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 empirically based on concrete experiments and metrics of maximum frontier size, the total time taken to solve the towers of hanoi problem with varying number of disks, as well as nodes traversed to find a solution. In the experiment, 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. The paper also discusses the optimality and the speed of each search algorithm.**

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” [3]. 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. In a a age in which more and more emphasis is being put on transforming data into information, search is a critical field in which we can process large amounts of data to obtain the information needed. Today, where significant developments are being made in machine learning and data mining thanks to the huge increase in computing power, search algorithms are more important in the field of computing than they ever were. Search algorithms were used to solve puzzles and derive logic based on a set of rules in the past and today, and this can be demonstrated with the Towers of Hanoi.

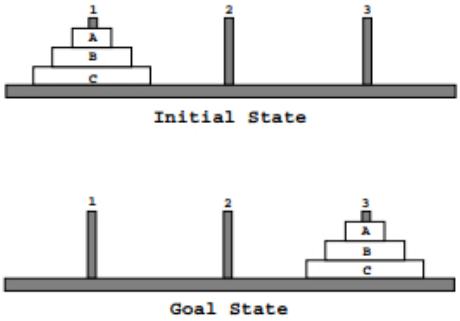

Fig. 1. Initial State and Goal State of the Towers of Hanoi Puzzle

B. The Towers of Hanoi Problem

The Towers 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 $2^n - 1$. [4]

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.

From these 3 simple rules, it can be seen that with each additional disk, steps to solve the problem increase exponentially. Hence while smaller number of disks, for example $n = 3$ would only require a minimum of 7 steps to solve the puzzle, at $n = 12$ puzzles, a minimum of 4095 steps are required. This non-linear increase causes the problem's state space to become very large, making it an ideal puzzle to compare each search algorithm in terms of space and time complexity as well as optimality. An illustration of the problem with initial and goal states is shown in *Figure 1* [1].

In this paper I attempt to make an experimental analysis on the performance and memory of complexity of the 4 search algorithms on the Towers of Hanoi problem. The Towers 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. 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 instance, and provide some insight on the most performant algorithm based on time and memory use 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 will discuss the algorithms on a theoretical level, in which complexity is discussed. This is followed by a discussion on the experimental set up, and then presentation of the raw data. An analysis and discussion based on the data is then provided by inferring the data collected with relation to the nature of the algorithms, and finally form a conclusion as well as explore 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 [5]. 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 the branching factor is finite. The solution found by BFS is also optimal if and only if the path cost is a nondecreasing function of the depth of the node [5].

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. The algorithm is able to reduce the number of nodes traversed in total as for each increasing depth of the search tree traversed, there are exponentially more nodes and states to visit. Having a search tree generated from the initial state and the goal state it is believed that a solution (whereby one BFS algorithm has visited a state previously visited by the other BFS algorithm) can be found earlier in the shallower parts of the search tree.

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 that there are 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, although it should be noted that the difference between the recursive and iterative implementations are negligible as the interpreter will perform the optimizations to the function calls and stack frames automatically.

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 at each iteration from the top. The algorithm terminates when a solution is found. By restarting the search and limiting the maximum depth, 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

The best algorithm in terms of performance and time complexity is Depth First Search. While Bidirectional BFS will have a better performance and runtime than BFS itself, DFS triumphs both algorithms in terms of finding a solution state for the Towers of Hanoi. Whereas 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 originating from the problem state and one originating from the goal state. Finally, runtime of IDDFS will be in between the runtime of Bidirectional BFS and BFS as the search will continuously restart. This property of this algorithm is believed to make IDDFS unsuitable for finding solutions that are always located in the leaves of a search tree.

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. IDDFS is also complete and optimal given the nature of the algorithm's traversal. DFS is not will not yeild an optimal solution and is not complete.

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, and moving back to a previous state is considered useless and does not cause the algorithm to arrive closer to the goal state. 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. [5]

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

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 search space as well as the depth of possible states to search. A state graph (that also contains illegal actions) has a maximum of 3^n nodes for a 3-peg Tower of Hanoi problem. The solution depth will have a size of $2^n - 1$ as previously mentioned.

The experiments where 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 30 hours taking up over 6 GB of RAM. The experiment was halted at 11 disks because of the exponential increase in runtime. Upon completion of every Tower of Hanoi problem, statistics are then saved in comma-separated value (CSV) files.

The Tower of Hanoi problem is abstracted by representing each peg 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 [2]) 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 unnecessary computation by generating and traversing down a branch of states that will not arrive to a legal solution state. A dictionary of visited states are also kept in memory to ensure algorithms are not caught in a loop revisiting the same states infinitely. The key of the dictionary is a string representation of the state, and the value is either an integer of 1, or the actual reference to the node, this allows us to fetch the node for bidirectional BFS when a solution is found.

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 runtime to the Number of Disks will also be generated for a concrete analysis of algorithms.

