Softwarepraktikum SS 2025
Assignment 4

Group 6

Henri Schaudinn	456738	henri.schaudinn@rwth-aachen.de
Malte Ewald	456139	Malte.Ewald@rwth-aachen.de
Frithjof Alms	456247	frithjof.alms@rwth-aachen.de

Additional contents

1.1 Attribution

Malte worked on task 1 and implemented the move sorting algorithm. Henri compared the move sorting algorithm with normal alpha beta pruning and completed task 2. Fithjof implemented the iterative deepening algorithm for task 3 and the improved move sorting algorithm. The javadoc documentation and code comments were written by the the author of the corresponding function.

1.2 Analysis of computation time usage

1.2.1 Improved move sorting algorithm

This algorithm works like the moveSort function described in Section 1, but, when called with a depth limit of 1, enters a moveSortLeaves function. This function works like the MiniMaxLeaves function described in the report for assignment 3, but additionally performs alpha and beta cutoffs according to the alpha-beta pruning algorithm. The advantage of using heuristics calculating a score for a given move on a given position, instead of calculating the position after the given move on the given position, is, as for MiniMaxLeaves compared to the standard MiniMax implementation, an improvement of computation time.

Total computation time distribution in percent			
Map Name	MiniMax	alpha-beta pruning	improved move sorting
50_50_8_25_rnd_1	78,9	16,75	2,35
2025_g6_3_competetive	70,25	21,9	5,57
standard map	29,2	19,33	9,65
2025_g6_2_pikatchu	83,28	11,31	4,53

Figure 1: The shares were calculated during a single game on the specified map, all three move finding algorithms were called on the same map position.

To test this algorithm, we decided to let MiniMax, alpha-pruning and this algorithm run in every move on the four maps seen in 1, giving each algorithm the same depth restriction. We made sure that all three algorithms came up with a move having the same evaluation for the given depth (the moves do not have to be equivalent, as the sorting of moves may result in a different, but equally strong move to be discovered first). The resulting distribution was calculated using the

IntelliJ Profiler. The improved move sorting algorithm manages to find a move in at most half the computation time that alpha-beta pruning needs. On larger maps, offering a lot of moves like the 50_50_8_25_rnd_1 map, advanced move sorting performs even better, as the amount of leaves that need to be evaluated takes up a larger part of the evaluation tree. Overall, the improved move sorting algorithm is the fastest algorithm yet implemented for finding the best move in a given position in the game.

1.3 Analyzing the weaknesses

In order to find the next important steps needed to speed up the computation time, we used the IntelliJ Profiler to get a deep insight into the improved move sorting algorithms computation time distribution. The distribution is visualized in 2. An arrow between two functions, represented by blue rectangles, expresses the lower function was called during the execution of the upper function. The percentages above the functions represent the share of the total computation time in which the function was executed. The executeIfpossible function, which takes a move and a position and tries to calculate the resulting position, takes up a major part of the computation time. The moveSortLeaves function only calls it when an inversion tile or a choice tile is captured. We will work on a more efficient way to calculate the heuristics after such a tile is captured. The cornerSideHeuristic is called by the moveSortLeaves function on every map position passed to the moveSortLeaves function as an argument. We will also try to minimize these function calls as the heuristic evaluation was already computed in the moveSortHelper function to enable move sorting, so each game position is currently evaluated twice.

