UPPSALA UNIVERSITET

High Performance Programming
Individual Project
4x4 tic-tac-toe AI

Anton Sjöström

March 12, 2021

1 Introduction

The subject of AI has become popular in recent times and the field of AI is regularly ranked as one of the most interesting and fast-growing fields and is generating over a trillion dollars in revenue each year (Russel Stuart, Norvig Peter, chapter 1.1). Before solving real world problems, it can be good to restrict our environment to games such as chess or tic-tac-toe. This simplified environment is a good place to start when studying AI because the agents are restricted to a limited set of actions. (Russel Stuart, Norvig Peter, chapter 5).

In this project, an AI has been built in C to play the game tic-tac-toe. The rules of the game are straight forward. Two players play against each other, one using crosses and one using circles, and the first person who get three in a row either vertically, horizontally or on the diagonal wins. The game is played on a 3x3 grid and the players take turn placing their particular piece.

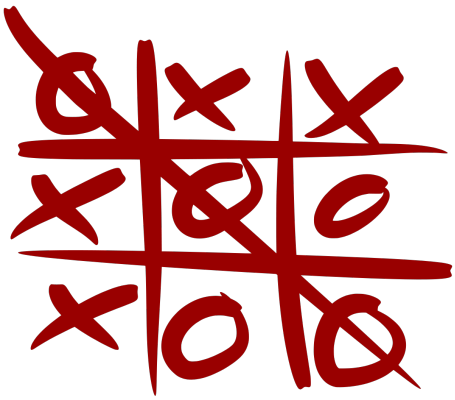

Figure 1: A finished game of tic-tac-toe where circle won. Source: Tic tac toe, Symode09, 2009. Taken from https://en.wikipedia.org/wiki/Tic-tac-toe, 2021

2 The problem

To make an AI play tic-tac-toe, multiple approaches can be made. The aim of this project is to make a working AI and also demonstrate code optimization. Therefor a search algorithm has been implemented to evaluate the best move since it can be optimized in several ways, both in serial and parallel. By letting the AI go through all the possible moves from the current board configuration to all the possible endings, the AI will know which moves lead to a win, a draw or a loss. The AI can then pick the best move based on this search. If two perfect players play from the start to the end, a draw will always occur. But if the AI faces a human, it might win. In either case, the AI can make sure it will at least never lose.

How do the AI know what the outcome of the game will be several moves in advance? It will use an algorithm called **Minimax**. Minimax is used to play the optimal move assuming that the opponent will respond by also playing the optimal moves. (Russel Stuart, Norvig Peter, chapter

5). By telling Minimax that a win for the AI is given by the value of 1, a draw 0 and a loss -1 one can use Minimax recursively to play out the game and find the optimal move. The AI wants to maximize the value of the game, but the opponent will try to minimize the value of the game. Minimax will be explained in more detail later.

After the first move there will be eight different possible responses by the AI. The AI will then check what is the best response by hypothetically play out every possible game that can branch out from that move. This will lead to a tree structure where the first node has eight children, the second layer has seven children and so on until a leaf is reached where the game has either ended in a win, a draw or a loss. An upper bound of the amount of moves that has to be considered from an empty board is 9!, which equals to 362880 moves. This is a reasonable number to look through. It is in fact small enough to quickly be searched through in C. Also, the second move will only be 8! which is 9 times less than the first move. The longer the game has been played, the quicker the search for the best move will become.

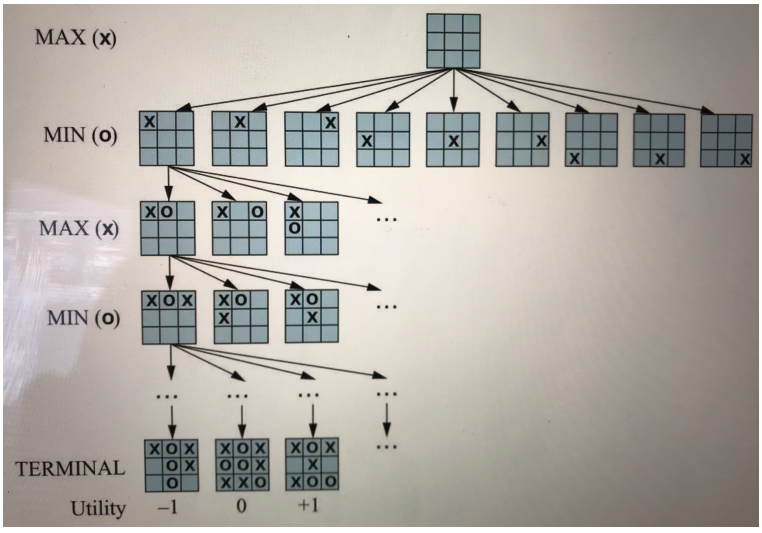

Figure 2: Search tree for tic-tac-toe. Source: Russel Stuart, Norvig Peter, chapter 5

This is a good sign for the AI but bad for the project since almost no optimization will be necessary. It will either way be really fast to solve. Therefor this project has raised the bar and let the AI play a 4x4 tic-tac-toe where a player needs four in a row to win. This drastically increases the upper bound of possible moves to 16! which might not look that much bigger than 9!, but it is 57657600 times larger than 9!.