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

V. Results Discussion

From the results (figures are presented on the next page) 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

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 (˜12.4 minutes)
10	0.726265783	118.008149897 (˜2 minutes)	389.933532058 (˜6.5 minutes)	8319.07307241 (˜138 minutes)
11	1.543463721	900.394615177 (˜15 minutes)	3774.456782876 (˜1 hour)	84143.691529322 (˜23.3 hours)

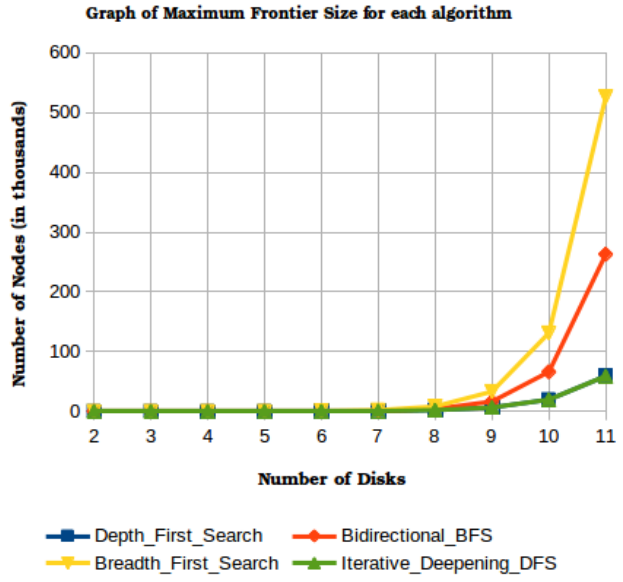

Fig. 2. Graph of Maximum Frontier Size to Number of Disks

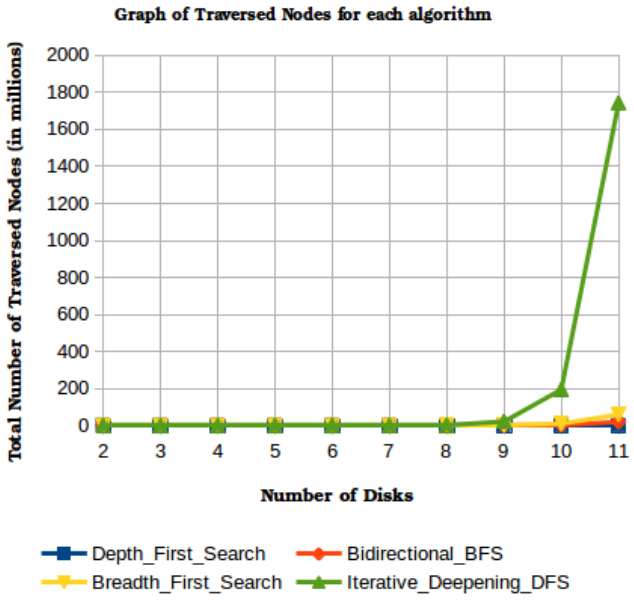

Fig. 4. Graph of Number of Traversed Nodes to Number of Disks

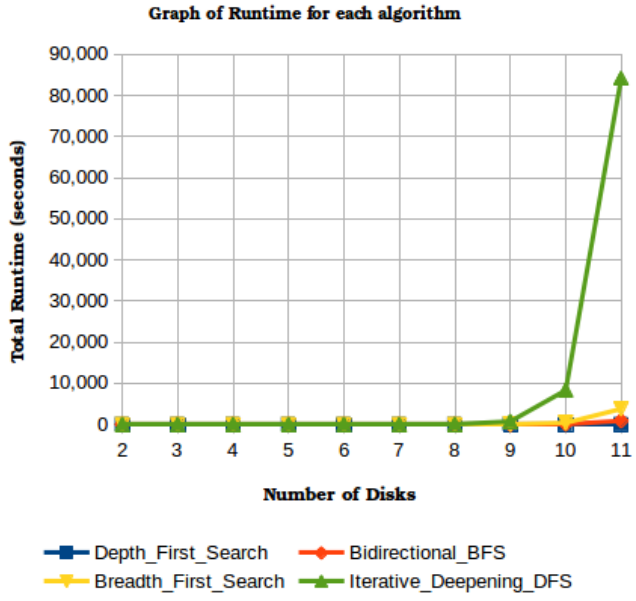

Fig. 3. Graph of Runtime (seconds) to Number of Disks

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.

It can be seen that the number of traversed nodes for each algorithm remained fairly similar in the graph that displays the number of nodes traversed to the number of disks *Figure 4*. This is actually due to the Iterative Deepening DFS traversing a total of 1743480775 nodes. This is a huge difference compared to only 59049 nodes for DFS, 19768542 for Bidirectional BFS, and 59096340 for BFS, over 29526

times larger than DFS. This is expected due to the nature of Iterative Deepening DFS which repeats the search upon reaching the depth limit. This complete restart everytime the algorithm has reached a max depth is computationally expensive and wasteful as the nodes are revisited from the top. For problems such as the Tower of Hanoi, where the solution is only found at a specific tree depth, Iterative Deepening DFS is not an ideal algorithm given how the algorithm has to restart from the beginning constantly before arriving at the leaves of the search tree which will contain all possible solution nodes.

