Investigating the Efficiency of Optimizing Search Algorithms in Determining Best Moves within a Search Tree

To what extent do optimizations on a negamax algorithm such as alpha-beta pruning increase its efficiency when searching for best moves in a game of Connect 4?

A Computer Science Extended Essay

3694 Words

Table of Contents

1. Introduction

With continuous rapid development in technology, society has seen increasing consumption in digital media, including but not limited to social media, video games, online streaming platforms, and more. Within the video game community there are a multitude of genres, such as Role-Playing Games (RPGs), Massive Multiplayer Online games (MMOs), First Person Shooter games (FPS), and more. However, a common trend that applies to essentially all genres is the usage of computer-substituted players, known better by the term “bots.”

As the growth of the gaming market grows increasingly more competitive, video game producers look towards new technologies as well as improving existing technologies to provide the best user experiences possible through superior graphics, better computer bots, virtual reality, etc. Especially with computer graphics, the number of calculations needed can rise multitudes greater than previous years, taking a greater toll on the client’s computer and reducing the quality of gameplay. Although better hardware such as better CPUs, GPUs, and RAM may be able to offset the performance deficit, it is nevertheless imperative that the algorithms used are as optimal as possible to provide the fastest performance. In some programs, the performance of algorithms can make or break the program altogether.

Furthermore, as performance becomes a major determinant in the quality of video games, optimizations on algorithms will become necessary to provide the fastest performance possible. As such, this paper investigates the extent to which optimizations on an algorithm designed to find the best move in response to another player in Connect 4 can increase its efficiency.

Although this paper focuses on video games, the research done in this paper could potentially prove useful in other fields of technology as the underlying concept and data structures used ultimately boil down to finding the best choice within many possibilities. Furthermore, a deeper understanding of search algorithms could drastically improve the performance of similar algorithms outside of just the game development industry. For example, a map program which gives directions to the user to a location they want to go. In the context of someone’s house, the range of possibilities can be relatively miniscule, but in the context of the city, or even the world, the possibilities are too great with a straightforward approach to the problem. This is where optimizations can vastly reduce calculation times to provide directions in a reasonable amount of time instead of days, or months.

In this paper, a negamax algorithm was built for the purpose of “solving” a Connect 4 game and used as a base case in comparison to further optimizations of such. The algorithms are coded in C++ and will be subject to multiple testcases of varying difficulty, where the runtime of each algorithm was recorded after each testcase. The mathematical and empirical implications for the algorithms and their results were analyzed.

2. Background Information

Data Structure:

A data structure is a general term for some implementation that allows for the search and retrieval of some element within its content in an efficient manner (Britannica, 2017). An example could be a planner app, which can store a person's to-do items, and retrieve them to show the user what they must do that day. Data structures can be very general, as with dictionaries, or they can be built to solve intricate and specialized problems which can't be solved with general data structures otherwise. In this investigation, a critical data structure that stores the results of previous searches.

Algorithm (in computer science):

Algorithms are the foundation to solving problems in computer science. In general, an algorithm is a procedure to solve a specific problem using a computer (Belford & Tucker, 2020). There are no strict guidelines on what an algorithm can/should do; it could be adding a set of numbers together, sorting a set of numbers, finding the shortest path from one point to another, etc. Although it may be easy to think of algorithms as just code in a computer, the nuances between algorithms as well as the environment to which an algorithm will be executed in are crucial for the algorithm to perform efficiently not only in the local scope of the program, but to make sure it does not hinder the performance of the app. As such, time complexity is a huge factor in determining how to define an algorithm.

Big O Notation / Landau's Symbol (in computer science):