With this upper bound, optimization will carry large importance, in fact such a large importance that without optimization the AI will never be able to come up with a response in any reasonable amount of time. However, as one will see later on in this report, the optimization carried out in this project allows the AI to actually handle this task and play very well.

2.1 Minimax

The Minimax algorithm is described briefly above but in this section it will be described more thoroughly. The Minimax algorithm is a general algorithm for AI to make the optimal move in various games, not only tic-tac-toe. (Russel Stuart, Norvig Peter, Artificial Intelligence A Modern Approach, fourth edition, chapter 5). The reasoning behind Minimax is to play the best move with the hypothesis that your opponent will do the same. This means that your opponent wants to make your optimal move as bad as possible. This thought process will go on recursively until all moves have been hypothetically played and the AI knows which move leads to the best outcome in the end. Then the AI will play that move.

The algorithm will have a tree structure where the root (empty board 4x4) has 16 children and each of these children have 15 children and so on. This means that the amount of children will have an upper bound of 16!. This is an upper bound and no tree will be exactly this size since a game can be ended already after seven moves. This means that some branches will end faster than others. However, it can be good to have an upper bound.

Instead of going through each and every branch, one can use a serial optimization trick called Alpha-Beta pruning (Russel Stuart, Norvig Peter, chapter 5). Alpha-Beta pruning works by setting an upper and a lower bound of the value of a node in the tree by using knowledge from the search. Since one already knows from the start that the minimum will be -1 and the maximum 1, one will tell the Minimax algorithm that. This means that some branches do not need to be investigated further since the AI knows which branch will be picked as soon as it sees either -1 or 1, depending on which player's move it is investigating. During the search, the alpha and beta can change as well, making it not necessary to only look for -1 and 1, which makes the algorithm more dynamic. A Minimax tree in general is presented in figure 3 where the shaded parts do not need to be evaluated because of Alpha-Beta pruning.

In general, the Minimax algorithm is of order $\mathcal{O}(b^d)$ but with Alpha-Beta pruning, in the best case, it can be reduced to $\mathcal{O}(\sqrt{b^d})$. (Russel Stuart, Norvig Peter, chapter 5)

3 Optimization techniques

3.1 Serial optimization

To make this section as clear as possible the report will divide it into three parts. The first part will be about algorithm optimization, the second part will be about memory optimization and the third will be about code optimization. Part two and three go a bit hand in hand since a reduction of memory will increase the speed but the author think this division is clearer.

3.1.1 Algorithm optimization

As mentioned above, Alpha-Beta pruning has been implemented to optimize the algorithm in a serial way. This would, in the best case, reduce the time complexity down to $\mathcal{O}(\sqrt{b^d})$ instead of $\mathcal{O}(b^d)$. There is more algorithm optimization that can be done and this will be discussed in the end.

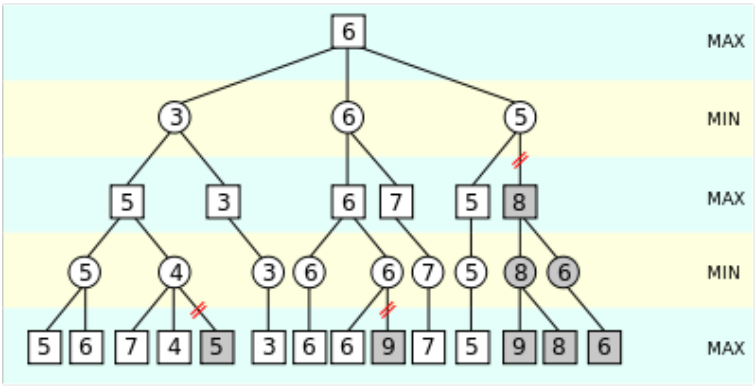

Figure 3: Figure of a general Minimax-tree. In tic-tac-toe, every leaf will be evaluated either as -1, 0 or 1. Here the shaded parts will never be evaluated because of Alpha-Beta pruning. Source: Alpha-Beta pruning example, Jez9999, 2007. Taken from *https* : *//en.wikipedia.org/wiki/Alpha–beta$_p$runing*, 2021

3.1.2 Memory optimization

The memory was optimized a lot. At the beginning the idea was to actually build this tree using a struct called "node" which had a board, a parent and several children. This was hard to implement since each node would have one less child than its parent and therefor another type of tree was considered. This other type of tree have parents, children and siblings. To construct such a tree, let the parent only have one child, and then let that child have a sibling. So if a node would have two children the node would instead have one child and that child would have one sibling. If the node would have four children it would have one child and the child would have a sibling. Then that sibling would have a sibling and lastly that sibling would have a sibling. This is called a "left-child-right-sibling" tree and is demonstrated in figure 4.
This approach worked for small trees, but was very inefficient. Too inefficient to be of practical use. A lot of malloc and free calls had to be done and every board configuration was saved. This took a hit on the memory and a 4x4 grid could not be saved on my computer. This was a problem, however, the great thing about Minimax is that it does not actually have to have an existing tree to go through. The recursion handles this by itself and no more board than the current evaluated board has to be saved! (Some boards at a time will be saved on stack since the algorithm goes through in recursion, but as a programmer one does not have to "manually" save the board). This is great since zero malloc calls had to be done and one could save the board as an ordinary array on the stack. No malloc calls, no free calls and only one board to be saved, this is a lot better.