In addition to that, a table of runtime of each algorithm to number of disks *(B)* is provided to further illustrate the major difference in run times, as the scales on the graph figure *Figure 2* are affected by Iterative Deepening DFS on 11 disks. All the algorithms had very similar runtimes for the Tower of Hanoi problem up to 6 disks. However Iterative Deepening DFS took significantly longer runtimes when the number of disks were 7. The run time shot up exponentially from 0.5 seconds to 4.9 seconds. The algorithm that found a solution in the least amount of time was the Depth First Search for tests from 2 to 11 disks. At 11 disks, DFS only required 1.5 seconds, while the slowest was Iterative Deepening DFS, which took over 23 hours. However, as DFS may not find the most optimal solution, bidirectional BFS was the second fastest algorithm that finds an optimal solution.

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.

Table A that provides the data for the maximum size of the frontier by the number of disks is used as an empirical representation of the frontier size throughout the entire algorithm runtime. As all the algorithms require a method of checking if the state has been visited before, a dictionary that keeps the references of visited nodes are maintained, which results in using the maximum size of the frontier as an inaccurate representation of the memory used. However it does illustrate how the algorithm requires a large queue to keep track of the number of nodes to be visited later. As Breadth First Search traverses the search space from left to right, both BFS and Bidirectional BFS have the largest search frontiers.

From the results, we are able to 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. However, it should be noted that solutions yielded by DFS might not be optimal, that is, do not solve the problem in the shortest and most efficient number of steps. On the other hand, IDDFS, Bidirectional BFS and BFS all yield a complete and optimal solution, at the cost of significantly longer runtimes for the Tower of Hanoi problem.

Regarding the hypotheses set forth in the beginning of this paper, it is true that the best algorithm in terms of performance and time complexity is the Depth First Search algorithm. As it quickly traverses down the search tree before returning to shallower depths to visit other solution branches.

However, the previous hypothesis that the runtime of Iterative Deepening DFS will be in between the runtime of Bidirectional BFS and BFS since the search will continually restart is false. Although this is the case for instances of the Towers of Hanoi algorithm with small state spaces, it can been seen from the results collected that Iterative Deepening DFS takes a *much* longer time to complete and find an optimal solution for the Towers of Hanoi since the solution state is located deep inside the search tree.

It is also interesting to observe that the maximum frontier for Bidirectional BFS and Breadth First Search has such a large difference on instances with more disks (i.e more complex and large state space). On 11 disks, Bidirectional BFS had approximately half the number of nodes compared to BFS at a maximum frontier size. This huge difference would equate to the faster runtime of Bidirectional BFS as lesser nodes need to be traversed down the search tree to arrive at a solution. In addition to that, the algorithm is also complete and optimal (always yields an optimal solution) as opposed to Depth First Searches.

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 at the leaves of a tree, DFS will be the fastest algorithm to yield a solution, despite the possibility of not yielding the most optimal solution.

Should a most optimal solution be desired on a problem that has a definite initial state and goal state, Bidirectional BFS would be the second most performant algorithm, again in terms of memory usage and runtime. By having 2 searches originating from the initial state and the goal state meet halfway, the size of the tree can be reduced as the algorithm is able to meet halfway down the tree as every level down the tree there are exponentially more disks, it is wise to reduce the depth traversed.

It has also been shown that the Iterative Deepening DFS, a Depth-first search algorithm that is also complete and optimal due to the depth-limiting characteristic, is not an ideal candidate algorithm for solving problems with solutions found only in the leaves of a search tree, such as the Tower of Hanoi problem, as there are too many restarts that begin again from the tree. Iterative Deepening DFS however, would be an ideal algorithm if the solution can be found in varying depths of a search tree.

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. Concurrency and parallelism could also be introduced into this problem, as currently the Python implementations of the Tower of Hanoi problem conveniently ignores multi-threading to reduce the complexity of the program (additionally, CPython currently still requires careful handling of data structures in a multithreaded environment). Each of the search algorithms could also definitely be sped up with the use of multiple threads, especially the Bidirectional BFS, as it has 2 instances of search running which could further reduce the run time of solving the problem at the cost of complexity given the additional threads and need to create mutex locks around data structures that are not thread-safe.

VII. Acknowledgments

This paper would not have been possible without the knowledge gained from the Intro to Artificial Intelligence class of Fall 2015, as well as Professor Amy Larson for providing support and knowledge on search algorithms, and then later exploring possible methods of representing the Towers of Hanoi problem in code. Additionally, I am grateful to the course staff for being available during office hours, as well as providing timely responses to questions posted on the class forum. This paper is the result of other even deeper and detailed research on the subject matter by scholars made available on the Internet and in literature, as mentioned in the references, along with peer reviews by fellow classmates.

References

[1] Craig A. Knoblock. Abstracting the tower of hanoi.

[2] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, 2004.

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

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

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