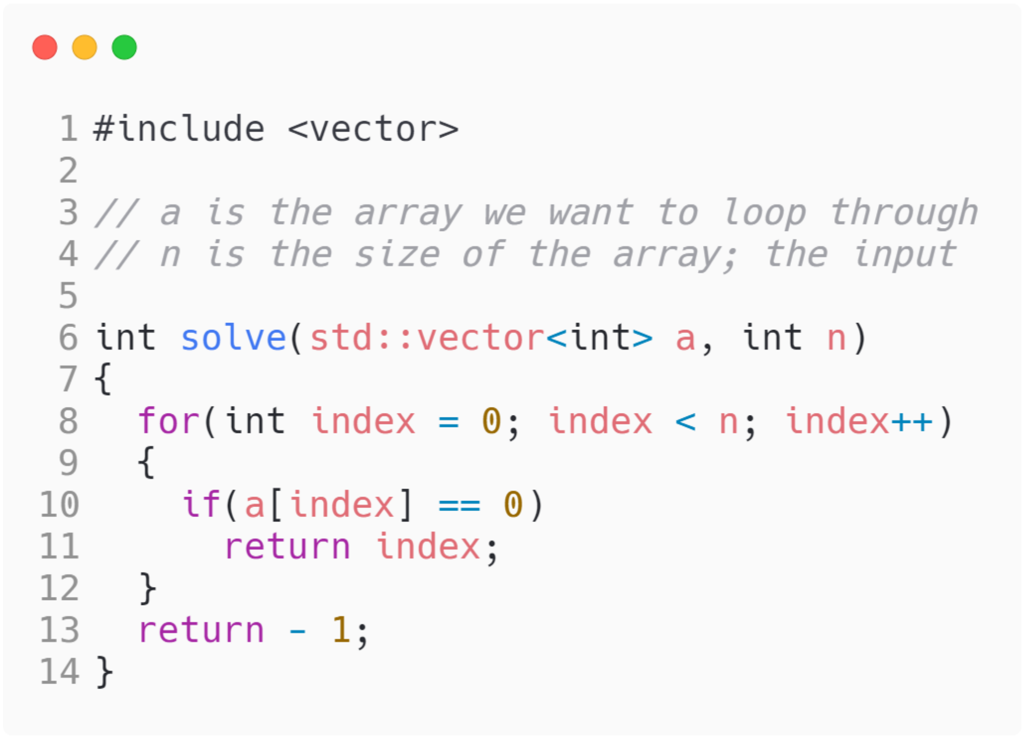
In computer science, big O notation is a way to denote the scaling of time and space needed for an algorithm to solve a problem as the input size for the algorithm increases (Mohr, 2007). Big O notation takes the form of $O(n)$, where n is the input to the algorithm. Note that algorithm may have multiple inputs – in which case the parameters of O could contain an arbitrary number of arbitrarily named variables representing one input. For example, the time complexity of a Depth First Search algorithm could be $O(V + E)$, where V represents the number of vertices in the graph, and E represents the number of edges in the graph.

Computational Complexity:

In computer science, performance is a crucial aspect of the field. The goal for a program is for it to perform as optimally as possible. As such, there are different aspects of complexity which an algorithm is judged by. In general, the complexity of an algorithm refers to the extent to which a computer's resources is expended to run it. The resources that are often into account are time and space (Belford & Tucker, 2020). For each resource exists its own complexity, with its own methods to define such. Furthermore, depending on the environment the algorithm is run in, the problem it is attempting to solve, or other factors, the amount of resources that should be allocated may fluctuate.

Time Complexity:

Time complexity is a subset of computational complexity that deals with determining the relative amount of time that an algorithm will take to run to completion. Like computational complexity, big O notation is typically used to represent time complexity. For all intents of this explanation, it will be assumed that the algorithm has one input, n. An important thing to note is that constant factors and coefficients are disregarded in big O notation. This is because since time complexity is the sum of rudimentary operations, such as adding, subtracting, modulo, etc. which are limited by the hardware capability of one's computer, giving an unreliable estimation of algorithm running time. Furthermore, big O notation is most applicable for estimating worst-case run time. This is because an algorithm may complete before it goes through the entire input, in which case its time complexity would not be representative of the runtime for that case. Consider this case: given an array a of size n that holds the numbers from zero to n exclusive, in a random order, determine the index of the number 0. The simplest algorithm that could be devised would be to loop through the array from the beginning, stopping when we find the element 0. The code for the algorithm is depicted below:



```cpp
#include <vector>

// a is the array we want to loop through
// n is the size of the array; the input

int solve(std::vector<int> a, int n)
{
  for(int index = 0; index < n; index++)
  {
    if(a[index] == 0)
      return index;
  }
  return - 1;
}
```

Figure 1 basic for loop

Since time complexity is the sum of rudimentary operations, we see that we will check an element n times before the algorithm finishes. This means that the algorithm is $O(n)$. However, we know that if the random array generated has 0 as the *first* element, we will only do one operation before returning, not n operations. Therefore, big O notation is best for estimating worst-case runtimes. There are several classifications of time complexity which apply to many common algorithms.

- Constant time, $O(1)$, which means that the input size does not play any role in the runtime of the algorithm, such as a function to return the sum of two numbers.
- Logarithmic time, $O(\log(n))$, meaning that the input halves every iteration, such as a binary search.

- Linear time, $O(n)$, meaning that the algorithm will loop through every element in the input at least once, such as summing all the numbers in an array of size n.
- Linearithmic time, $O\big(n\, log(n)\big)$, such as merge sort.
- Quadratic time, $O(n^2)$, meaning that in an array of n elements, we will loop through each element n times, such as an algorithm to naively find all the pairs in an array.
- Cubic time, $O(n^3)$, such as an algorithm to naively find all pairs of three elements in an array of size n.
- Etc.

