Softwarepraktikum SS 2025
Assignment 1

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

The tasks were split fairly on to all group members, Frithjof worked on task 1, while Malte implemented alpha-beta pruning required for task 2 and Henri developed an measurement system for task 3. The corresponding section in the report is written by the member who worked on the task. Additionally, the contents described in section 1.2, 1.3 and 1.4 were implemented by Frithjof, the changes to heuristics explained in section 1.5 were carried out by Malte.

1.2 More efficient bombing algorithm

1.2.1 Method

The new bombing algorithm makes use of two Arraylists. In the nth iteration of the algorithm, the Arraylist currentTiles stores the tiles, which can be reached in n steps from the starting tile. In iteration 0 only the tile where the bomb is placed is contained in currentTiles. During each iteration, the tiles that are not already a hole, directly neighboring any tile contained in currentTiles are added to the second Arralist, nextTiles. Furthermore, these tiles are also set to be holes. After each iteration, currentTiles is set to nextTiles and the nextTiles Arraylist is emptied. The number of iterations is set to the bomb strength of the map.

1.2.2 Comparison

Average time per bomb in Nanoseconds		
bomb strength	New algorithm	Old algorithm
1	25441	19250
2	49633	64583
3	60516	6555156
4	41991	4937100
5	51216	24025208

Figure 1: Comparison of the bombing algorithms

The measurements seen in 1 were calculated by placing 15 bombs on random map tiles and measuring the average time per bomb that it takes both algorithms to calculate the new map. The lower average bomb times for the bomb strengths of 4 and 5 of the new algorithm can be attributed to the higher probability of randomly selected bomb tiles already being a hole. The same accounts for the average time per bomb of the old algorithm at bomb strength 4. It can be seen that the new algorithm especially shows better performance at bomb strengths greater than 2, as only the worst-case performance takes exponentially many steps compared to n,where n is the bomb strength, and the usual performance is better than that. The better performance of the old algorithm for bomb strength 1 is caused by the comparatively costly creation of two ArrayLists. Overall, the new algorithm proves to be a great performance improvement, especially for stronger bombs, and is now in use to calculate new positions.

1.3 Array versus Hashmap

1.3.1 Measurements

As of the last assignment, a Hashmap was used instead of an Array to store the owners of each tile. To test the efficiency of both data structures, two games on the competetive map of our group were played, one with each data structure. In each game, the three different MiniMax algorithms were run separately, and the average time per state of the three algorithms throughout the game was measured. The games were checked to be exactly the same.

Average time per state in Milliseconds			
data structure	MiniMaxZwei	MiniMaxDrei	MiniMaxVier
Array	3,01	2,341	1,58
Hashmap	23,23	30,25	6,72

Figure 2: Performance comparison of Hashmap and Array

1.3.2 Measurements

The test described in 2 revealed that the Hashmap has an immense impact on the performance of our algorithms, since for each map access the hashfunction must be calculated. Because of this, in the subsequent implementation, an array was used.

1.4 The gameState class

To efficiently store a game position, we implemented a gameState class. It stores the amount of bombs and overwrites of every player in an Integer Array and the owners of every tile in a doubly nested Byte Array. A number between 1 and the maximum number of players indicates that the tile is occupied by the player with the corresponding player number. A 0 indicates that the tile is not currently occupied. A number between 10 and 14 indicates that the tile is a hole (the number is 10), a choice tile (the number is 11), a bonus tile (the number is 12), an inversion tile (the number is 13), or an expansion tile (the number is 14). Furthermore, this class offers simple getters and setters, a duplicate method returning a new instance of the gameState class, containing the exact same values, and a printMap function printing the map in the terminal.

1.5 static vs relative Heuristics

After a lot of deliberation on how to make our Heuristics more efficient in both computation time and move quality, we came to three fundamentally different approaches.

1.5.1 static Heuristics

These Heuristics are cross-dependence-free, so different stones, for example, do not impact each other's heuristic values. This group of heuristics is, in our case, comprised of the item heuristic, the count all heuristic, and the least possible heuristic. Also in this group of heuristics is the corner/side heuristic, after a values map is computed at the start of the game. This is the case as the heuristic works with static values. The advantages of static heuristics are mainly on the efficiency side, as "static computing" is much faster than "relative computing". This can be seen in the next part.