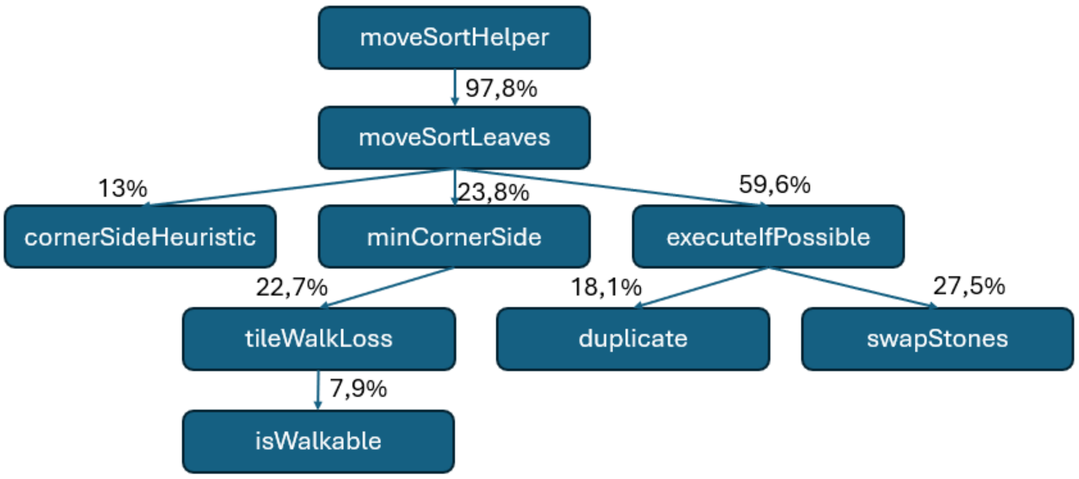

Figure 2: Total time distribution in the moveSortHelper function on the 50_50_8_25_rnd_1 map during a whole game.

Task 1

2.1 basic Idea

As it turns out, our idea about biased pruning as presented in the last report was viable. Here we will now begin to implement it. We first faced problems concerning the overhead being too inefficient, therefore making the move sorting so costly that there was no benefit to the improvement. In the later part of the development, we were able to increase the efficiency, therefore making it viable.

2.2 first implementation

2.2.1 primitive implementation

The first idea was to use the list of moves we iterate through in pruning and always use the one with the best rating by iterating through this list of moves and removing it from the List afterwards.

This was not efficient enough to make move sorting viable, as for the last depth of nodes, we would create an amount of additional states that needed checking, according to $O(n^2)$, where n is the number of these nodes.

$O(n)$ would not be a problem because the number of leaves is far higher than the number of nodes, because of the branching degree of the tree being quite high. In this primitive implementation, the Overhead created was significantly higher than the gain in efficiency.

2.2.2 first improvements

To solve this new problem with Overhead, we decided on a time memory trade off by saving the computed values of the nodes in a second List, with the indexes of the lists corresponding with each other. This reduced the amount of additional states to be checked to $O(n)$, but introduced a large overhead of filling and removing items from the Lists. This still made our Algorithm less efficient than pruning, even though fewer states were checked. At this point, the benefit of using the move sorting Algorithm slowly became tangible. But the large Overhead caused our move sort to be far less efficient as visualised in the Data in 2.3.2.

2.2.3 first testing results

Let us start with the positive results. The number of states visited in movesort is significantly lower than the number of states visited in pruning.

This can be visualised by imagining a $f(x) = x$ function inside the following plot. Every outcome above this line is a case in which movesort visited fewer states than pruning. (These values are computed at depth 5)