Time complexity is incredibly important to understand because for this investigation, the time complexities play a huge role in the efficiency of each algorithm.

Depth First Search (DFS):

In graph or tree data structures, DFS is an algorithm used to traverse nodes. Starting at an arbitrary node A, and an arbitrary target node X, a DFS algorithm will explore all nodes connected to node A once, checking if any nodes in the path are connected to X. If there are, there is a path from node A to X. This algorithm is used in the minimax algorithm, where from the starting node, the minimax algorithm uses DFS to find the terminal nodes, where either the first player wins, the second player wins, or the game ends in a tie. Without DFS, the minimax algorithm could not know what values to minimize or maximize.

Minimax:

Minimax is a decision rule, used to find the best possible move that a player should take. More specifically, it minimizes the damage that the player takes, while maximizing the gains, hence the name minimax (Brilliant.org, n.d.). An arbitrary number, which will be called score, is used to determine the impact each move has on the player. In combinatoric games such as Connect 4, chess, or go, the minimax algorithm leverages the fact that on an even turn (if the enemy goes second), the player always wants to pick the move that gives maximum score, and on an odd turn (the enemy's turn), the player will always want to pick the move that minimizes their score losses (See figure 2).

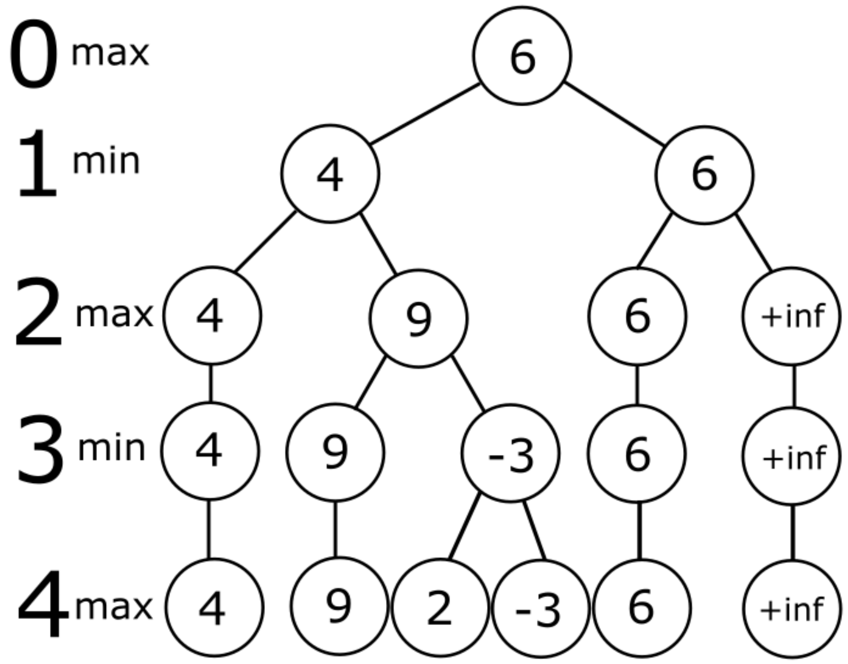


Figure 2 Minimax representation

While in principle the algorithm could work well, a central issue with the minimax algorithm is the amount of nodes it searches. Without optimization, the algorithm will try to search the best move at for every node, at each depth. In small sample sizes, the time complexity is not too bad, but for larger and more complicated games such as Connect 4, the amount of nodes is too much to naively brute force in a reasonable amount of time.

Negamax:

Negamax is a simplification of minimax which relies on the observation that in minimax, the max value is calculated as such: $\max{(a, b)}$. Since in combinatoric games the score of the enemy at any given depth is the opposite of the player, the negamax function can use $-\max{(a, b)}$ to get the enemy score as $\max(a, b) = -\max{(a, b)}$.

Alpha-Beta Pruning:

Alpha-beta pruning is an optimization on the minimax algorithm. Unlike the minimax algorithm, when it finds a branch of the search tree that is unviable, meaning that there is already a move that provides a better score, the branch is removed, or *pruned* from further searches (Russell & Norvig, 2010). This markedly reduces the number of nodes that the algorithm needs to search, and thus drastically reduces the run time as well. Without this optimization, it would take far too long for the minimax / negamax algorithms to solve a Connect 4 game with a difficulty above easy in a decent amount of time. The algorithm itself uses two values: alpha, representing the minimum score that the player can get, and the beta, representing the max score of the enemy player. When the minimax algorithm runs, alpha-beta pruning checks whether for the branch it is currently searching satisfies the condition $beta < alpha$. Since the max score of the enemy cannot be less than the minimum score of the player, the algorithm prunes the branch.

