A Comparison of Search Algorithms on the Tower of Hanoi Problem

Wen Chuan Lee
College of Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN
Email: lee@leewc.com - leex7095@umn.edu

***Abstract*—This paper compares the performance of 4 major search algorithms, the Breadth First Search, Bidirectional Breadth First Search, Depth First Search as well as Iterative Depth First Search on the Tower of Hanoi. The comparison is done emphrically based on concrete experiments and metrics such as runtime as well as nodes traversed to find a solution. Depth first search is the fastest and most memory efficient algorithm in solving the Tower of Hanoi, whereas Bidirectional Breadth First Search is the second fastest. The Tower of Hanoi problem was selected as the problem to be researched as the puzzle has multiple states and rules to be followed, with large search spaces as the number of disks are added.**

I. Introduction

A. The Field of Search

Since the late 1950s, search has been considered to be influential to the field of Artificial Intelligence. Early AI programs had an algorithm that would search through possible states to arrive at an eventual solution, some even being able to backtrack when the algorithm reached a dead end. This paradigm was considered by McCorduck and others to be ”reasoning by search” [2]. Search has also been an important to the field of computer science as it allows information to be gathered from data in a manner that would be nowhere fast nor accurate enough if done by human beings. There has been much emphasis on this field as computers aim to simplify our increasingly information-oriented lives. A good search algorithm not only provides the right results, but also accurate, precise and as efficient and possible. With the great increase in computing power, search algorithms are even used to solve puzzles and derive logic based on a set of rules, this can be demonstrated with the Towers of Hanoi.

B. The Tower of Hanoi Problem

The tower of Hanoi is a mathematical and logical toy problem that has exactly 3 pegs with a varying number of disks. The goal state of the problem is to have all disks moved from one peg to another peg with regards to specific rules.The complexity of the problem increases exponentially with the increased number of disks. This is because with n number of disks, the *minimum* number of moves required to solve the Towers of Hanoi puzzle is $2^n - 1$. [3]

This mathematical problem has 3 simple rules:

1) Only one disk can be moved at a time.
2) A larger disk cannot be moved on top of a smaller disk.
3) Only the top most disk of each peg can be moved to another peg.

In this paper I attempt to make an experimental analysis on the performance and memory of complexity of the 4 search algorithms on the Tower of Hanoi problem. The Tower of Hanoi problem was selected as the problem to be researched as the puzzle has multiple states and rules to be followed, with large search spaces as the number of disks are added. 4 well known algorithms are chosen in an attempt to find the fastest algorithm that provides solution path set. The problem will be represented in Python along with the search algorithms, after which problems of varying difficulty are provided to the algorithm by increasing the number of disks in the problem. Finally I will analyze and discuss the memory used and the time complexity of each algorithm to solve each of the problem, and provide some insight on the most performant algorithm based on time and memory that solves the Tower of Hanoi problem. The data collected will display a major difference of the runtime between algorithms to provide clarity on how the performance of these algorithms work.

The paper is organized as follows, the next section discusses the algorithms on a theoretical level, in which complexity is discussed, followed by a discussion on the experimental set up, and then presentation of the raw data, before an analysis and discussion based on the data and finally a conclusion that explores possible further research.

C. The Algorithms

One of the more well-known algorithms of search is the breadth-first search (BFS). As an uninformed search algorithm, BFS expands the root node and then the successors of the root node, followed by their successors and so on [4]. All nodes at a specific depth are searched before the following depth are expanded. An advantage of the BFS algorithm is that it is complete, in which breadth first search will eventually find a solution that exists in the state space, if there is one. It is also considered to be the optimal if and only if the path cost is a nondecreasing function of the depth of the node [4].

Bidirectional breadth-first search (bidirectional BFS) is a modified version of BFS in which two simultaneous BFS searches are started, one from the initial state and one from the final state. However, the key difference is that instead of a goal state test at every level, this is replaced by a check to see if the two frontiers have an intersecting node. If there is one, then the solution path is found. The distinction on this is the assumption that the sum of 2 simultaneous searches will be less than the complexity of a single search from the initial state.

Depth First Search (DFS) is another uninformed search algorithm that traverses down a tree vertically unlike the horizontal breadth first search. The algorithm will arrive at a solution faster if the path that is being traversed down is the solution path (i.e, contains the goal state). Upon finding an no other child nodes to traverse down to, the algorithm will then go back to the previous node and explore another node at the same level. There are 2 implementations of DFS, in which one is iterative, and the other recursive. This paper explores DFS using the iterative implementation.