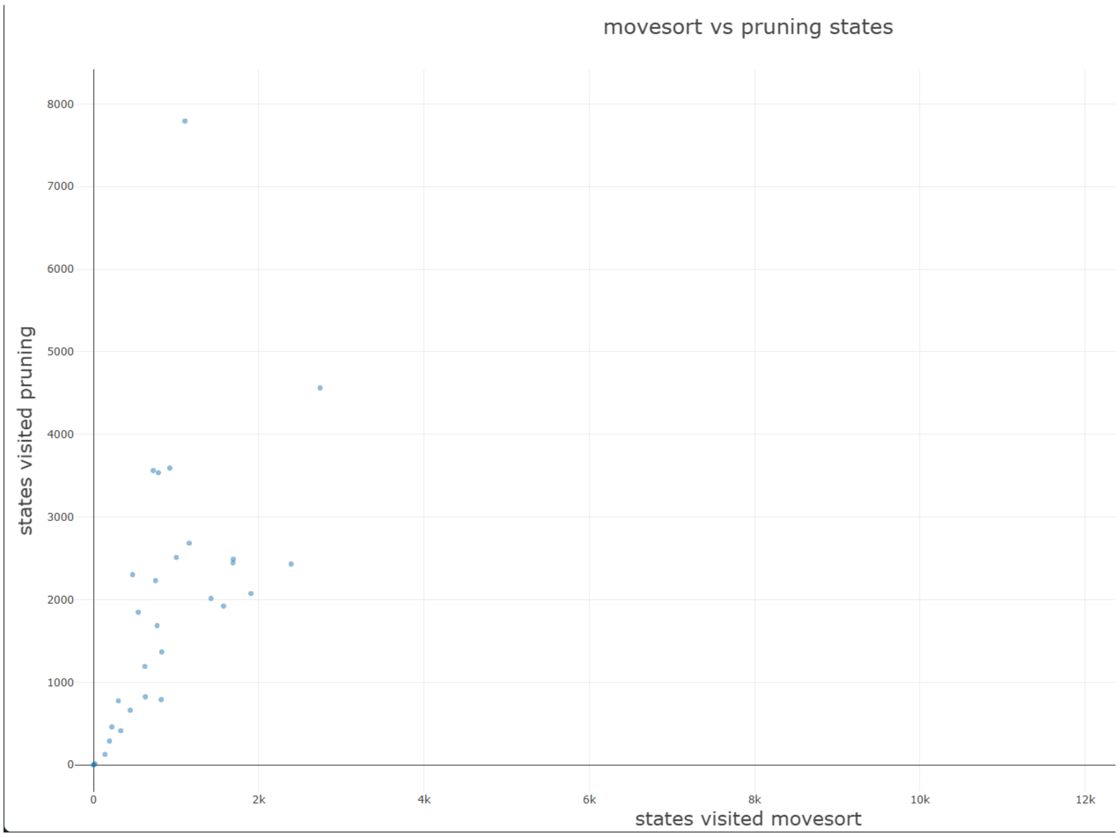

Figure 3: The number of states checked in a move in movesort compared directly against the number of states checked by pruning

But as mentioned in 2.2.2, even though the number of states visited decreased, the overhead increased.

This caused a decrease in efficiency over the board by a very significant margin. This is visualized in the following plot.

We can again visualize our partitioning function. This shows that even though fewer states are computed, the overhead is still far too high.

(These values are computed at depth 5)

The relative length of the axis is a little skewed. This is done for visualisation purposes and does not change the result nor the perceived result.

