

1. Memory Bounded Heuristic Search Algorithms

i. Iterative Deepening A Star (IDA*)

Applying the iteration to A* search is the simplest way to reduce memory requirements. As the result, the IDA* algorithm can solve the memory issue. Instead of using depth in standard iterative deepening, IDA* uses the f-cost(g+h) as the cutoff at each iteration. The pseudocode of IDA* search algorithm is as following:

```
// Pseudocode for IDA*
function IDA*(node 'n', depth 'd', bound 'b')
{
        if (f value of node 'n' > bound 'b')
        return f value of node 'n';
        if node 'n' is goal
        return 'found';

        minimum = max value;
        for each child 'ni' in successors{
        'returned value' = IDA*('ni', 'd' + 1, b);
        if ('returned value' is 'found')
                return found;
        if ('returned value' is smaller than minimum)
                mimimum = 'returned value';
        }
        return minimum;
}
```

ii. Recursive Best First Search (RBFS)

RBFS is a simple recursive algorithm that improves upon heuristic search by reducing the memory requirement. RBFS uses only linear space and it attempts to mimic the operation of standard best-first search. Its structure is similar to recursive depth-first search but it doesn't continue indefinitely down the current path, the f_limit variable is used to keep track of the f-value of the best alternative path available from any ancestor of the current node. RBFS remembers the f-value of the best leaf in the forgotten subtree and can decide whether it is worth re-expanding the tree later. However, RBFS still suffers from excessive node regeneration. The pseudocode of RBFS algorithm is as following:

```
// Pseudocode for RBFS
function RBFS( node n, depth, f-limit)
{
        if ( node n is goal)
        solution  node n;
        else
        successors = expand(n);

        if (successor is empty)
        return max value;

        for each child 'ni' in successors{
        if (F('ni') is larget than f - limit of 'n')
                F('ni') = max(F('ni'), F('ni'))
        else
                F('ni') = f value of 'ni'
        }

        sort(successors, successors + size());
        n1 = best;
        n2 = alternative;

        while (f value of 'n1' < f - lilmit and F(n1) < max value) {
        n1 = RBFS(n1, min(f - limit, F(n2)));
        sort(successors, successors+size());
        n1 = best;
        n2 = alternative;
        }

        return F(n1);
}
```

2. Heuristic Functions

We applied two admissible heuristics in the idea of relaxed problem that has fewer constraints or preconditions on actions.

i. Manhattan Distance (MD)

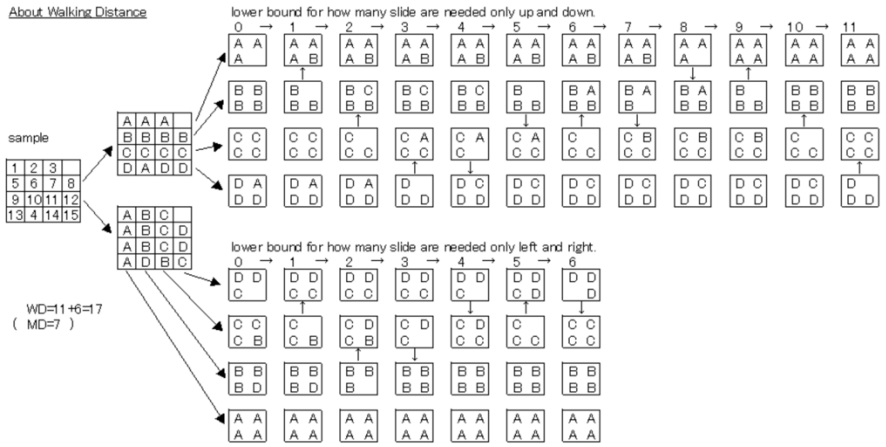
The most well-known admissible heuristic function for 15 puzzle is Manhattan Distance. It counts the distance by the sum of the horizontal and vertical distance because tiles cannot move along diagonals. The pseudocode of Manhattan Distance heuristic function is as following:

```
// Pseudocode for MD
function heuristic(node) =
    dx = abs(node.x - goal.x)
    dy = abs(node.y - goal.y)
    return D * (dx + dy)
```

ii. Walking Distance (WD)