Iterative Deepening Depth First Search (IDDFS) is a modification of DFS, but instead of traversing down the search tree until no possible child nodes are generated before backtracking, IDDFS has a maximum depth of traversal that increases by 1 and then restarting the search again from the top. This would avoid the algorithm going endlessly going down the search tree and yield a solution state faster if the state is contained in the middle or the right side of the search tree. Despite being a depth first search, this depth limited search visits nodes in a breadth first fashion cumulatively.

II. Hypothesis and Theoretical Analysis

Bidirectional BFS will have a better performance and time complexity than BFS itself. As BFS has to explore each node at every level before moving on to the next, the memory and time required will be much longer while Bidirectional BFS will reduce the time of exploring each node at every level by running two simultaneous searches, that is one from the problem state and one from the goal state.

Both BFS and bidirectional BFS are both complete and optimal if the step costs are all identical (in the case of the Towers of Hanoi problem they are). As bidirectional BFS uses BFS in both simultaneous searches, the algorithm shares the complete and optimal properties of BFS.

The difference however, is the theoretical bounds on time and space. For BFS, the time and space complexity is

$$O(b^d) \tag{1}$$

where *b* is the branching factor of the tree and *d* is the depth of the tree. Specifically for the Towers of Hanoi problem to be represented, the branching factor will be 1 or 2 as only the top most pegs can be moved at each time, as moving back to a previous state is considered useless and does not cause the algorithm to Therefore, the depth of the tree for this solution will be:

$$d = 2^n - 1 \tag{2}$$

For Bidirectional BFS, the time and space complexity will be

$$O(b^{d/2}) \tag{3}$$

By having 2 simultaneous searches, bidirectional BFS attempts to run in less than the time and space complexity of BFS. If the following is true then bidirectional BFS will be much faster than running a single BFS algorithm only. [4]

$$O(b^{d/2}) + O(b^{d/2}) \leq O(b^d) \tag{4}$$

Depth First Search runs with

$$O(b^d) \tag{5}$$

for space and time complexity where d is the maximum depth limit traversed. Whereas Iterative Deepening Depth First Search (IDDFS) will have a time complexity of

$$O(b^d) \tag{6}$$

as well, but only a space complexity of

$$O(bd) \tag{7}$$

as less memory is used for node storage on limited depths. Although the states are visited multiple times which may seem as a wasteful action, the behavior is not too costly as most of the nodes are at a lower level, making the multiple visits at the higher levels have less impact on performance.

III. Experimental Set-Up

The set up for testing both algorithms will be done by varying the number of disks, *n*. This increases the size of the graph as well as the depth of the graph of states. A state graph (that also contains illegal actions) has a maximum of 3^n for a 3-peg Tower of Hanoi problem. The solution depth will have a size of $2^n - 1$ as previously mentioned.

The Tower of Hanoi problem is abstracted by representing each disk as a list, and the entire set up that consists of 3 pegs into another list, yield a state that is represented as a list of lists of integers. A sentinel value, which is a value to indicate the end of data (commonly used in searching [1]) is used to represent that no more disks are on a particular peg. The sentinel values have a value of n+1 where n is the total number of disks this allows for simplified implementation and faster computations.

The initial state will be when all disks are on the first peg, and the goal state is when all disks are placed in order from largest to smallest on the third peg. Pruning was built into the problem framework by ensuring that possible states generated conformed to the rules in the Tower of Hanoi, ensuring that illegal states (larger disks on top of smaller disks, moving multiple disks at a time without a specific order) are not generated, to reduce the size of the search space and avoid unneccessary computation. A dictionary of visited states are also kept in memory to ensure algorithms are not caught in a loop revisiting the same states infinitely.

After running all 4 algorithms with an increasing number of disks and recording the number of nodes traversed to find a solution as well as the total computation time to find the solution, a graph of Number of Nodes Traversed to the Number of Disks will be generated, where both the algorithms can be compared. A graph of Computation time to the Number of Disks will also be generated for a concrete analysis of algorithms.

The experiments where then executed on a Intel Core i7-4790 machine running at 3.60GHz with 32GB of RAM. The computer used was a lab computer that had SSH access as well as the ability to keep background processes executing with the screen utility. The experiment ran for over 11 hours taking up over 6 GB of RAM. The experiment was halted at 11 disks because of the exponential increase in runtime.