1.5.2 dynamic Heuristics

This group of heuristics focuses on the interdependence of stones on the board. One example where this is relevant would be our upgraded corner side heuristic. A stone directly next to a corner opens the corner up for capture by an opponent. Because of this, we rate it negatively. But if the corner is held by us, suddenly these stones become very strong. The reason for that lies in the "ideal" game idea, not including bombs, overwrites, and inversions/swaps. In this ideal game would attain control over a corner and afterwards propagate from this corner, thus getting a connected mass of stones, which can not be captured. This change in the significance of tiles can not be addressed statically. Thus, to increase the quality of our moves, a dynamic heuristic has to be created.

1.5.3 first idea

Our first idea was to simply use a static heuristic, for example, corner/side, and recalibrate the values of the tiles based on the current game situation. But as it turns out, there was a reason why we started with static heuristics in the first place. The efficiency of the heuristics tanked to an unacceptable degree. So we have the contrary situation to before, where the move quality is high, but the efficiency is low. So we tried to find a compromise between the two.

1.5.4 quality efficiency trade-off

Now we have decided to engineer a heuristic that on one hand behaves statically inside a MiniMax Algorithm. While on the other hand allowing us to adjust to the changing situation on the board. Our solution was to use the same approach as in the corner/side heuristic to compute the values for the tiles at the start of the match. But instead of only computing them at the start of the game, we recalibrate them at the start of the move-finding process. Our new Algorithm works by viewing corners captured by us as holes, as they share the same properties in relation to our heuristic. This causes the corners to "propagate" inwards, in theory, then promoting moves in line with our perfect game. As a trade-off happens, there are of course negatives. The negative in this case is that changes to the game state in between depth iterations do not influence the heuristic. This lowers the move quality. But still, this kind of quality efficiency trade-off is exactly what we need if we want to use dynamic heuristics.

1.5.5 move delta heuristics

Another very efficient idea is to not compute the resulting states from moves to then analyze them, but rather to analyze the impact of the move using a heuristic. This approach will be explored later in Task 1. It is currently a quality efficiency trade-off, as we use heuristics with even less information. This works great for static heuristics, like most stones. But it proves rather difficult to implement dynamic ideas. We will have to conclude on this issue after further exploration of the idea. But if we find a way to do something similar to section 1.5.4. We would be able to implement dynamic heuristics in this much more efficient Algorithm. These heuristics are used by MiniMaxVier.

Task 1

2.1 Efficiency problems

After the first tests of MiniMax functions, it turned out that efficiency would be a problem for any further MiniMax implementations. In particular, the large maps, with a width and height of 50 tiles, posed difficulties for simple algorithms, even on a depth limit of 2. Therefore we implemented 3 different algorithms, which will be explained and compared in the following sections.

2.1.1 MiniMaxZwei

This function is the basic implementation and provides the foundation for the other MiniMax implementations. The function parameters are the depth limit, the gameState to evaluate, the number of the player whose turn it is, and the game phase. The function returns a double array of length 2 containing the evaluation of the given gamestate and the move which leads to this evaluation. The function works using recursion. First, it checks if the passed depth limit is 0, and, if so, returns the heuristic evaluation for the passed gameState. Else, a list of possible moves for the given player in the given gameState is calculated using the getListOfMoves function. If the list is not empty, each move is evaluated by recursively calling this MiniMaxfunction. As parameters, the current depth limit decreased by 1, the gameState after the corresponding move, and the players whose turn is next are used. The chosen move to return depends on the player whose turn it is; if we are to play, we maximize the evaluation and return the move which leads to the highest heuristic score; if the enemy is to play, we minimize the evaluation and return the move which leads to the lowest heuristic score. If the list of possible moves is empty, the lists of possible moves for all participating players are computed. If not all lists are empty, this MiniMax function is recursively called with the exact same parameters as the current function instance, except with the player number of the first player who has a legal move ordered by their next turn. If no player has a legal move and the game is in the build phase, we enter the elimination phase, so this MiniMax function is recursively called with the exact same parameters as the current function instance, except that the game phase is set to elimination phase. Otherwise, the game is already in the elimination phase and has now ended, then the heuristic score of the given gameState and an empty move is returned.