We applied Walking Distance heuristic function to improve the efficiency of the search. Walking Distance is a sophisticated lower bound for how many moves are needed to solve an arbitrary board configuration. The idea

of Walking Distance is based on pattern databases, and its vertical and horizontal moves are counted separately [1]. The way to counting moves are shown below:



III. Experimental Setup

As mentioned, this paper experimented the performance of IDA* and RBFS searches on 15 puzzle problem with two different heuristic functions, MD and WD. In the experiment, we implemented those search algorithms and heuristic functions in C++ as the first trial, and ran programs 10 different 15 puzzle problems in 10, 20, 30, 40, and 50 steps each. However, it was not successful. There were two different reasons; the purpose of this experiment is to measure and compare the performance of two different algorithms and two different heuristic functions. However, we thought finding optimally solvable puzzles only could be our input, but it was very difficult to find them to make a data set for this experiment. Thus, our logic to implement those algorithms became twisted harshly. After the introspection of the first trial, we tried to implement them in Python. We sill had a hard time to debug RBFS search algorithm, but we successfully recorded the result of the experiment. As the result, we measured the length of the solution found, the number of nodes searched, and the total CPU time spent on evaluating the heuristic and on solving the whole problem for the performance comparison. We ran the python program in Ubuntu 16.04 docker environment with quad-core 2.2Ghz Intel Core i7 CPUs and 16 GB memories. The process of the experiment is described as following:

```
For m= 10, 20, 30, 40, 50 do
  For n=10 random problems p generated by Scramble(m)
    For the two algorithms A
      For each heuristic function h,
        Solve p using A and h
        Record the length of the solution found, the number of nodes
          searched, and the total CPU time spent on evaluating
          the heuristic and on solving the whole problem.
```

IV. Results

Following plots are the results of the experiment. Details will be discussed in Discussion and Conclusion section.