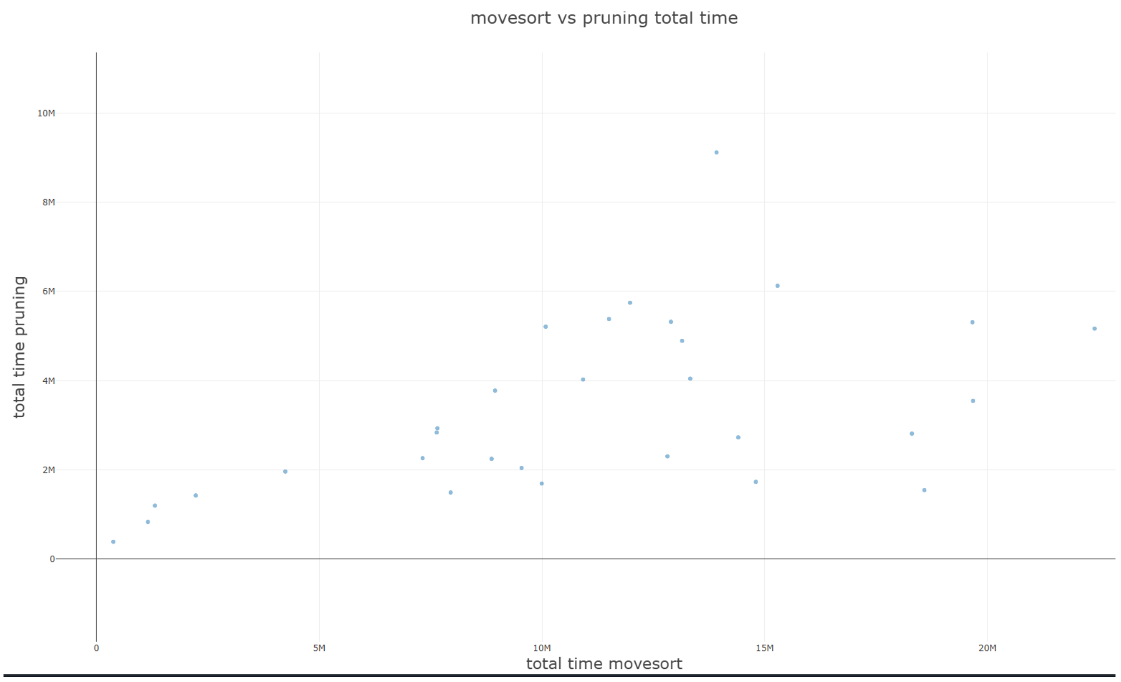

Figure 4: The total time needed to find a move with move sorting compared directly against the time needed with pruning.

2.3 streamlinging efficiency

After much deliberation, we concluded that the large overhead is based on inefficient "sorting Algorithms" on our part.

And as we already use Lists with the already implemented sortable property, we decided to stop manually coding the sorting of the list and instead use the already optimised list sort operation.

To make this even more efficient, we decided to store the result of the heuristic in the same list as the moves, so we do not have to juggle two lists at the same time.

After streamlining the efficiency of the basics, we started to improve move sorting on the basis of concepts we had already explored on the most efficient minmax variation in the previous assignment.

This is explored further in 1.2.

Task 2

3.1 Difficulties

This task required us to test on a multitude of different maps, in doing so we got to confirm a problem that has been plaguing us on Matchpoint as well; On big maps with many players our ai performs well, often we are in a winning position until the bombing phase comes, in the bombing phase itself we then time out and get disqualified. Having now identified the exact root of the problem by measuring the number of states (we look at way too many in the bombing phase), we will hopefully be able to fix this issue by limiting the depth whenever we enter the bombing phase.

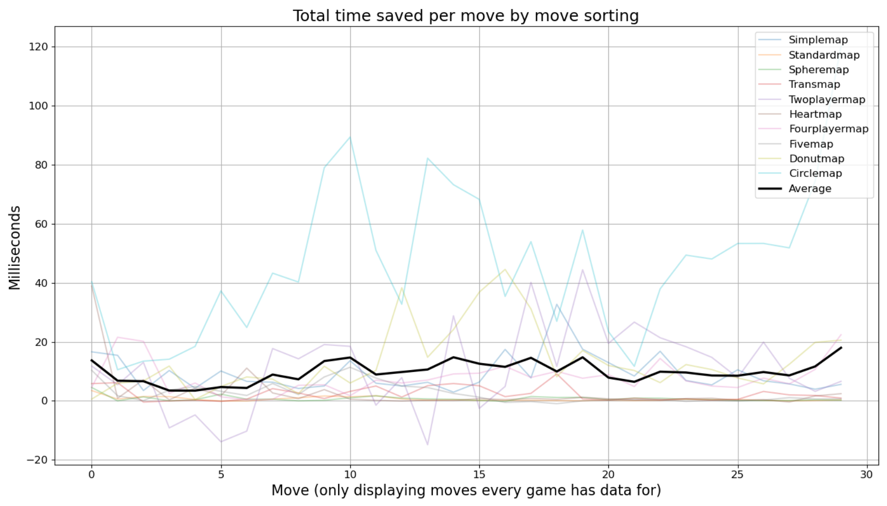

Figure 5: Total time saved by move sorting with depth 3 on various maps in comparison to pruning without move sorting.

