Advanced Algorithmics:
Project Report

April 2023

Abstract

Project report for the Advanced Algorithmics course of the academic year 2022-2023, implementing different strategies for the game Blobwars.

Introduction

Below, a summary of optimizations and performances of the various algorithms implemented for the game Blobwars.

1 Minmax

1.1 Basic Algorithm

The first algorithm we implemented is the simple version of Minmax.
The Minimax algorithm is a technique commonly used in two-player games, such as chess, tic-tac-toe, or Blobwars.

The operation of the algorithm is relatively straightforward. It uses a recursive approach to explore the complete game tree, evaluating each possible move for each player until reaching a final game state or the desired depth.
Each final state will then be evaluated by a function that will assign it a numerical value indicating the active player's gain.

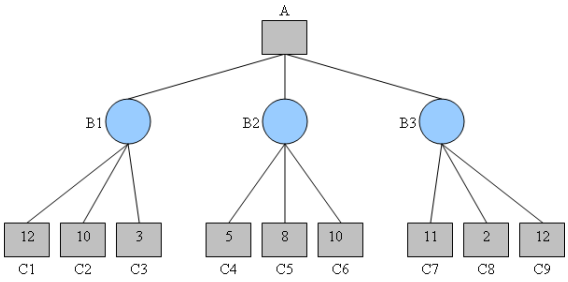

Figure 1: Minmax Algorithm - Wikipedia

During the search for the best move, players will be alternated for each level explored. At each turn, the active player will seek to maximize the value of their move, while the opponent will try to minimize this same value.
Finally, at the root of the tree, we will take the maximum of the minimum values of all possible moves, to maximize our player's gain.
The only conflict with the code was that the function evaluating the final states did not calculate the gain but the loss, so instead of maximizing the minimum values, we minimized the maximum values.

1.2 Parallel Minmax

The main problem with the Minmax algorithm is that it requires exploring the entire tree of possibilities. This exploration can go relatively well with games like tic-tac-toe, which do not have many possible moves to calculate, but the algorithm quickly becomes laborious for games with high possibilities like chess.
For Blobwars, we could only look 4 moves ahead. Indeed, with a depth greater than or equal to 5, the calculation was much too complex, hence the need to parallelize.
Here, with the help of Rayon, we were able to implement a parallel version of Minmax to explore the game tree faster. This is possible because there is no intersection or dependency between two different nodes.
Parallelization was therefore relatively simple once we understood how to use Rayon, but it did not improve performance as we will see later.

2 Alphabeta

2.1 Basic Algorithm

The Alpha-Beta algorithm is an optimization technique for the Minimax algorithm, which reduces the number of nodes explored during the search for the best decision to make.

The Alpha-Beta algorithm prunes branches of the search tree that cannot influence the final decision. To do this, the algorithm uses two values called "alpha" and "beta", which represent the lower and upper bounds of the values a player can obtain from the current game state.

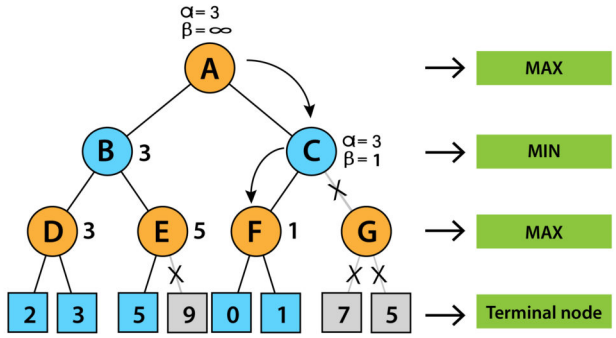

Figure 2: Alpha-Beta Algorithm in AI - MyGreatLearning

Similar to the Minmax algorithm, the Alpha-Beta algorithm traverses the game tree recursively. When a node's alpha value exceeds its beta value, it means that the search tree branch under this node cannot influence the final decision as it leads to a situation less favorable for the player. This branch is therefore pruned and won't be explored further.

2.2 Alphabeta Table

The implementation of the transposition table is an optimization technique for the Alpha-Beta algorithm that reduces the number of nodes explored during the search for the best decision to make using a cache memory.