2.1.2 MiniMaxDrei

The MinMaxDrei and MiniMaxVier functions are called from a helper function called MiniMaxHelper, which calls the MiniMax functions on every possible gameState after one move of our ai. It returns the move that achieves the highest evaluation. Therefore, the MiniMaxDrei and MiniMaxVier functions do not need to return the corresponding moves and only return the evaluation for a given gameState. The parameters of MiniMaxDrei and MiniMaxVier are equivalent to the parameters of MiniMaxZwei. The major difference between MiniMaxZwei and MiniMaxDrei is the computation of valid sucessor gameStates. While MiniMaxZwei uses the getListOfMoves and calculatePos functions to find all sucessors, MiniMaxDrei uses the executeIfPossible function explained in the next section. Apart from that the functions are similar.

2.1.3 ExecuteIfPossible

This function unites the isMoveValid and calculatePos functions. To check if a move is valid, the isMoveValid function calculates if the move captures enemy stones (except the game phase is elimination phase or the stone is placed on an expansion tile). The calculatePos function, which returns the gameState achieved through the gameState and the move given as parameters. Therefore, the calculatePos function again calculates which tiles are captured by a move. This redundancy is eliminated by the executeIfPossible function, which calculates the captured stones once for a given move and gameState, passed as parameters, and if a move is valid directly replaces the calculated captured stones. We compared the executeIfPossible function to the combination of isMoveValid and calculatePos by calculating the gameState after four random moves in 10 sample positions of the competetive map. The results in the table 3 demonstrate the improved efficiency of the excecuteIfPossible function as it takes 20% less computation time on average.

2.1.4 MiniMaxVier

The MiniMaxVier function works identical to MiniMaxDrei, but when called with a depth limit of 1 (meaning that the function only calculates one more move ahead), it calls the MiniMaxLeaves function. This function makes use of a special set of heuristics which do not evaluate a given gameState, but a move executed in a given gameState. It has one significant advantage compared to the other MiniMax implementations: The final gameState does not have to be computed; instead, the

Average time per Position in Nanoseconds		
Position	isMoveValid + calculatePos	executeIfPossible
1	3100	4225
2	2775	1950
3	2925	2025
4	2625	1825
5	3125	1975
6	2750	1925
7	2450	1100
8	2575	1575
9	3325	1725
10	3175	4500
Average	2882,5	2282,5

Figure 3: Comparison of executeIfPossible and isMoveValid with calculatePos

heuristics receives the previous gameState and the move the player would play as arguments. The heuristic then returns the difference the move would cause in the gameStates heuristic score. After the MiniMaxLeaves function evaluated all possible moves for the gameState and player it receives as parameter, it checks whether the player is an enemy, whereupon it saves the worst resulting heuristic change; else, the best heuristic change is stored. This change is then added to the heuristic score of the gameState passed as a parameter, which is calculated using a standard heuristic evaluating a gameState, and returned.

2.1.5 Comparison

Average time per state in μs			
Depth Limit	MiniMaxZwei	MiniMaxDrei	MiniMaxVier
2	20,86	20,33	21,34
3	3,42	4,13	2,00
4	5,25	6,96	3,8
5	3,88	5,60	3,55

Figure 4: Comparison of the MiniMax algorithms on "2025_g8_1_trans.MAP"

As seen in tables 4,5,6, and 7, we tested the three different algorithms on different maps. The time measurement shows the average time per state of each algorithm during the duration of the game (except for the times shown in 5, in which the games were cut after 10 moves, which already took half an hour). For

Average time per state in μs			
Depth Limit	MiniMaxZwei	MiniMaxDrei	MiniMaxVier
2	11,70	11,92	1,89

Figure 5: Comparison of the MiniMax algorithms on "50_50_8___25_rnd_1.MAP"

Average time per state in μs			
Depth Limit	MiniMaxZwei	MiniMaxDrei	MiniMaxVier
2	20,66	27,00	17,35
3	10,22	14,70	7,77