Figure 1

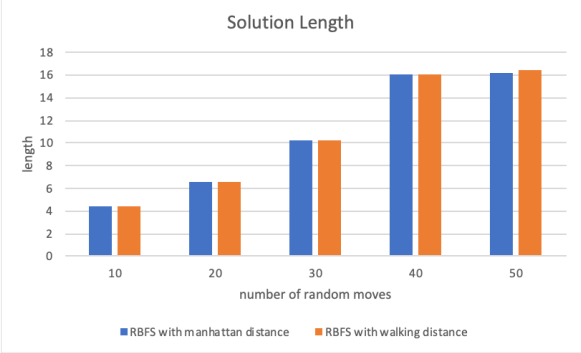


Figure 2

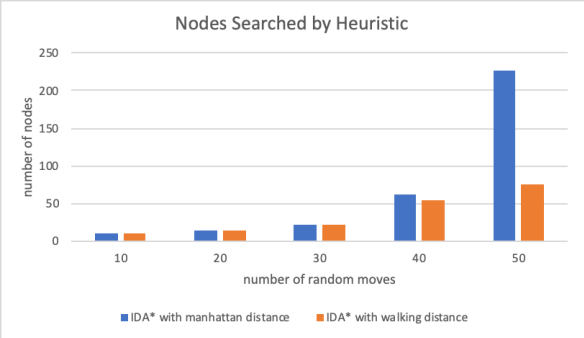


Figure 3

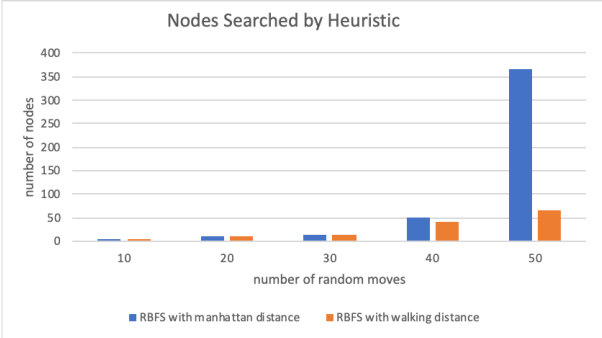


Figure 4

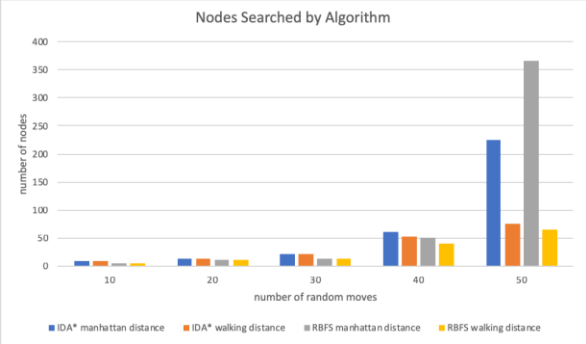


Figure 5

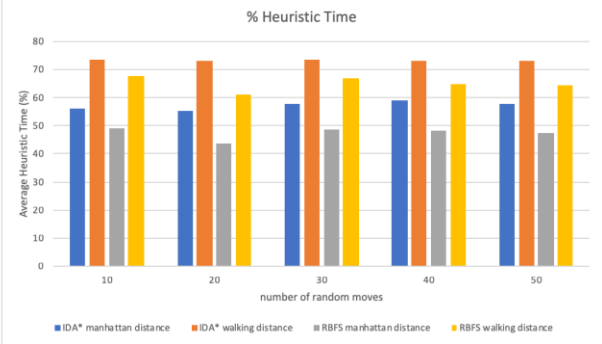


Figure 6

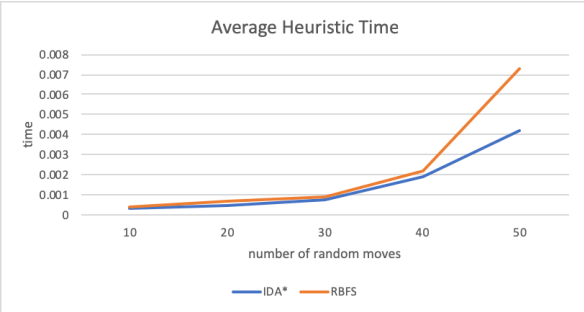


Figure 7

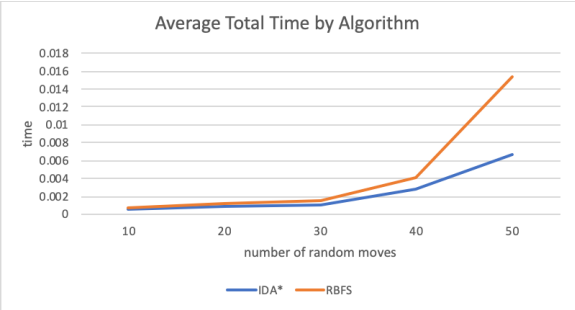


Figure 8

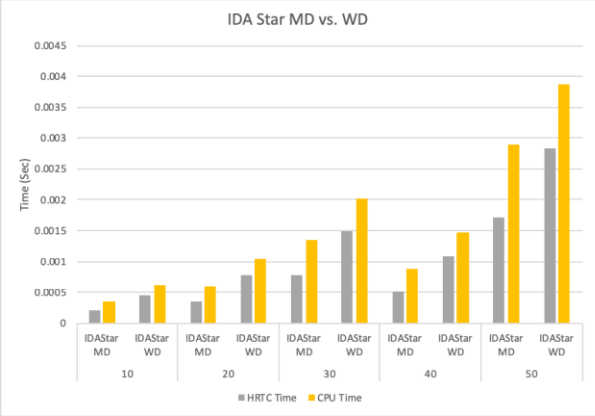


Figure 9

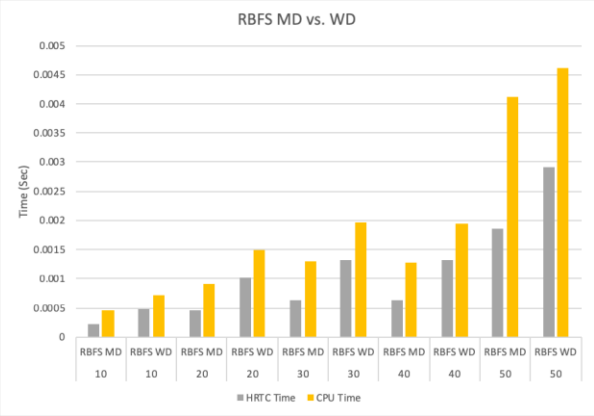


Figure 10

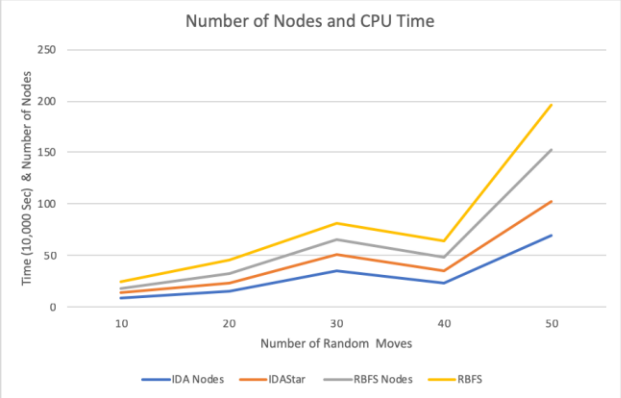


Figure 11

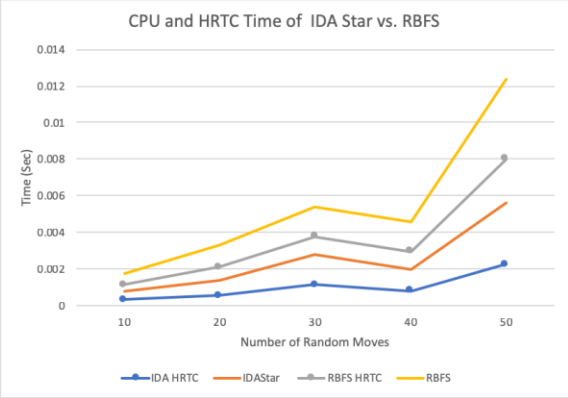


Figure 12

V. Discussion & Conclusion

1. Is there a clear preference ordering among the heuristics you tested considering the number of nodes searched and the total CPU time taken to solve the problems for the two algorithms?
   We set the program to see the performance of the walking distance heuristic function. We expected that it would have shown not as good performance as the Manhattan distance heuristic function in time spending. Also, the walking distance showed less nodes expansion as we expected.

2. Can a small sacrifice in optimality give a large reduction in the number of nodes expanded? What about CPU time?
   According to the Figure 3 and 4, the walking distance expanded less nodes than the Manhattan distance, but Figure 6 shows that the walking distance consumes more heuristic time relatively. Also, Figure 9 and 10 shows the walking distance consumes more time than the Manhattan distance. Figure 11 and Figure 12 supports this observation.