Branching Factor:

The branching factor, when talking about trees, refers to the average number of branches that stem from a given node. The term is commonly used when describing complexity of a tree (branching factor).

Bitboard:

A bitboard is a representation of a board, using bits. A bit can be one of two values: zero or one. In Connect 4, it is easy to represent the game using a bitboard, since there are only two players, and each player can only be one type (a coin). This means that a player can be assigned a bit to be represented on a bitboard. The advantage of using a bitboard is that bit operations tend to faster than ordinary operations, which leads to reduced computation time for the algorithm.

Transposition Table:

A transposition table stores the previous searches of some search by an algorithm (Laramée, 2000). In the case of this investigation's Connect 4 bot, a transposition table is a lookup table for previous searches that the bot's negamax algorithm has already done. Since in Connect 4 there are multiple sequences of moves that will result in the same board formation, the transposition table can be used as an optimization by having the bot first check if the transposition table has already seen a search; if it has, just continue from there instead of wasting time by searching through the nodes again only to reach the same position. If not, then keep going through the search tree to find a unique board formation, after which gets stored in the transposition table.

Connect 4:

The game that the bot used in this paper will focus on is Connect 4. Connect 4 consists of a 7 x 6 board. Each player takes a turn placing their coin in a non-full column, where their coin will fall to the lowest spot available. The game ends when a player gets at least four coins in a row horizontally, vertically, or diagonally. Additionally, the game can end in a tie when no player gets four coins in a row by the time the board is filled up by coins.

3. Methodology

This paper uses empirical data to conclude the findings from the experiment. The code used was written by Pascal Pons, taken from https://github.com/PascalPons/connect4. The bots tested were a negamax implementation with alpha-beta-pruning, a bitboard, and a negamax implementation with alpha-beta pruning, a bitboard and a transposition table with further optimizations on the bitboard and transposition table. An experiment was chosen because using only mathematical and logical deduction, the impacts of each optimization are not clear-cut.

Furthermore, empirical data allows for more in-depth and real-life applicable analysis compared to limited complexity analysis. A limit to this methodology is that hardware and software plays an extremely crucial role in the operation of the algorithms explored in this paper. Discrepancies between computer hardware may lead to drastically different results.

3.1 Controlled Variables

As a point of reference, the relevant specifications for the computer and other static factors which the experiment was conducted on are listed below:

Variable	Details	Description
Computer	**CPU:** Intel® Core™ i7-8700K Processor **RAM:** 2 x 8gb running at 2133mHz CL16 **Storage:** Samsung 970 Evo 500gb NVME SSD	The CPU is the biggest factor in computation speed, followed by RAM, and less impactfully, storage.
Time	**Unit of measurement:** milliseconds **Method:** the <chrono> library from the C++ Standard Library, using the high_resolution_clock and other related functions.	The time it takes for each trial to complete will be measured in microseconds, stored as a 64-bit double primitive, using the <chrono> header from the C++ Standard Library.

3.2 Performance / Efficiency Metric

As the efficiency of an algorithm depends on many factors, especially for different types of algorithms, it is important to recognize which facets of this experiment play a role in answering the prompt at hands. 2 major factors will be considered in determining each solver's efficiency:

1. Runtime: runtime refers to the amount of time it takes for a program to run, measured in milliseconds. The runtime will be measured within the program using the high_resolution_clock from chrono, a header from the standard library. Chrono's high-resolution clock can measure time to the nanoseconds, ensuring a high degree of accuracy and precision. This investigation will compare the average runtime for each set of test data.
2. Node-count: the search algorithm in this experiment relies heavily on **Depth First Search (DFS)** and iterative deepening, which serves to make DFS more efficient by reducing the number of nodes explored. Analyzing the amount of nodes explored is crucial in the analysis of speed for search algorithms.

4. Procedure

The source code was modified to provide ease of experimentation: a local directory containing all the test cases was added, as well as modifications to the source code to make data collection easier. Testing data is provided by Pascal Pons (Pons, n.d.), and consists of 1000 randomly generated games.

1. Compile the code for the weak solver to test using cmake.

2. Run the solver for the set of testcases in question, recording the average time taken per test case and the average number of nodes explored.
3. Repeat steps 1-2 for a strong solver.
4. Repeat steps 1-3 for a solver without a transposition table and a solver without bitboard nor transposition table.