Figure 6: Comparison of the MiniMax algorithms on "2025_g6_2_pikatchu.MAP"

each move, all three algorithms were run, and it was ensured that all checked the same number of states. For each map, the maximum depth limit differs as on larger maps calculating at higher depths proved to be too time consuming. The most significant result of this comparison is the strong performance of MiniMaxVier, which outperforms MiniMaxZwei and MiniMaxDrei on all maps for most of the depths by quite a significant factor. Secondly, the poor performance of MiniMaxDrei catches the eye, as it performs worse than MiniMaxZwei for most of the measurements, although it was implemented as an improvement of MiniMaxZwei. We will ensure to tackle that issue and find the reasons responsible for MiniMaxDrei's poor efficiency, as the executeIfPossible function as seen in 3 should enable us to write a function more efficient than MiniMaxZwei. Third, in 5, an unusual gap between MiniMaxVier and the other algorithms shows the impact a map can have on the performance. On this map, more than a million states were checked on depthLimit 2, which indicates that around a thousand moves are possible in an average game position. This extremely favors MiniMaxVier, as it only computes a small fraction of the gameStates the other algorithms need to compute, as it only calculates the gameStates to depth limit minus one. In conclusion, the MiniMaxZwei algorithm is the most efficient algorithm using the standard heuristics, but we did compare MiniMaxDrei to alpha-beta pruning in task 3, as it was the first algorithm, which returned moves equaled alpha-beta prunings returned moves. The MiniMaxVier algorithm is used on Matchpoint and proves to be the most interesting algorithm for further development, but we will need to develop more heuristics compatible with this algorithm.

Average time per state in μs			
Depth Limit	MiniMaxZwei	MiniMaxDrei	MiniMaxVier
2	11,96	10,32	6,48
3	5,24	6,07	3,09
4	3,30	4,04	2,50
5	2,03	3,5	2,3
6	3,02	4,08	2,54
7	3,18	3,75	2,11

Figure 7: Comparison of the MiniMax algorithms on standard map

Task 2

alpha beta pruning

After the depth was raised to 3, we started encountering Timeouts. This established the need for either a more efficient program or a more efficient Algorithm. Trying to maximize the efficiency of the program yielded good results, but it was not in itself enough to get us inside the time bounds. This is explored in great detail in Task 1.

4.0.1 alpha beta pruning

As explained in the last meeting, it is enough to calculate the result of the enemy's moves until the moves "invalidate" the move in accordance with the heuristic used. I will take the information given to us last meeting as a given and will not repeat it to preserve your time.

4.0.2 our implementation

We decided to use MiniMaxDrei as the basis for our pruning Algorithm. In retrospect, we should have used MiniMaxZwei or MiniMaxVier. We will implement pruning on the templates of MiniMaxZwei and 4 at a later point in time, and then we can repeat the comparison made in Task 1 for the different pruning Algorithms. The results should be relatively similar, but as the cost has already been sunk, we should try to capitalize on it. MiniMaxDrei is a recursive function that results in a tree similar to the one shown in the last meeting. So we simply carry the best value of the already explored branches as alpha and check enemy moves against this best value. If it is equal to or lower, we abort the branch. This equal to or lower is a very important detail as the used MiniMax Algorithm also has to use equal to or lower, else the results will not be the same. Equally, we check our own moves compared to the beta value, if its equal or higher, we cut off the branch. The current alpha and beta values are passed as additional parameters. The starting alpha and beta values are set to -10000 and 10000, as these numbers will not be surpassed by any heuristic score.

4.0.3 compare Flag

To check that the result of our pruning is correct, we compared it with the result of the base MiniMax Algorithm. For this, we used the optional "-c" Flag in our program.

4.1 first conclusion

As the result of pruning is the same as the result of the MiniMax Algorithm, alpha-beta pruning results in a comprehensive increase in Efficiency. This makes the MiniMax Algorithm obsolete in most cases. This conclusion is backed by testing in 5.6.