3.2 Move sorting - was it worth it?

It is safe to say that move sorting has delivered major efficiency improvements, as can be seen in the figures above. Most important of all we can observe that move sorting saves around 10 milliseconds on average with every move played, this value increases dramatically with the size of the map: Sadly due to previously mentioned difficulties we were not able to collect data on very large maps and had to stay with smaller ones, but even with less variety a pattern is apparent. All the maps used are from the Matchpoint, and when looking them up you will notice that the maps with the largest amount of time saved are also the largest in size

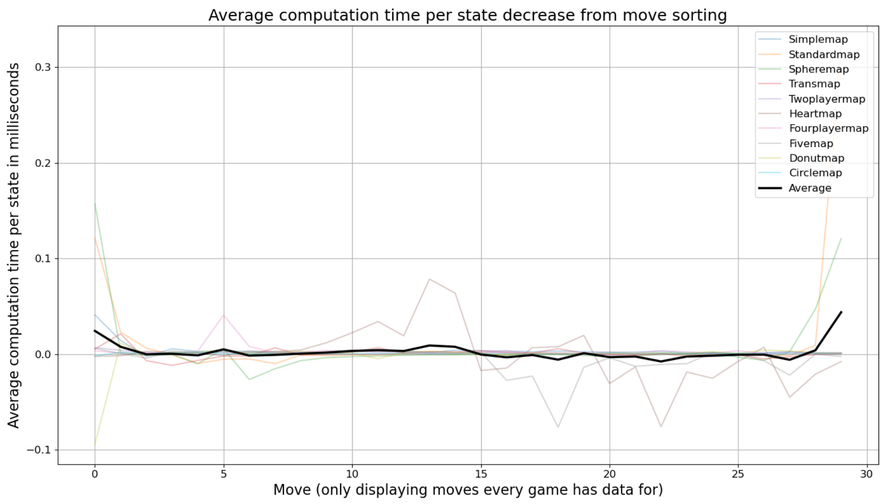

Figure 6: Time per state difference caused by move sorting with depth 3 on various maps in comparison to pruning without move sorting.

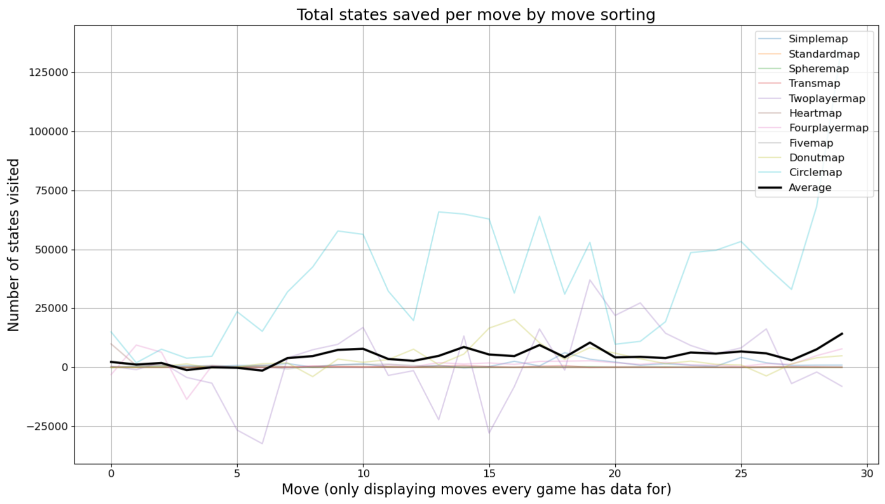

Figure 7: Decrease in the number of states visited caused by move sorting with depth 3 on various maps in comparison to pruning without move sorting.