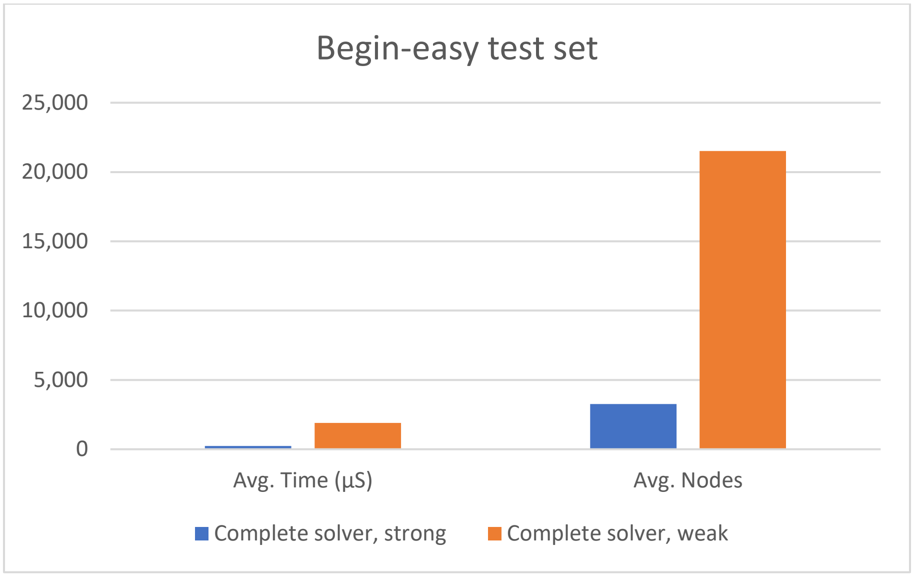
4.1 Data Collection

The table of data below represents the average time per test cases and average number of explored nodes for each solver. For harder test cases, solvers that were unable to complete the test set under 100 hours are labeled as “too inefficient.”

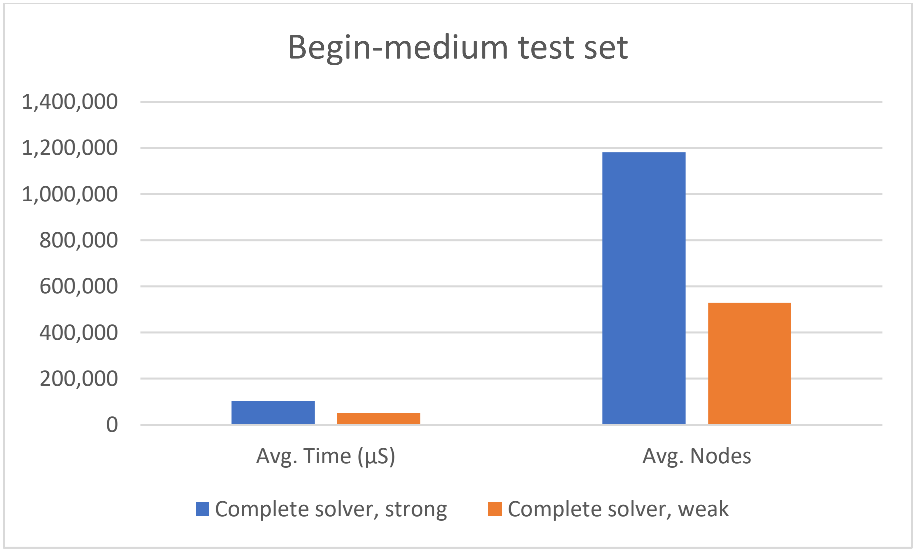
	Test cases					
	Beginning-easy		Beginning-medium		Beginning-hard	
Solver	Avg. Time (µS)	Avg. Nodes	Avg. Time (µS)	Avg. Nodes	Avg. Time (µS)	Avg. Nodes
Complete solver, strong	245.837	3266	104186.721	1181394	5134412.506	56778320
Complete solver, weak	1896.114	21505	52800.452	529421	3204580.482	33313393
No transposition table, strong	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient
No transposition table, weak	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient
No bitboard or t-table, strong	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient
No bitboard or t-table, weak	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient	Too inefficient

	Test cases					
	Middle-easy		Middle-medium		End-easy	
Solver	Avg. Time (µS)	Avg. Nodes	Avg. Time (µS)	Avg. Nodes	Avg. Time (µS)	Avg. Nodes
Complete solver, strong	42.478	449	3979.553	39800	5.715	51
Complete solver, weak	59.226	532	2098.145	20172	4.315	29
No transposition table, strong	30408.961	2081790	600911.126	40396714	3.959	139
No transposition table, weak	14509.815	927942	362813.452	23685434	3.144	107
No bitboard or t-table, strong	4091060.887	54236696	33016615.556	453613671	28.288	283
No bitboard or t-table, weak	3027678.037	41401187	21452970.166	308113945	24.29	222