4.2 biased alpha beta pruning idea

As alpha-beta pruning increases in efficiency, if a good move is found quickly, we might be able to bias the search so that the average efficiency increases. If we were to calculate the heuristics of the states resulting from our moves, we could first explore the branch with the most promise, hopefully finding a good move quickly. This would allow us to prune a lot of moves. This would remove the comparability of the Algorithm to the MiniMax Algorithm, as they do not deterministically result in the same move. But this does not make pruning worse, as if we do not know which of for example, four moves with the same heuristic value is better, it does not matter if we choose the first one of those moves as in MiniMax or a random one.

There are other, more significant disadvantages. First off, the overhead. We have to calculate more heuristics. Currently, this overhead is not significant as the share of time our heuristics need is very low relative to the amount of time our move-finding Algorithm needs. But this will change in the future when our Algorithms get more optimized and thereby open up more time for the heuristics. Another disadvantage is that the worst case gets worse, which is the most dangerous case, as this could lead to a disqualification under unlucky circumstances, which would cause us to need more safety reducing the usable time. So, on average, this biased pruning might bring advantages and thereby merits testing, but it is in no way comprehensively better. So, whether or not it is worth implementing will depend on the test results.

4.3 further pruning ideas

Building on the idea of pruning, we revisited Overrides and their significance. Our current working theory is: Overrides should be saved to the end of the game, because they enable moves, when usually you would just pass. This gives the Override the value of an extra turn. Two average turns are usually better than one good turn. The only problem are the corners, because are two average turns better than capturing an edge in the first half of the game? This factor changes a lot, because if we could attain that override moves are "worse" than normal moves if normal moves are possible, then we could prune every override move if any other move is possible. This would be an enormous increase in efficiency, because usually far more override moves are possible than normal moves. If our assumption stands, then this would be a simple increase in efficiency. But, rather likely, the quality of moves would suffer because at least in some cases, this is not the case. So, again, this would be a quality efficiency trade-off. There are, of course, compromises worth exploring. For example, if we do not find a heuristically good move, we could check overrides. We could also only check the overrides for corners.

Task 3

5.1 The Challenge

This Task required us to create many diagrams by collecting large amounts of data. The data collection process can be done simply by printing time values to the console whenever seems fit, but to ensure the quality and accuracy of the diagrams and make it easier to create them anytime we need in the future, we decided to implement a whole system dedicated to collecting, sorting, storing, and plotting data.

5.2 Architecture

The measurement system must fulfill several key requirements: It should be easy to use, modular, and upgradable. It also should deliver accurate measurements and store them in a way that enables plotting later down the line without much work. We used an architecture that consists of 2 classes, a measurement class and a DataManager class. The basic idea is that measurements can be created at any point we want to measure, and then these measurements are stored within the data structure of the DataManager. This Data structure should allow for different data pools to be collected at the same time (e.g., measuring performance for minimax and alpha-beta pruning at the same time). It also needs to be dynamic since the length of a game is unpredictable. Therefore, we settled on a nested List; We create one List consisting of as many lists as there are data "Buckets" we want to collect, these lists, on the other hand, consist of measurements that have been collected for that specific bucket. Said measurements are created with a bucket number, which then determines in which list they get added. The measurements themselves can be created in two ways: either we just give a value in the constructor, or we measure time. Measuring time is done using the methods start() and end() of a measurement object, with them, the Instant (data type for a point in time by Java) for the start time and end time are stored, and once the measurement has ended the difference between the two in nanoseconds is derived as the time value which will later be displayed in graphs. The end of a measurement is also the moment at which it is added to the nested list for later storage. This way of storage also ensures that the individual entries are sorted chronologically, since the measurement that ended first will be stored in the frontmost entry of the corresponding buckets list. Finally, when the game has ended, we call the generateCSV() method. This method parses the list into a time-stamped CSV file, since the data structure already resembles a table due to its nested nature, parsing it into CSV is rather straightforward.