(Simplemap) while the maps with the least amount of time saved are the ones that are the smallest in size (Spheremap, Standardmap). This, of course, poses the question of why that is the case, this question can be answered by the figure on the amount of states saved by move sorting. As we can see, the large variation in time gain seems to originate from a large variation in states saved. It makes sense that move sorting saves more states on large maps since there simply are a lot more states. With a large map usually come more possible moves, which then have an exponential impact on the number of states visited. Lastly it can be observed that there are no major gains or losses in the computation time per state; this is to be expected since we did not make the computation itself faster, we just made sure we had to do less of it. The outliers on either side can be explained by a low number of states overall, therefore small changes based on chance are immediately visible.

3.3 What is the ideal sorting?

Move sorting aims to increase efficiency by reducing the number of states visited. This is done by increasing the amount of pruning, which in turn can be achieved by looking at the most promising branches first, since those will not be pruned anyway. It is also important to underline that the most promising move is different depending on the perspective; if the minimum evaluation of the next set of states will be picked, then the state that is expected to deliver the "worst" rating should be visited first. At this point, it is important to define more clearly what is meant by rating or a "promising" move, even though it is true that sorting by how good/bad the moves are can be a good strategy, it is of importance to understand that it is the output of the heuristics in use that determines wether the sorting is good or not - for example we could have heuristics that don't look at the promise of a position but rather at wether the amount of stones under the players control are odd or even, in this case sorting by how good the moves are would be a bad idea, instead we could achieve a high amount of pruning by using those same heuristics to sort through game states. This is because the sorting's use is to predict which branches the algorithm will choose, not which branches are the best choice. Having understood this, it is easy to see that the ideal sorting will sort the different game states by how highly the algorithm will rate them later.

3.4 Our sorting vs ideal sorting

Currently, we sort based on the same Heuristics used to later calculate the ratings. This, of course, leads to 100 Percent accuracy on depth 1, or in that sense, the sorting for every node with children that are leafs is perfect. This is easy to see since in that case the calculated rating is the same as the rating we use to sort the

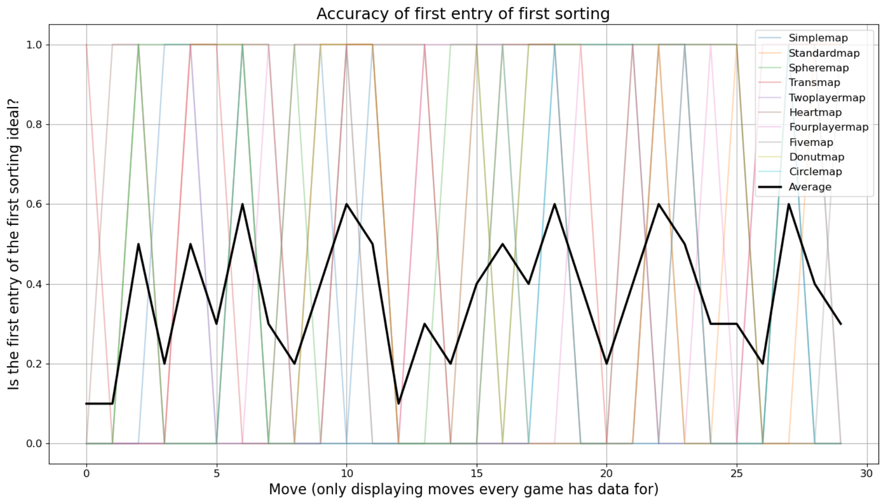

Figure 8: Accuracy of move sorting with depth 3 on various maps in comparison to pruning without move sorting.