The evaluations of positions are stored in a transposition table, so they can be reused later if encountered again during the exploration of the search tree.
For a two-player game like ours, a Zobrist hashing function will be used for the hashmap, a common method to uniquely identify each position on the game board. Thus, you have implemented the *zobristkey* function to generate the key corresponding to each position on the game board, using a Zobrist hash table.
The Zobrist hash table is initialized with random values, ensuring that each game board position will have a unique key, and two identical positions will have different keys.

During the exploration of the search tree, the Alpha-Beta algorithm stores in the transposition table the evaluated positions with an alpha and beta value. If the same position is encountered later during the tree exploration, the algorithm can use the stored alpha and beta values to prematurely cut off the search if the stored value is accurate enough.

2.3 Alphabeta Pass

Another optimization of the Alpha-Beta algorithm was implementing Null Move Pruning. This technique involves conducting a "null" move search (i.e., without making a move) at certain levels of the search tree.

If a player is in a very favorable position, they should be able to forfeit a move without suffering too much damage. Thus, by conducting a null move search, it is simulated that the current player does not make a move, and the other player makes two consecutive moves instead.

If the current player is still in a favorable position after forfeiting their turn, it can be assumed that their position was excellent, and the moves that led to this position are equally good. The search can then be prematurely cut off using the evaluation of the current position.

However, using this technique, it is possible to miss an even better move, which can therefore impact the game's outcome. It will therefore be necessary to test several arbitrary values to decide when a position is very favorable (for example, the player wins with 9 blobs ahead), in which case null move pruning will be tested.

3 Benchmark

3.1 Criterion

To test the performance of the different algorithms implemented for the Blobwars game, we used the Criterion.rs library to perform benchmarks. We created a *benchmarkpergroup* function that takes a benchmark group, a name for the algorithm to be tested, and the maximum depth to be tested as parameters. This function goes through all the depths and performs a benchmark for each depth. This benchmark corresponds to a game of the algorithm to be tested against the MinMax of depth 3. We then measure the time it takes for the game to finish, which is not perfect because we should rather measure the performance to find a cost, because if the game drags on the measurement is skewed.

We then grouped the benchmarks together for each algorithm, and put each group together to be able to trace in the same graph the performances of the different algorithms.

3.2 Results

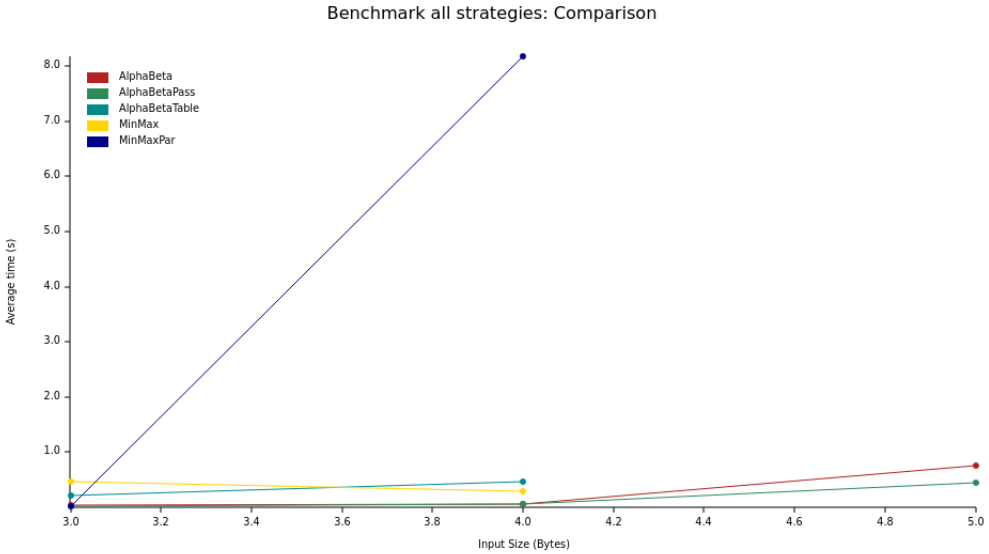

Figure 3: Execution time of a game according to the depth

The first visible result is the very poor performance of Parallel MinMax. Thanks to the benchmark, it is easy to see that our implementation has errors. We therefore decided not to display this algorithm in the graph to better distinguish the performance differences of the other algorithms. Moreover, we increased the maximum depth, sacrificing the number of samples (going from 100 to 20).