3. Is the time spent on evaluating the heuristic a significant fraction of the total problem-solving time for any heuristic and algorithm you tested?
   As shown in Figure 6, 9, 10, 11, 12, the time spending of the heuristic evaluation takes a significant fraction of the total problem-solving time.

4. How did you come up with your heuristic evaluation function?

In the text book, it says that RBFS is an optimal algorithm if the heuristic function h(n) is admissible. Thus, we started from MD which is the most well-known admissible heuristic function for the 15 puzzle problem. Then we paid attention to “3.6.3 generating admissible heuristics from subproblems: Pattern databases”. The idea of pattern databases is to store exact solution costs for every possible subproblem instance. From it, we searched other studies related it, and we found WD developed by Ken'ichiro Takahashi. In respect to the idea of disjoint pattern databases, WD divides its costs in vertical and horizontal moves and it counts them separately. However, the sum of the two costs is still a lower bound on the cost of solving the entire problem. Therefore, it is much efficient than MD.

5. How do the two algorithms compare in the amount of search involved and the cpu-time?
As you can see in Figure 11, the number of nodes searched and its CPU-time can be an effective measurement for the algorithm comparison since there is a positive relation between those two factors.

Reference

[1] *15puzzle Optimal solver*. [Online]. Available: http://www.ic-net.or.jp/home/takaken/e/15pz/index.html.