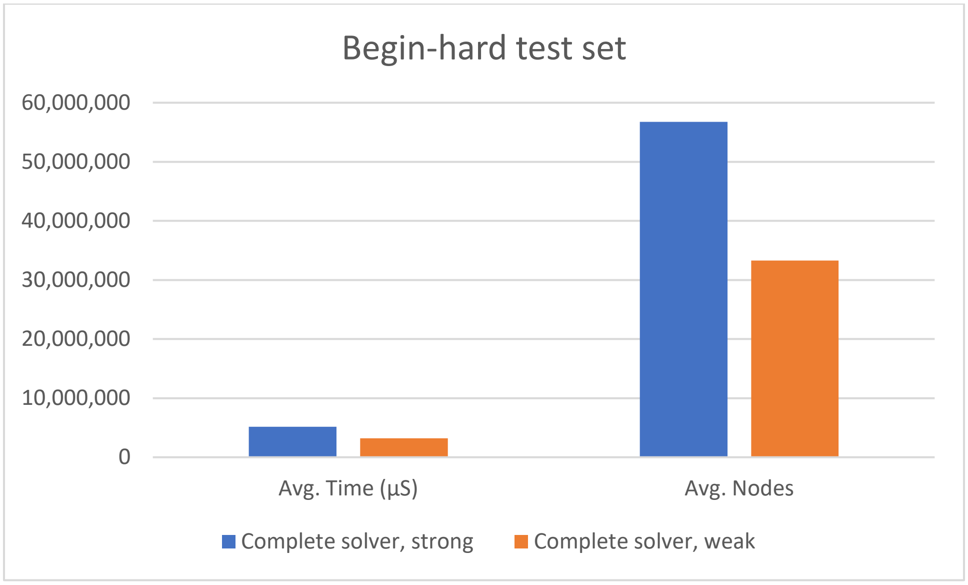
To easily represent the performances of each solver, the tabular data has been visualized for each test set below. Note that solvers that were too inefficient to solve the test cases are not included in the respective graphs.



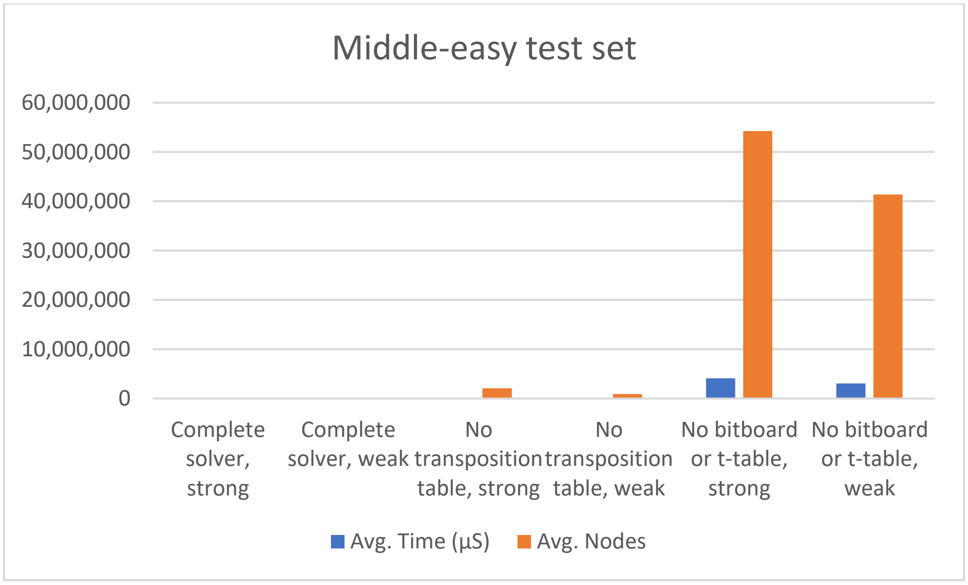
Graph 1 Results of the begin-easy test cases