Then we obtain:

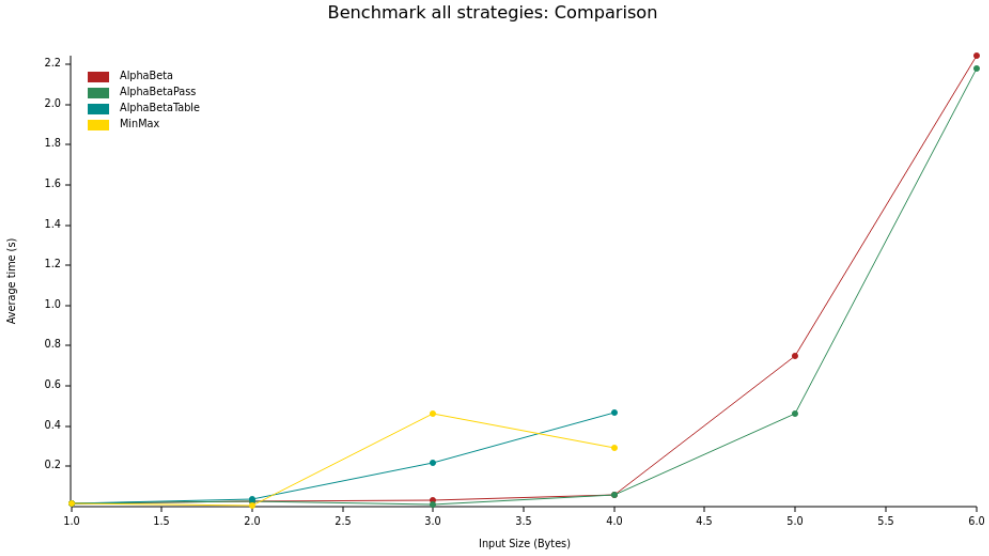

Figure 4: Execution time of a game according to the depth, without MinMaxPar

Firstly, we notice that the MinMax algorithm is the least performant. Its performance improves when moving from a depth of 3 to 4, which may seem surprising. However, remember that we

measure the time of a game. Here, with a depth of 4, the algorithm will win in far fewer moves, which totally counterbalances the performance loss due to the increased depth.

Furthermore, the AlphaBeta algorithm with a transposition table is not very performant, so we had to limit ourselves to a depth of 4. We can therefore say that there are very few collisions, so we almost never save time by not recalculating a position already in the table. Moreover, for almost every move, we have to create a hash of the position and store it in the table, which therefore significantly increases performance.

Therefore, we are left with the AlphaBeta and AlphaBetaPass algorithms, corresponding to the null move pruning optimization. It is firstly notable that these two algorithms, like with the transposition table, have exponential performance in depth, as they are all based on AlphaBeta.

However, we can note a slight improvement in performance for AlphaBeta with null move pruning. We have found, after tests, that to optimize the win rate, null move pruning should be tried from the depth $maxdepth - 3$, because before that we too often risked missing a better move. We had set additional conditions that the player had to be in a very favorable position, meaning with a lead of 7 blobs. If these two conditions were met, we could try the null move, and if after the two consecutive moves of the opponent, the player was still winning by 7 blobs (unchanged score), then we stopped the search and chose the considered move.

If we decrease the restriction of these two criteria, with a lead of only 4 blobs, and a lead after the two consecutive moves of the opponent of 2 blobs, we increase the cases of null move pruning and therefore the performance. However, this also results in a considerable loss of game quality, the algorithm could then lose against an AlphaBeta of depth 4, which is unthinkable with strong restrictions on the criteria.

4 Desired Optimizations

It would also have been possible to parallelize Alpha-Beta, by only parallelizing the first level of nodes.
The difficulty lies in the fact that the alpha and beta values are dependent on previous calculations. The use of mutexes on alpha and beta was conceivable, so they are not modified concurrently.

Conclusion

In conclusion, several versions of Minmax and Alpha-Beta have been implemented. The only interesting optimization was null move pruning, which with good arbitrary values for setting parameters, significantly increased the win rate. Moreover, implementing a complex algorithm in parallel is not easy, and often seems less performant than the non-parallelized algorithm, due to the many effective optimizations of the Rust compiler.