5.3 Plots

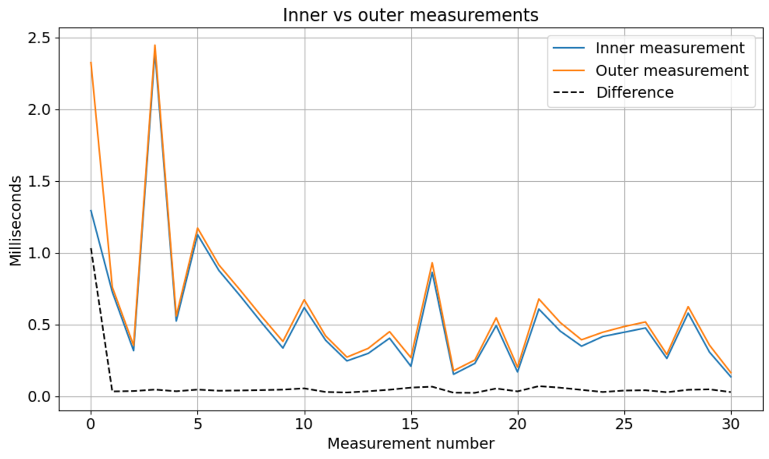
Once the CSV files are obtained, we can easily create plots using Python and the pandas library. One Specific feature we implemented here was to create a script that automatically generates a plot for every data bucket, so every column in the CSV file. This allows us to from now get a near instant visual representation of newly obtained measurements and while these generated plots will usually need some adjustments (labels, etc) to appear in a report such as this one, they are incredibly helpful when trying to get an understanding of the current measurements.

5.4 Why even bother?

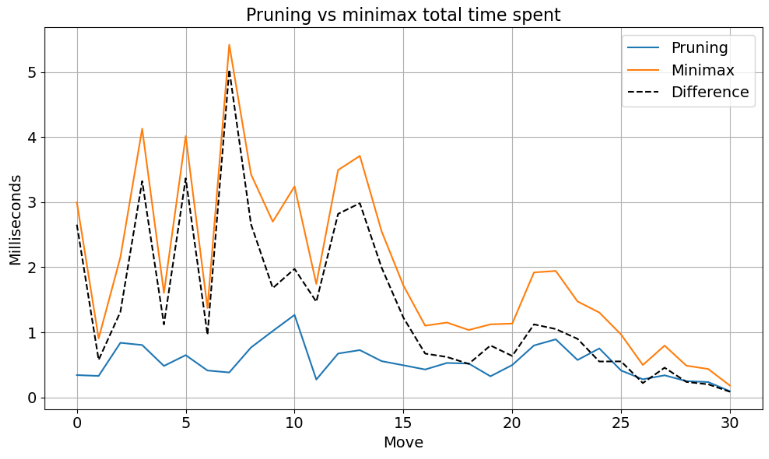
Our solution is Complex - Overcomplicated, one might say - but doing things this way has a few key advantages: Creating new measurements is incredibly easy! Because we decided to not only make the amount of measurements in a data bucket dynamic but also make the number of data buckets dynamic, one can just add a whole new bucket and fill it with measurements in just three lines of code. And even better: When that data isn't needed anymore, we can simply delete the measurement wherever we measured and do not have to worry about changing any hard-coded sizes or names, which might be the case in solutions using a fixed number of data buckets. Also, having a separate data type for measurements enables us to change it whenever we please. Let's say some time in the future we want to store something other than numbers, we can do this easily. The same applies to different units. Right now, we measure in nanoseconds to achieve maximum accuracy, but if our AI ends up taking so long in the future that nanoseconds would lead to an overflow of the long value, we have already implemented an option to use milliseconds instead. Lastly, we chose to use ArrayLists in our data structure, this choice was made because they can be indexed, so that we can get to a specific bucket, but also because they are very fast when appending values to the end of the structure which is all we do when adding measurements to their buckets.

5.5 Accuracy

When it comes to measurements, accuracy is everything. But what might seem straightforward turns out to present a whole new set of challenges. Since we store measurements in a separate data structure, which can get quite large depending on how long the game goes on, it is reasonable to assume that the overhead created by these operations might skew our results. Thankfully, we were able to work around this issue; We only start doing any operations in the data structure after the measurement has ended, and therefore the time value is already set, as for the writing to a csv file, we do this when the game has ended so that is not

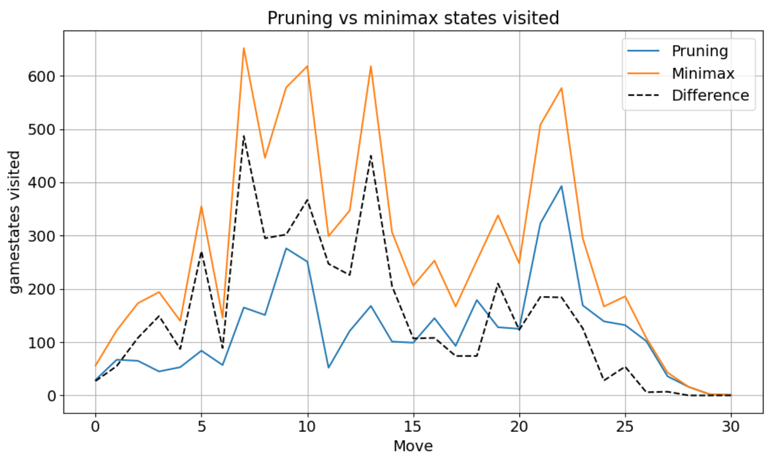


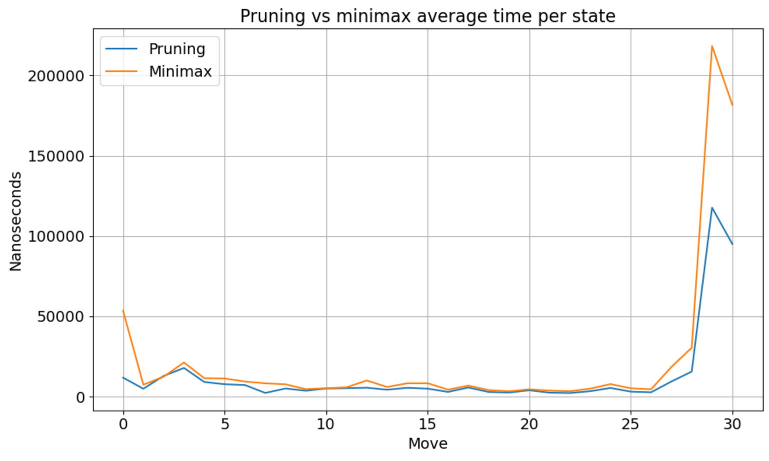
an issue either. But one last problem still stands; measurements might include other measurements, so that way we one measurement will be influenced by the overhead of the "inner" measurement. To ensure that this does not make the results useless, we plotted exactly this case (see figure above). This plot represents two Measurements that started and ended right next to each other in the source code, with one being the "inner" and one being the "outer" measurement. The dotted line below represents the difference between the measurements at any given point. As we can see, the difference between the two is around 0,1 milliseconds most of the time, which is a good sign because it means that the outcome is hardly different (as can be observed above). But there remains one outlier. We have tried to resolve this issue by initialising the Lists prematurely and getting rid of any initial computations made, still the overhead remains. Therefore, for now, we will have to refrain from using nested measurements to not falsify the results.



5.6 Comparison

With the new Measurement System, it has now become easy to measure and then plot anything we want, so let's now leverage our new power by comparing the two previously introduced algorithms, "pruning" and "minimax", on the standard map against the default AI. For this local test we used MiniMaxDrei, with MiniMaxZwei we might be able to deliver minimal performance improvements. As we can see in the figure above, the pruning Algorithm needs significantly less time per move than the MiniMax Algorithm. We can also see that the difference between the two is highest at the start of the game, around move 10, and decreases significantly in the end. This might be because the pruning Algorithm only discards a percentage of all states, and therefore, the performance gain is relative.





Let us investigate using another plot: In the figure above, we can see that the pruning Algorithm can discard a lot of states in comparison to the MiniMax

Algorithm. It also lines up that the difference of states visited is smaller, later in the game, but it does not seem like the decrease in difference is due to a smaller number of states overall. One possible reason for this behavior could be that when approaching the end of the game, our heuristics do not produce very distinct ratings (e.g, the possible moves are closer in quality) because high-value positions, etc, are already taken.

Lastly, when looking at the average time spent per state, we can observe a significant spike in the last moves. This is because when there are only a handful of states to evaluate, the method still has to run, and even though there are fewer recursive calls, the load of a single call has to be distributed over way fewer states.