individual states. Where it gets interesting is when we go to deeper depths; in this case, we used depth 3 for all tests. Measuring the quality of our sorting was quite a challenge, since, of course, when we first sort, we do not know the ideal sorting. Additionally, we were challenged by the sheer number of states that are visited, compiling an ideal sorting in retrospect for all of them would be a very complex endeavor, Subsequently, we decided to simplify: Since all positions in the move list are calculated using the same Algorithm it is reasonable to assume that the likelihood of each of them being correct is roughly the same. Also, we do the same on every node, so we can simplify here as well by just looking at one to get the accuracy for the whole depth. Of course, with increasing depth, the accuracy will decrease since more and more possible changes are introduced into the mix. Using these simplifications, we arrive at one truth value per move as seen in the Figure above: When the value is one, it means that the first entry of the first sorting on a specific move was the same game state that was returned by the algorithm. This measures the difference from the ideal sorting since we know that the first entry of the first sorting would ideally always equal the returned move. When taking the average value of the average line, we therefore get the accuracy: Over all ten maps tested, there was a 36.7 Percent chance that the first entry was right. Of course, this does not mean that all subsequent values would have been correct as well, so the overall accuracy is significantly lower, but this can act as a future reference point to decide whether a change was good or bad.

Task 3

4.1 Current implementation

The current iterative deepening search algorithm function is a very simple implementation. It gets called with a time limit and depth limit as arguments. First, the global variable deadline is set to the systems current time in nanoseconds plus the given time limit minus a puffer of 50 ms. A for loop is then used to call the desired algorithm (works for MiniMax, alpha-beta pruning, and move sorting), the depth limit for the algorithm is set by the for loops counting variable, which is increased by one on every iteration until the depth limit given by the server is reached. In each iteration, after a move is found, the function checks if the deadline is surpassed by calling javas System.nanoTime function. If so, the returned move is discarded, and the best move found in the previous iteration is returned. Otherwise, the found move is stored as the current best move, and the for loop enters the next iteration. This implementation assumes that the algorithm responsible for finding a move always terminates on depth limit one and is not interrupted by a time limit, which is the case for all three algorithms on every map. To ensure that the selected algorithm ends on time after the deadline is exceeded during execution, in every recursive call of a function in MiniMax, alpha-beta pruning and move sorting the systems current time in nanoseconds is checked. If it is exceeded, the function instance instantly returns zero.

Time spent after the deadline in milliseconds		
Map Name	Average time left	Minimal time left
50_50_8_25_rnd_1	34,5	29,5
2025_g6_3_competetive	47,72	41,9
standard map	49,61	49
2025_g6_2_pikatchu	43,48	23,5

Figure 9: The times were measured during one sample game on the specified map. The "Average time left" column shows the average time the moveSort function had left until the server would have disqualified the client. The "Minimal time left" column shows the minimal time the client had left until a timeout.

4.2 Testing the puffer

As seen in 9, we tested the iterative deepening search algorithm by measuring the time the function had left after the deadline was exceeded during execution and the algorithm responsible for finding a move was terminated. The tests were carried out using the move sorting algorithm and a puffer of 50 milliseconds. The resulting times with MiniMax and alpha-beta pruning are similar. The most important results can be found in the "Minimal time left" column, a value smaller than zero would result in a disqualification. The puffer of 50 milliseconds successfully prevented any timeouts, as at least 23,5 milliseconds were left. Furthermore, all average left times are above 30 milliseconds, leaving enough room for possible lags. The worse average times for larger maps can be explained by the larger number of possible moves, as in function instances entered before the deadline, all possible positions are still calculated. The puffer of 50 milliseconds also proved sufficient on Matchpoint, as several sessions were completed without a timeout.

4.3 Planned improvements

The current iterative deepening algorithm cannot make use of the results of the interrupted iteration of the move calculation algorithm, as the interrupted function instances return zero and the resulting move often is not a strong move. This is especially wasteful, as for most executions, the bulk of the total execution time is spent on the move calculation for the highest reached depth until the time limit is reached. For the next assignment, we plan to implement an advanced iterative deepening algorithm, which also takes the move of the interrupted iteration into consideration, checks whether it is stronger than the current strongest move, and therefore minimizes the time spent on calculations whose results are not used.

Task 4

As documentation tool of our project we chose javadoc. The documentation can be automatically created by executing the command "gradle documentation" in the folder "group6". The generated html files are stored in the "documentation" folder.