The board was saved as a an array of chars where each element was either a "x", an "o" or a "-" representing blank. This was done since it was an easy and fast way to represent the board and it was intuitively to understand. Another optimization technique was to use an array of length 16 instead of an 4x4-matrix. Since one will go through the elements in order from index 0-15 it is better to have them next to each other in memory.

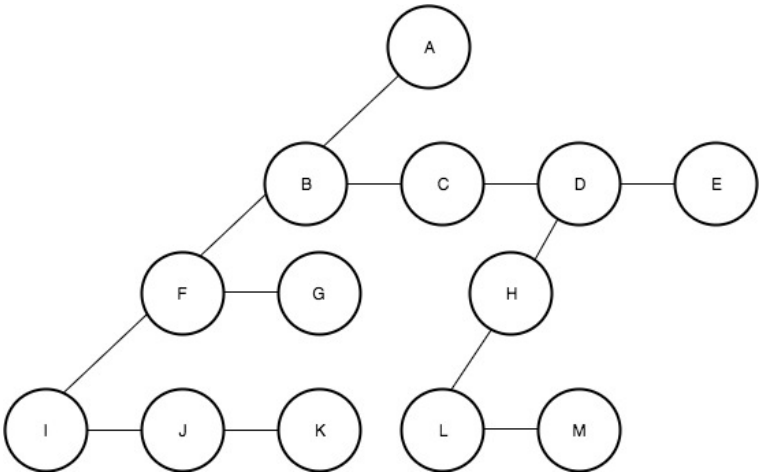

Figure 4: Figure of a general Left-Child Right-Sibling tree. Source: GeeksForGeeks, 2020. Taken from https://www.geeksforgeeks.org/left-child-right-sibling-representation-tree/, 2021.

3.1.3 Code optimization

The optimization flag -O3 has been used to let the compiler optimize as much as possible. Also, by short-circuiting the win-function by first checking if the "pieces" that are four in a row is a blank instead of a cross/circle, one does not have to check further to understand that that row is not a winning row. Inlining the functions that check for a win/draw and possible moves have been done since these are short functions. The other functions have however been accepted as too long to try to use inline.

Loop-unrolling did not seem to be applicable for this project and was not considered. The same regarding SSE vectors and other types of optimization techniques that did not seem relevant to this project.

Since the win/draw and possible move functions are called a lot, it would be good to optimize these functions further. Also, changing places of pieces is also done a lot, therefor the code would also benefit by increasing that speed. For these reasons, another representation of the board was also considered. Instead of saving the board as an array of chars one can save it as two 16-bits unsigned short integers. One integer represents where the crosses are and one represents where the circles are on the board. It is the bits in these integers that represent where crosses/circles are. If the bit is 1, then a cross/circle is there, otherwise it is empty. By using the integers with its bit-representation, one can see where the crosses/circles are. This can intuitively be seen in figure 5. This will save a small amount of memory but the optimal thing is that we can use bit manipulation to change the board configuration, and bit manipulation can be very fast. To place a cross at index 4, one can just switch the fourth bit on the integer "cross" to a 1 from a 0. One can also use bit-operations to determine if someone won or if a move is possible etc. This technique should increase the speed of the code and the project will investigate how much time is saved by this change in board representation.

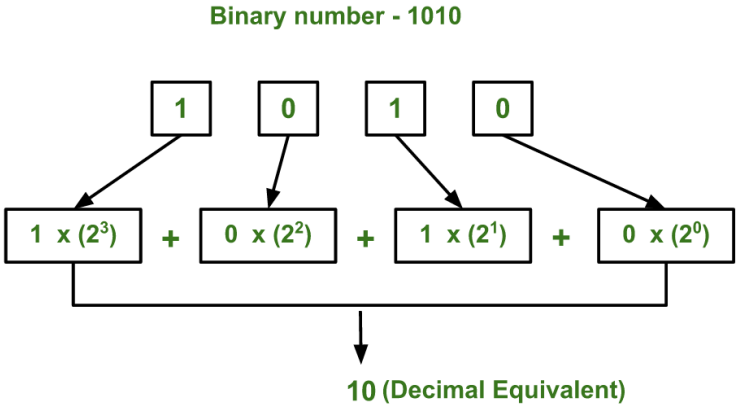

Figure 5: How one integer would represent where the crosses are. This would be on a 2x2 grid, but the idea is the same for a 4x4 grid. Source: GeeksForGeeks, 2021. Taken from https://www.geeksforgeeks.org/program-binary-decimal-conversion/, 2021.