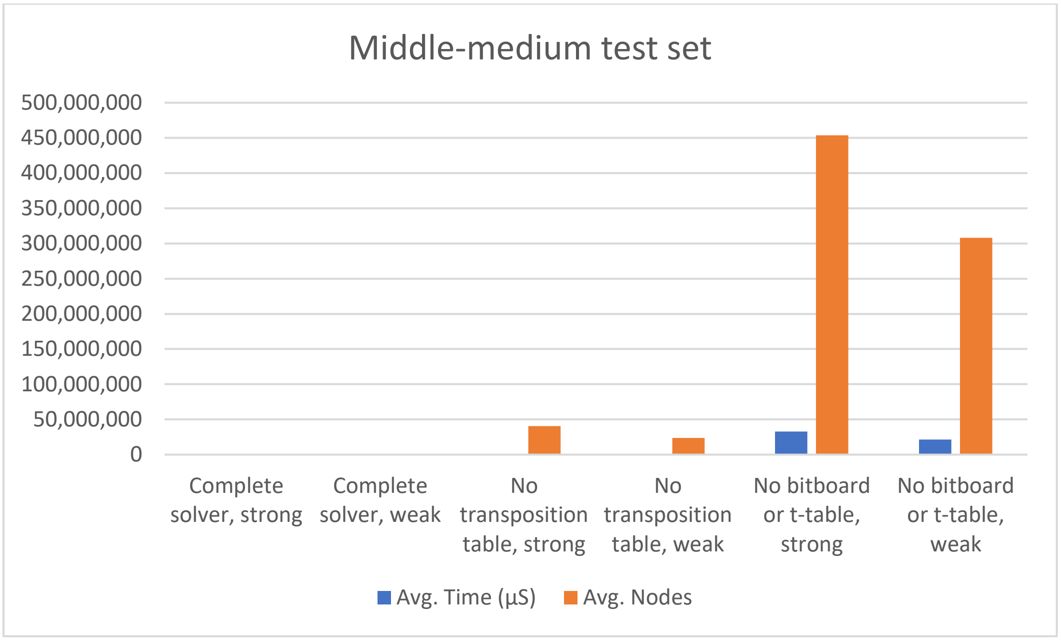
Graph 2 Results of the begin-medium test cases



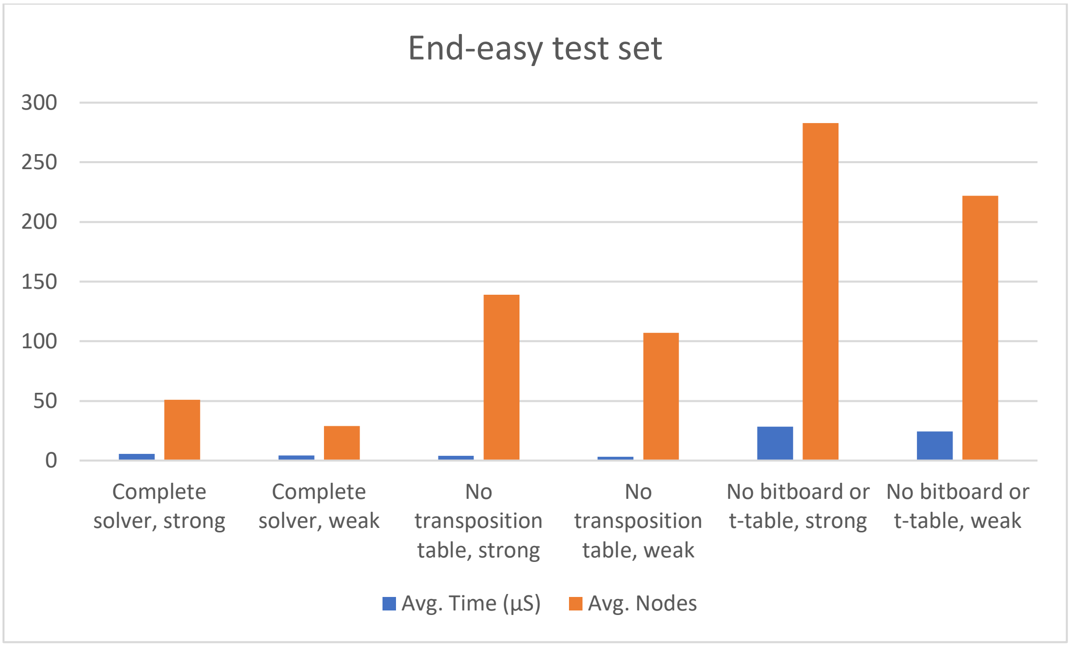
Graph 3 Results of the begin-hard test cases



Graph 4 Results of the middle-easy test cases



Graph 5 Results of the middle-medium test cases



Graph 6 Results of the end-easy test cases

4.2 Data Analysis

Observing the data above, it is evident that each optimization exponentially cuts down on both the average computation time, as well as the average number of explored moves (nodes) per test set. In general, the complete solver was hundreds of times faster than the solver without a transposition table, and thousands of times faster than the solver without a bitboard or transposition table.