IV. Results

A. *Maximum Size of Frontier by Number of Disks*

Number of Disks	Depth First Search	Bidirectional BFS	Breadth First Search	Iterative Deepening DFS
2	4	3	3	5
3	9	8	14	10
4	28	21	36	29
5	81	79	156	82
6	244	254	528	245
7	729	1079	2164	730
8	2188	4247	8256	2189
9	6561	16599	33236	6562
10	19684	66135	131328	19685
11	59049	262999	526164	59050

C. *Number of Nodes Traversed to Number of Disks*

Number of Disks	Depth First Search	Bidirectional BFS	Breadth First Search	Iterative Deepening DFS
2	6	10	5	28
3	9	22	36	55
4	49	86	184	534
5	81	486	1194	3403
6	428	2506	6738	30992
7	729	15182	43288	266815
8	3831	91516	257792	2404618
9	6561	542326	1606086	21533203
10	34450	3286532	9671326	193828356
11	59049	19768542	59096340	1743480775

D. *Graph of Traversed Nodes to Number of Disks*

(to be added later, composite graphs that will illustrate the major differences between algorithms)

E. *Graph of Runtime to Number of Disks*

(to be added later)

F. *Graph of Nodes Traversed to Number of Disks*

(to be added later)

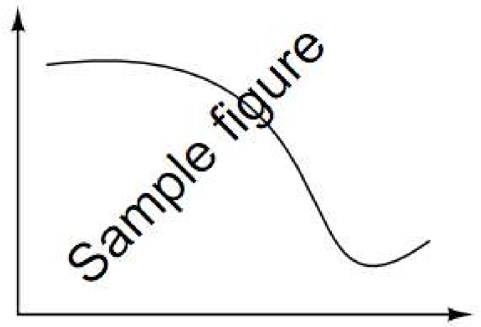

Fig. 1. Graph of Runtime to Number of Disks

V. Results Discussion

From the results it can be seen that Depth First Search is the fastest algorithm to obtain a goal state that is the solution. This is expected as the solutions are always deeper down the tree. Bidirectional Breadth First Search is the second fastest algorithm, although the algorithm takes an exponentially larger amount of time to solve larger number of disks. Although bidirectional BFS still yields a solution by over a factor of 2 faster than a single instance of the BFS. The results generated for number of nodes traversed, runtime and maximum size of frontier all confirm this. Bidirectional BFS manages to reduce the depth traversed by separating the problem into 2 instances (from initial state and goal state) as there are lesser number of nodes at shallow levels. This greatly reduces the number of states to visit before obtaining a solution. Iterative Deepening DFS however yielded the worst performance, as nodes are constantly revisited as the search is restarted, and the Tower of Hanoi problem only has solutions stored in the deepest level of the tree. This caused the algorithm to perform badly in terms of memory and runtime.

There are negligible differences between the algorithms on small number of disks, but as the Tower of Hanoi problem increases in complexity exponentially as the number of disks increases, the difference is evident. On a problem with 11 disks, DFS only took 1.54 seconds to find a solution, while Bidirectional BFS took 900.4 seconds (15 minutes), and BFS took 3774.5 seconds (1.04 hours), and Iterative Deepening DFS took 84143.70 seconds (23.4 hours) to find the solution.

The results prove that the Iterative Deepening DFS algorithm do not work well for problems that have solution states at the leaves of the search tree. DFS is the fastest algorithm to yield a solution provided that there is a solution and the problem is not intractable.

VI. Conclusion and Future Work

This paper has shown the performance of 4 search algorithms, BFS, DFS, Bidirectional BFS and, Iterative Deepening DFS on the Tower of Hanoi problem, based on metrics of runtime, nodes visited and maximum size of frontier. The algorithms worked on a search space that is continually pruned based on the rules of the problem. This proves that if a problem is known to have a solution, DFS will be the fastest algorithm

B. *Runtime (seconds) of each algorithm to Number of Disks*

Number of Disks	Depth First Search	Bidirectional BFS	Breadth First Search	Iterative Deepening DFS
2	0.000156392	0.000212669	0.000096286	0.000472911
3	0.000204044	0.000455272	0.000719421	0.000987642
4	0.000950633	0.001794590	0.003294816	0.009155311
5	0.002016366	0.007936978	0.017867410	0.055075039
6	0.006868233	0.041572044	0.106164721	0.537936986
7	0.012976039	0.265300220	0.740262595	4.933634893
8	0.067659920	2.026069019	6.571621341	55.26996347
9	0.128477633	16.297773958	52.6292188720	744.907908745
10	0.726265783	118.008149897	389.933532058	8319.07307241
11	1.543463721	900.394615177	3774.456782876	84143.691529322

to yield a solution, despite the possibility of not yielding the most optimal solution.

Further optimizations can be made to the DFS algorithm by introducing even more advanced pruning techniques, or to short circuit goal checking until the algorithm is at a certain depth.

References

[1] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press.

[2] Pamela McCorduck. *Machines who think, 2nd ed.* A K Peters/CRC Press, 2004.

[3] Miodrag Petkovi. *Famous Puzzles of Great Mathematicians*. AMS Bookstore, 2009.

[4] Peter Norvig Stuart Russell. *Artificial Intelligence, A Modern Approach, Third Edition*. Pearson, 2009.