3.2 Parallel optimization

Since one is working on a tree structure, it should be very easy to search in parallel, just give the threads one branch each. This is true with some slight modification. If one uses Alpha-Beta pruning the algorithm is no longer parallel. It seems like one will have to chose to either use pruning or parallelism. However, one can make a small adjustment. By not using pruning between the root's children but for every other children, one can actually have the combination of pruning and parallelism. By evaluating the scores for the root's children in parallel and then pick the best of these, it is possible to use some parallelism. This is good since if two AI face each other, no winning move will ever be discovered but all the root's children will have to be evaluated either way, since neither know they will not find a winning move. Another thing that will be considered is if the code will benefit by letting the parallelism go deeper into the tree by pushing the pruning further down.

The computer tested on was the school's computer Tussilago (Model name: AMD Opteron(tm) Processor 6282 SE) which had 16 cores. One can create 16 threads and give each thread one branch from the root each. This, as said before, is perfectly parallel. Depending on how far into the game the AI is, some threads will have to do a lot more work than others. Some threads will terminate directly since the AI will try to place somewhere that is already occupied by a piece and therefor terminate directly. Some threads will have to go a lot deeper and this will take more time. When one use all 16 cores the total time will be equal to the time it takes for the hardest working thread. One way to counter this is to also put the parallelism further down in the tree to try to get a better load balance but this demand that we push the Alpha-Beta pruning down even further. This will be investigated later on to see if it is more beneficial than to just use parallelism in the first layer.

If one works on a computer with less than 16 cores, one will have to split the work for each thread in a different way. Let's say one only has access to two cores and therefor only uses two threads. Instead of giving each thread 8 branches each one can give them one each and as soon

as they have finished one branch they will pick one of the remaining 14 branches. This is done by using the dynamic scheduling in OpenMP with a chunk-size of 1. This is beneficial since the load balance will be better and no thread will be unlucky and get 8 hard branches and the other thread only get 8 easy branches. It will be investigated if this increases the speed of the code, and the hypothesis is that it will.

Since we have reduced the memory load to only work with one board/two integers, no data sharing is necessary between the threads when one only uses parallelism on the first layer. They can just take its own private copy of the integer/board and manipulate that. If they would have to operate on the same integers/boards, one would have to use a lot of locks and unlocks which would decrease the performance greatly.

When working with deeper parallelism this is no longer the case. One would have to save more of the evaluations globally and make sure not to write to the same place at the same time. One way to fix this is to use a very large global array that every thread can write to and make sure that they never write to the same place. However, this is probably not the best way of doing it but it will make it work without using locks/unlocks which would decrease the speed a lot since it will have to be done all the time.

4 Measurements and discussion

In this section several measurements will be done. There is a lot of different versions of the now finished and optimal code and each one is saved to see how much specific optimizations increase the speed.

They will all, except for the first two subsections and the section with different parallelism depth, be tested on the board where the opponent has placed the first cross at the index 10. The AI will now have to calculate the best response. The test will be done on the computer Tussilago as mentioned before.

4.1 -O3 or not

Compiler flags play a crucial part when optimizing since the compiler can help the programmer out a bit. In this subsection the speed-up from the optimizing flag -O3 will be investigated. The code here will not include pruning since that will be investigated in the next subsection. The code will use the integer representation of the board. The start configuration will be slightly different since the code would take to long otherwise. A circle is placed at index 9 and 6, and a cross is placed at index 8 and 10 to reduce the computations needed.
One can see from table 1 that the compiler flag helped out a lot. The speed-up was around 4.5, which is significant.

4.2 Alpha-Beta pruning or not

This subsection will investigate how much gain one could get by including Alpha-Beta pruning into the algorithm. The only difference between the codes is that one uses Alpha-Beta pruning and the other one does not. Both will be using the integer representation of the board.

Table 1: Time comparison between no compiler optimization and -O3

	without -O3			with -O3	
Run	Real time	User time	Run	Real time	User time
1	2m16.741s	2m16.7173s	1	30.044s	30.03s
2	2m16.752s	2m16.729s	2	29.863s	29.857s
3	2m16.727s	2m16.705s	3	30.020s	30.013s
4	2m16.750s	2m16.727s	4	29.789s	29.782s
5	2m16.745s	2m16.723s	5	30.061s	30.055s
mean	2m16.743s	2m16.720s	mean	29.955s	29.947s

To make the non-pruning algorithm measurable, one need to have placed out four pieces already and let the AI place the next move. A circle is placed at index 9 and 6, and a cross is placed at index 8 and 10. Of course, the pruned algorithm will also be tested on the same configuration to make the comparison fair.

Table 2: Time comparison between non-pruned and pruned algorithm

	without pruning			with pruning	
Run	Real time	User time	Run	Real time	User time
1	30.044s	30.03s	1	0.045s	0.043s
2	29.863s	29.857s	2	0.045s	0.044s
3	30.020s	30.013s	3	0.045s	0.044s
4	29.789s	29.782s	4	0.045s	0.044s
5	30.061s	30.055s	5	0.045s	0.044s
mean	29.955s	29.947s	mean	0.045s	0.044s

By looking at table 2 Alpha-Beta pruning was left out, the mean user time was 29.947 seconds. When Alpha-Beta pruning was used, the mean user time was 0.044 seconds. This is a significant speed-up, the speed up is around 681.

The Alpha-Beta pruning should in the best case be $\mathcal{O}(\sqrt{b^d})$ and in the worst case be $\mathcal{O}(b^d)$. Is this something that can be seen when measuring? The average branching factor is quite tricky to measure since the root have 12 children, then these 12 children have 11 children each etc. This means that we have 1 node with 12 branches, 12 nodes with 11 branches, 12*11 nodes with 10 branches etc. We would also have 12! nodes. By this way of calculating the average branching factor would be around 2.7. However, this is slightly lower than what it should be, since a lot of games end earlier and never branches down the whole way. Thus, to compensate for this, one can increase the branching factor a little bit. Without any extra calculations this report assumes that a better average branching factor should be around 3.

An algorithm working with complexity $\mathcal{O}(\sqrt{b^d})$ should be $\mathcal{O}(\sqrt{b^d})$ times faster than the $\mathcal{O}(b^d)$.

This means that it should theoretically be around $3^6 = 729$ and is in the same ball park as our measured speed-up. Since one could expect Alpha-Beta pruning to be $\mathcal{O}(\sqrt{b^d})$ in the best case it seems reasonable that we get a slightly lower speed-up. This mathematically reasoning should be taking with a grain of salt since the branching factor is not exact, but it shows how effective pruning can be.

4.3 Boards or integer representation

In the beginning of this project the game state was saved to an array of size 16 which included chars to represent the board. This was beneficial since it was intuitively to understand. However, a more efficient way was to store it as two unsigned 16-bit integer, where each bit in this integer represented a position on the board. By representing the board this way, one could take advantage of bit-operations. Since manipulation of the board was done a lot of times, this should increase the speed of the code. Here pruning is done for both algorithms and since pruning is used one can start with just one cross placed at index 10, which makes the task much more difficult, around 2730 times harder than with four pieces placed out.

Table 3: Time comparison between the array and integer board representation

with array representation			with integer representation		
Run	Real time	User time	Run	Real time	User time
1	25.919s	25.913s	1	15.802s	15.798s
2	25.313s	25.308s	2	15.881s	15.878s
3	25.630s	25.625s	3	15.831s	15.827s
4	25.366s	25.360s	4	15.888s	15.885s
5	25.934s	25.929s	5	15.813s	15.810s
mean	25.632s	25.627s	mean	15.843s	15.840s

From table 3 one can see that the integer representation is faster than using an array of chars. The bit-operators are fast and perfect for this task and representation. The speed-up is around 1.62.

4.4 Parallelism

4.4.1 Task-queuing

When using Alpha-Beta pruning, the task is no longer completely parallel. However, by pushing down the pruning to the lower levels, the layers above can be evaluated in parallel. By using task-queuing one can decide how many layers to do in parallel and lay those on a task-queue for threads to evaluate, this requires nested parallelism. This is good since more parts of the tree can be evaluated at the same time. However, it has its disadvantages as well. One needs to save a lot more of evaluations in a global array and this array will grow exponentially. This is not good for memory efficiency, but if one does not use too many layers in parallel it seems to work. There might be a better way to save these by using locking and unlocking, but this seems to be the more promising approach. The hypothesis was that this would not improve the speed, in fact by reducing the pruning it should be slower. Therefor, not too much optimization was done before testing. If it slowed down the code too much there would be no meaning of making it slightly more efficient and more time could be spent optimizing the more promising result.

In this section the report will investigate if this is beneficial when no Alpha-Beta pruning is used and when Alpha-Beta pruning is pushed down to the layer below the last parallel layer. For these measures, the board configuration will be three crosses in index 9, 10 and 11 and the AI should realize that the only non-losing move is to put a circle in index 8. The algorithm is using the integer representation.

Table 4: Time comparison for different amount of threads and parallelism depth

Without Alpha-Beta and 16 threads

depth	time
0	3m 44 s
1	30.887 s
2	16.795 s
3	14.973 s
4	15.146 s

Without Alpha-Beta and 30 threads

depth	time
0	3m 44 s
1	31.012 s
2	11.694 s
3	10.144 s
4	10.134 s

Pushed back Alpha-Beta, 16 threads

depth	time
1	0.688 s
1	0.088 s
2	0.182 s
3	0.465 s
4	0.670 s

Pushed back Alpha-Beta, 30 threads

depth	time
0	0.688 s
1	0.098 s
2	0.153 s
3	0.335 s
4	0.464 s

One can see from table 4 and figure 6 that deeper parallelism is beneficial when the algorithm never uses pruning, at least up to a certain depth, after that it does not really help with more depth. But overall, it seems to be good to use some parallelism depth. This makes sense since no trade off between serial and parallel optimization is done. Also, another interesting thing is that when one uses 30 threads it actually increases the speed, even though the machine only has 16 cores. These extra threads help with the work balance.

When using Alpha-Beta pruning, it seems to be better to have a shallow parallelism depth since the pruning is that efficient. One can see in table 4 and figure 7 that the optimal parallel depth is 1. This makes sense since either way one would have to evaluate all these first layer branches to find which one is optimal, but that further down one can use pruning. This insight will lead us to the next section where only the first layer will be in parallel to make the most optimal algorithm.

One thing to also note is that one layer parallelism with pruning is much faster than pushing it down. Therefor, there would be no benefit by trying to optimize the code using task-queuing.

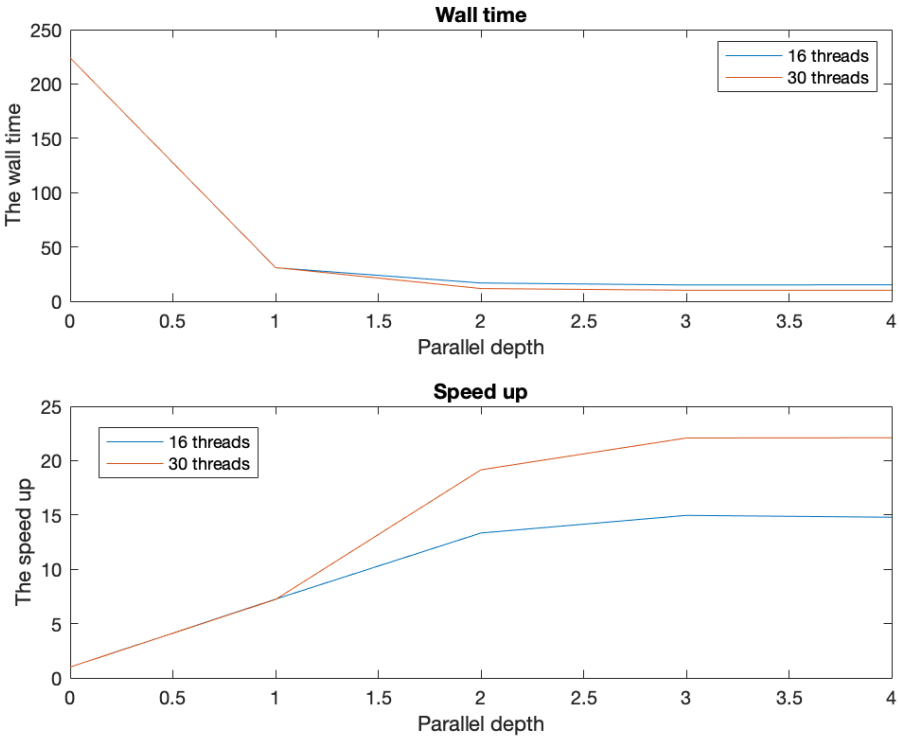

Figure 6: The wall time and speed up for the algorithm not using pruning

It would either way not improve the speed enough to benefit from deep parallelism. Therefor, that part of the project was never further optimized and instead abandoned. But it was a good experiment to see how the trade off between pruning and parallelism behaved and it gave some important insights.

4.4.2 Parallel loop

Alpha-Beta pruning is important for speed as one saw in section 4.2. It is also better than what one will get from deep parallelism as one saw in section 4.4.1. The optimal level of parallelism seems to be to only parallel the first layer. Since only the first layer is going to be in parallel, it is easiest done with the "parallel for" command in OpenMP.

For this section, a plot of the time and speed-up will be presented. The timing will be taken with the build in timer in OpenMP. Table 5 will present the wall time for the different amount of threads and figure 8 will plot the speed up and the wall time in a plot. It is also interesting how the scheduling (static vs dynamic) is affecting the speed up, hence this will be investigated as well.

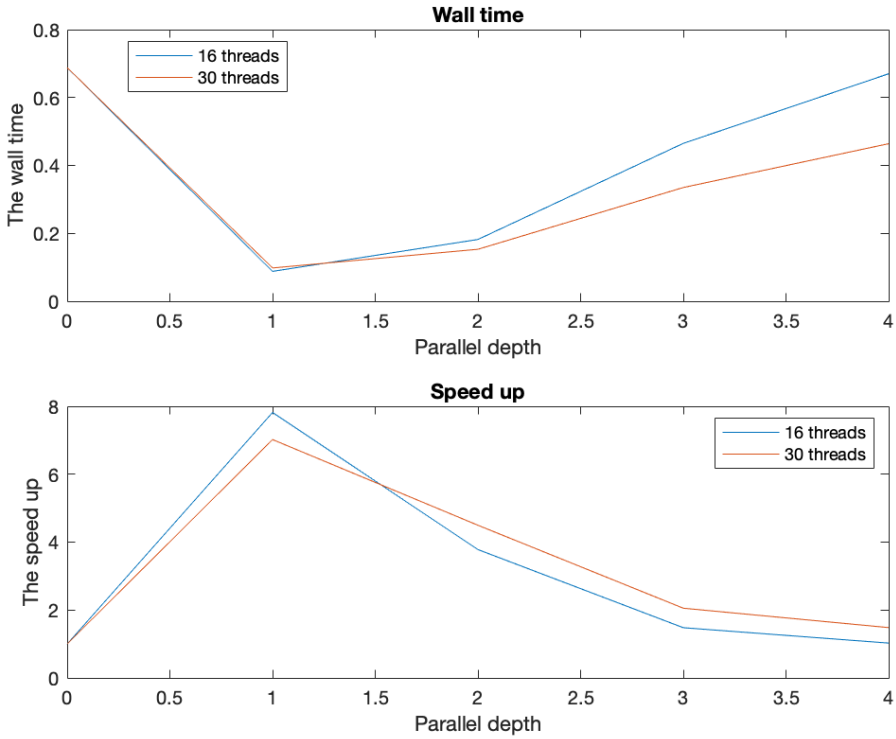

Figure 7: The wall time and speed up for the algorithm pushing back pruning

4.4.3 The dynamic result

One can see quite clearly from figure 8 that the speed up is linear with the respect to the amount of threads for the dynamic scheduling, which would make sense in this case since all moves are around equally as good. They should take around the same time to search through when only one piece is already on the board. However, this is not necessarily true when more pieces are already placed on the board. One interesting thing to notice is that the time did not really change between 10 and 11 threads. Why is this? This is because branch 10 is really fast to evaluate since no piece can be placed there, and will therefor terminate directly. Here it does not really matter if we have 10 or 11 threads since thread 10 would take both branch 10 and 11 in around the same time it would take thread 11 to take branch 11. Therefor, it should take around the same time or only be a little bit slower since one more thread has to be created. Another plateau seems to occur between 15 threads and 16 threads. One quickly realizes that this is because of the same reason. Since thread 10 can take both branch 10 and branch 11 in almost the same time as branch 11, one does not gain anything by having 16 threads instead of 15.

Why do one have a speed up spike between 14 and 15 threads? This is because at 15 threads they all get one branch each (except for thread 10 that get both branch 10 and another one) and

Table 5: Time comparison for different amount of threads

dynamic 1-16 threads		static 1-16 threads	
Threads	time	Threads	time
1	15.856s	1	15.890 s
2	8.527s	2	8.459s
3	5.735s	3	5.945s
4	4.562s	4	4.867s
5	3.563s	5	4.824s
6	3.320s	6	3.587s
7	3.053s	7	3.584s
8	2.596s	8	2.530s
9	2.466s	9	2.530s
10	2.311s	10	2.529s
11	2.312s	11	2.526s
12	2.111s	12	2.521s
13	1.970s	13	2.527s
14	1.832s	14	2.530s
15	1.364s	15	2.531s
16	1.376s	16	1.350s

therefor it takes around the same time to go through all branches as it takes to only go through the hardest branch. This is good for synchronization since all threads will be done around the same time and no thread will have to do double the amount of work.

4.4.4 The static result

For the static scheduling, one sees another interesting result. Between 1 to 8 threads it is quite linear, but after 8 threads it has a long plateau until 16 threads are used. Why is this? This is because before 8 threads, the threads get two or more branches to evaluate each. When one increases the number of threads up until 8, each thread get less work to do and more can be done in parallel. But from 8 threads and up, some threads get one branch and some still get two branches. This means that the total time will be the time it takes for the hardest working thread. This will continue until 16 threads when the threads get one branch each. When this happens, the speed up lines up with the dynamic as well.

In this example one can see the power of dynamic scheduling. Without using it, the load balance is bad, which is not good for the end synchronisation. But when one only gives a thread one branch each until they are ready to tackle the next, the speed up continues to be linear.

4.5 Further optimization

More optimization could probably be done to this AI but because of the time limit of the project, one would have to end somewhere. A lot has been tried and a lot of success has been achieved, but there is always room for improvements. One improvement is that one could probably combine the parallel sections and the pruning in such a way that it uses parallel search in the sub trees that can

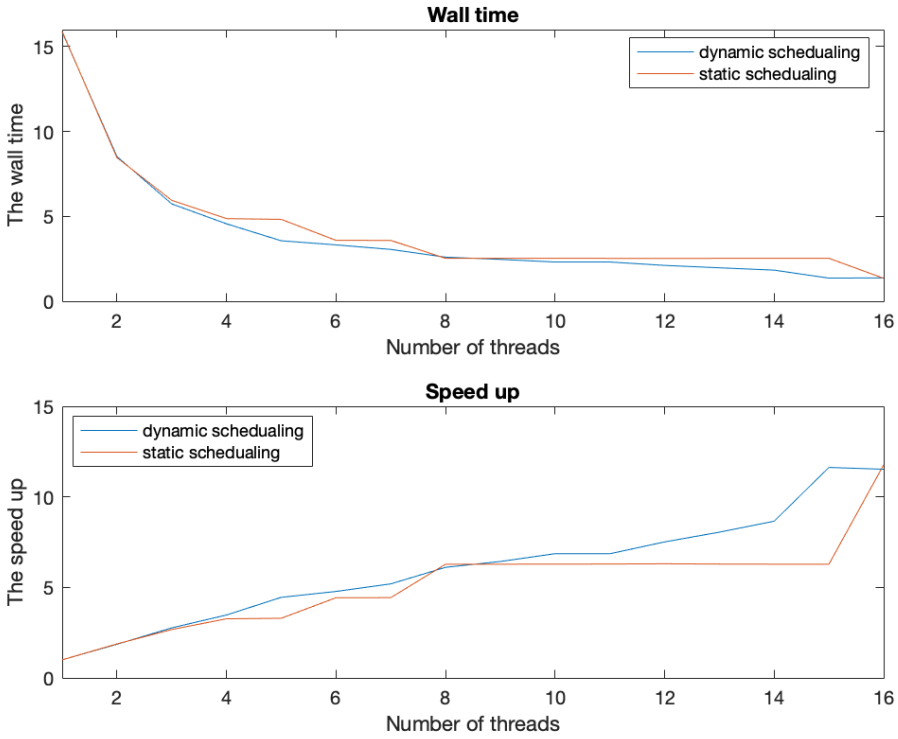

Figure 8: The wall time and speed up for the parallel code

not be pruned away. It is hard to know in before hand which sub trees this would be, but if one could sort that out it should be possible to do a more parallel optimization.

Another optimization that could be used is to change the way one saves the evaluations in the array in section 4.4.1. One could probably save the evaluations as two bits. Using 00 to represent a loss, 01 to represent a draw and 10 to represent a win. By doing this one could probably make the array take up less space, either by using packing in some smart way or just by using a smaller integer representation. Another way one could optimize it, is to decrease the size of the array by using lock and unlock. In that way, when some information is used, other information can be written over that and thus decrease the necessary size of the array. However, since this method was overall worse than to only use parallelism on the first layer, it was never further optimized.

More optimization could be done to the algorithm. Alpha-Beta pruning is the heavy lifter to increase the speed, but one can get a better result if one explores the good moves first instead of always exploring the moves from index 0-15. This would lead to better pruning and therefor faster evaluations. Another thing one can do is to not search the whole way to the end but instead search to a specific depth. Then one can evaluate the position at that point. This evaluation can be done in various ways. One common way is to use Monte Carlo tree search. In a Monte Carlo tree search,

one evaluates the positions by playing out random moves from a position for a lot of games and see what the mean result of these simulations converges to and let the evaluation be that score. This means that one can search more shallow and still find a good evaluation.

Lastly, there is multiple ways of getting to the same board configuration. Let's say we have an empty board and pieces are placed in these two orders: 1-2-3-4 and 1-4-3-2. In this case the board configurations will be the same. It would be unnecessary to search both since they will end up with the same evaluation. By letting the AI remember board configurations it has already searched, one can improve the speed. However this requires more memory.

5 End discussion

This report has investigated how to implement and optimize a tic-tac-toe playing AI using the Minimax algorithm. Several optimization techniques have been used to let the AI play a whole game of 4x4 without any significant delay, something the original algorithm could not do. The optimizations techniques have both been done in regard to algorithm optimization using Alpha-Beta pruning, memory optimization by not actually implementing a tree, code efficiency by using bit operators and compiler flags, and lastly OpenMP and dynamic scheduling to take advantage of the parallelism of the algorithm.

A script in the attached files let you play versus the AI where it uses the most optimized version of the code. This is only for demonstration of the AI and it can be fun to try out.

Disclaimer: The game is no way bullet proof from errors in the user input. This was a last minute implementation just for fun and is not part of the project over all. If you want to play it, only put in legal moves.

6 References

Alpha-Beta pruning example, Jez9999, 2007. $https://en.wikipedia.org/wiki/Alpha–beta_pruning$, 2021

GeeksForGeeks, 2020. https://www.geeksforgeeks.org/left-child-right-sibling-representation-tree/, 2021

GeeksForGeeks, 2021. https://www.geeksforgeeks.org/program-binary-decimal-conversion/, 2021.

Russel Stuart, Norvig Peter, Artificial Intelligence A Modern Approach, fourth edition, 2016

Tic tac toe, Symode09, 2009. https://en.wikipedia.org/wiki/Tic-tac-toe, 2021