4.2.1 Average Time Per Test Case

One may be drawn first to the results of the end-easy test cases. For the end-easy cases, since each test is only a short distance from ending the game, each solver can quickly iterate through every possible move in a short amount of time. However, looking at the other cases, the results demonstrate how optimizations play a larger role in harder test cases that necessitate exploring a larger number of deep branches. Looking at the time complexity for alpha-beta pruning can provide insight behind the results. The time complexity of alpha-beta pruning is defined as $O(b^{\frac{d}{2}})$, where b, denotes the branching factor, and d refers to the branching depth (branching factor; Knuth & Moore, 1975). It is important to recognize that the optimizations made to each solver do not modify the alpha-beta pruning algorithm, but instead optimizes the solver by reducing the amount of branches, and hence nodes, that the solver must explore for each test case. This means that the computation time of each solver is proportional to the amount of nodes needed to explore, determined by the branching factor b, as well as the depth d. Bitboards provide a computational boost by utilizing the fact that bitboards are more compact than a typical array, and that bit operations are generally faster than standard arithmetic operators. In turn, the CPU can simulate every possible position much faster, drastically reducing the average time spent per test case. This is illustrated by the significant decrease in average time between the solver with the bitboard but no transposition table and the unoptimized solver with neither bitboard nor transposition table.

4.2.2 Average Number of Nodes Explored Per Test Case

For the solver without a bitboard or transposition table, the branching factor is far greater than that of the complete solver. Furthermore, without iterative deepening, $b^{\frac{d}{2}}$ very quickly grows to a very large number, explaining the exponential increase in time between the optimized solvers and relatively unoptimized solvers. This also explains why weak solvers always outperform their strong counterparts. While strong solvers explore every single position to secure a win from any position, weak solvers utilize a smaller score window of $[-1, 1]$ to find *any* winning or draw position. As such, weak solvers can reduce the number of nodes explored by pruning branches that don’t immediately lead to a winning or draw position, unlike a strong solver which explores every single position. This is reflected in the results, which interestingly show that weak solvers generally halve the average node count and average time per test case of their strong counterparts.

5. Further Research

The experiment conducted in this experiment runs the program on a single thread, out of the many threads that could be utilized. As such, this presents an opportunity to explore how the DFS algorithm utilized in this experiment can be parallelized, and the implications of such on the performance of solvers, especially for unoptimized solvers on hard test cases.

Moreover, several challenges and discoveries appeared while running the experiment. I noticed that even when the CPU load was at 40%, the performance was considerably different than when the CPU load was at 27%. This issue was exacerbated since I had to use a Linux virtual machine, which performs sub-optimally compared to a native system. Additionally, other discrepancies in my data, such as the complete solver performing worse in the end-easy test set warrant further investigation into their origin.

6. Conclusion

In this investigation, a comparison between the efficiencies of optimizations for a Connect 4 game solver utilizing alpha-beta pruning was discussed. Data and logical reasoning, as well as theoretical discussion of big O notation was included for comprehensive analysis.

The results demonstrate that each optimization exponentially reduces the average number of nodes explored for each testing set, thus substantially cutting down on average computation time for each test set. This allowed for the complete solver to solve the hardest test cases in a reasonable time, while the unoptimized solvers took multiple minutes to solve harder cases. Furthermore, external factors and discrepancies in the experiment present further opportunities for exploration through external factors such as parallel processing, analyzing virtual machine performance, and more.

Finally, while the findings of this investigation focus on the application of DFS and optimizations in game theory, the underlying concepts can be applied to projects in far wider variety. Take for example, a navigation app. One could imagine the client's location as a node, with the world as a collection of possible paths to get to a location. Through optimizing the search, one can reasonably quickly find the optimal path from one point to another. All in all, the speed and efficiency of the underlying data structures and algorithms discussed in this paper serve as important considerations for computer scientists, game developers, and others in all types of fields.

References

Belford, G. G., & Tucker, A. (2020, September 1). *Computer science*. Retrieved from Encyclopedia Britannica: https://www.britannica.com/science/computer-science

branching factor. *Oxford Reference.* Retrieved 19 Jan. 2022, from https://www.oxfordreference.com/view/10.1093/oi/authority.20110803095524330.

Brilliant.org. (n.d.). *Minimax*. Retrieved September 19, 2021, from Brilliant: https://brilliant.org/wiki/minimax/

Britannica, T. E. (2017, April 12). *Data structure*. Retrieved from Encyclopedia Britannica: https://www.britannica.com/technology/data-structure

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. Artificial intelligence, 6(4), 293-326.

Laramée, F. D. (2000, June 11). *Chess Programming Part II: Data Structures*. Retrieved from gamedev.net: https://www.gamedev.net/tutorials/_/technical/artificial-intelligence/chess-programming-part-ii-data-structures-r1046/

Mohr, A. (2007). Quantum Computing in Complexity Theory and Theory of Computation. Carbondale, Illinois, United States of America. Retrieved September 19, 2021, from http://www.austinmohr.com/Work_files/complexity.pdf

Pons, P. (n.d.). Solving connect 4: How to build a perfect AI. Solving Connect 4: how to build a perfect AI. Retrieved January 12, 2022, from http://blog.gamesolver.org/

Russell, S. J., & Norvig, P. (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc.