Softwarepraktikum SS 2016
Assignment 2

Group - 3

Eric Wagner	344137	eric.michel.wagner@rwth-aachen.de
Stefan Rehbold	344768	Stefan.Rehbold@rwth-aachen.de
Sonja Skiba	331588	sonja.skiba@rwth-aachen.de

Task 1

For this task, we implemented the function

```
int32_t minimax(Map* map, int turn, int depth, int numberOfPlayers,
    bool isPlayingPhase);
```

which is executing minimax with paranoid strategy on a through *map* given board. The cut-off value is a given *depth*, which determines how many move-combinations are going to be evaluated. The *turn* (which players turn it is to make a move) needs to be known, as the node value shall be maximised for the own players turn, or else minimised. Also we need to know, in which phase the game currently is, playing or bombing phase, so that the corresponding evaluation function is used. A node represents a game state and multiple nodes may have the same game state. The children of a node are nodes, that result from their parents game state by a move from the current drawing player. That way a player's move is represented by the branches leaving a specific node. The board will be evaluated at the leaves of the game tree.

If the algorithms depth is 0, a leaf is reached. The state will then be analysed and the evaluation value returned. If the depth is yet 1 or greater, the function iterates through the map and goes deeper for every possible move from the current state of the board by calling

```
minimax(&mapCopy, (turn%numberOfPlayers)+1, depth-1, numberOfPlayers,
     true);
minimax(&mapCopy, (turn%numberOfPlayers)+1, depth-1, numberOfPlayers,
     false);
```

The first gets called in the playing phase and the second one in the bombing phase.

In the playing phase:

- For a standard move, this is called once, for the move itself.
- For a move on a tile 'b', a bonus tile, minimax is called for both choices, which are choosing a bomb or an override stone.
- For a move on a tile 'c', a choice tile, minimax is called for all possible choices, which are switching the stones with any of the participants, including himself.

In the bombing phase the function is simply called once per valid bomb move.

Task 2

For this task, we implemented the function

```
int32_t alphabeta(Map* map, int turn, int depth, bool isPlayingPhase,
    int alpha, int beta);
```

which is executing Alpha-Beta pruning, which prunes the “irrelevant” branches of the search tree. Most of the variables are the same as those of the MiniMax algorithm. The cut-off value is a given *depth*, which determines how many move-combinations are going to be evaluated as a upper bound. The *turn* (which players turn it is to make a move) needs to be known, as the node value shall be maximised for the own players turn, or else minimised. Also we need to know, which phase the game currently is in, which can be playing or bombing phase, so that the corresponding evaluation function is used. Newly added variables are the *alpha* and *beta* values which describe the worst case scenario for the player.
A node represents a game state and multiple nodes may have the same game state. The children of a node are nodes, that result from their parents game state by a move from the current drawing player. That way a player’s move is represented by the branches leaving a specific node. The board will be evaluated at some of the leaves of the game tree. Leaves will not get evaluated if a sister or brother leaf was already evaluated with a bigger value then *alpha*. *alpha* is called the first time with the minimal value possible and *beta* with the maximum value possible of the variable type, in this case int.

If the algorithms depth is 0, a leaf is reached. The state will then be analysed and the evaluation value returned. If the depth is yet 1 or greater, the function iterates through the map and goes deeper. In a maximizing case, the evaluated score gets compared with *alpha* and becomes the new *alpha* if it is higher. In a minimizing case, the evaluated score gets compared with *beta* and becomes the new *beta* if it is lower. If *alpha* is higher or equal to *beta* the rest of the possible moves of the current state of the board won’t get evaluated. Otherwise for every possible move from the current state of the board by calling

```
alphabeta(&mapCopy,(turn%numberOfPlayers)+1 ,depth-1, true, alpha,
    beta);
alphabeta(&mapCopy,(turn%numberOfPlayers)+1 ,depth-1, false, alpha,
    beta);
```

The first gets called in the playing phase and the second one in the bombing phase.

Task 3

What did we implement?

As the benchmark test are our only way to determine how efficient our program is running we implemented a few more metrics than necessary. We tried to keep the AI and the benchmarking as far separated as possible. To achieve this we adapted our makefile to produce 2 different binaries and try to separate them even more (see next section).
Our first two tests are only there to rate the speed of our evaluation function. We can input a map and a time limit and we get as output the amount of evaluation our program could execute in the give time limit.
This can be used when we revisit our evaluation functions. The first use is to see if a optimization does really increase the speed and a second one is to control that the evaluations do not become too slow when we implement other rating heuristics.

The second set of tests is the actual assignment and will analyse our tree search algorithms. For these we can input a map and a depth. As result we get metrics:

- The amount of visited nodes
- The time taken by the algorithm to find a move
- The average amount of evaluations done per second
- The time spend evaluating nodes
- Percentage of time spend evaluating the leafs

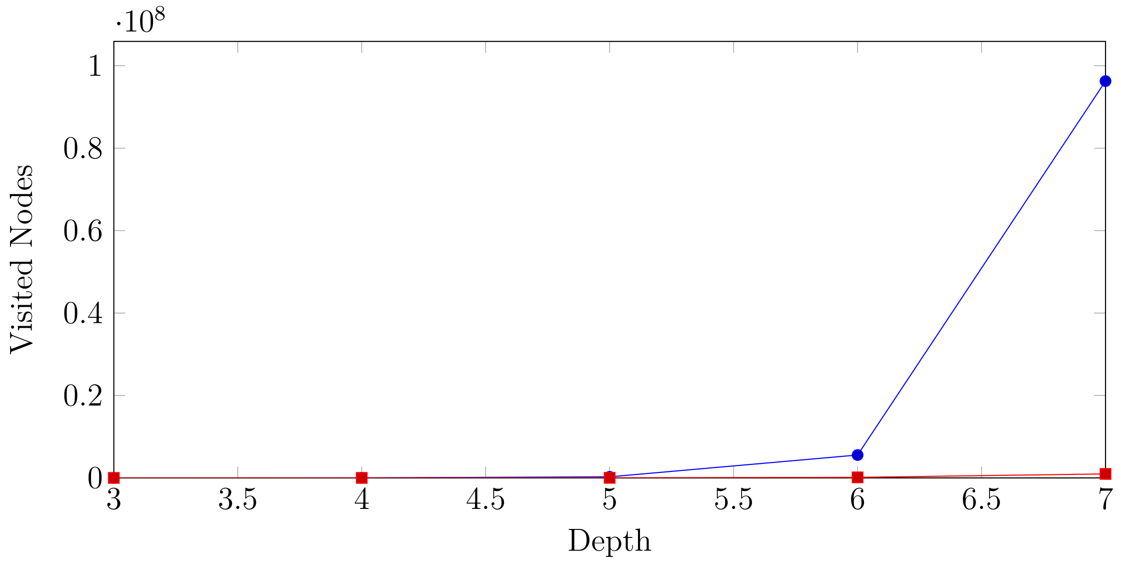
With these parameters we can see where the bottleneck of our system is. For example did it help use to decide to use the paranoid search strategy as we were spending over 99% of our time evaluating leafs for max^n.

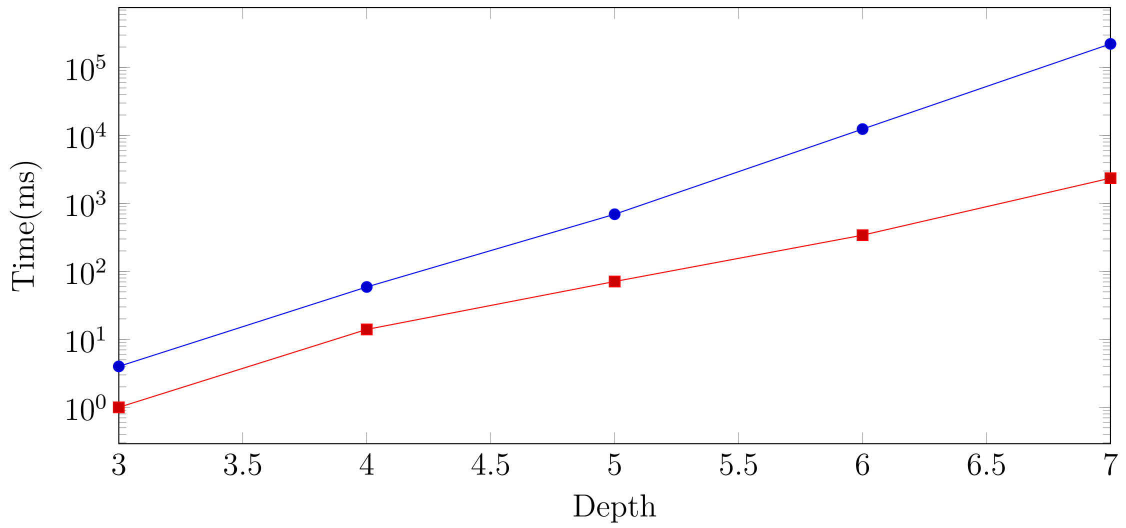
What remains to be solved?

For now, our only concern is that the frequent calling of the timer create an noticeable overhead that is really not necessary for our main program. After all, we save the time at the beginning and end of every leaf evaluation and need to keep track of a counter for the amount of nodes. Our current plans to avoid this, is to search for compiler instruction to only compile those lines of code for out benchmarking project, but not for the AI.

How are our results?

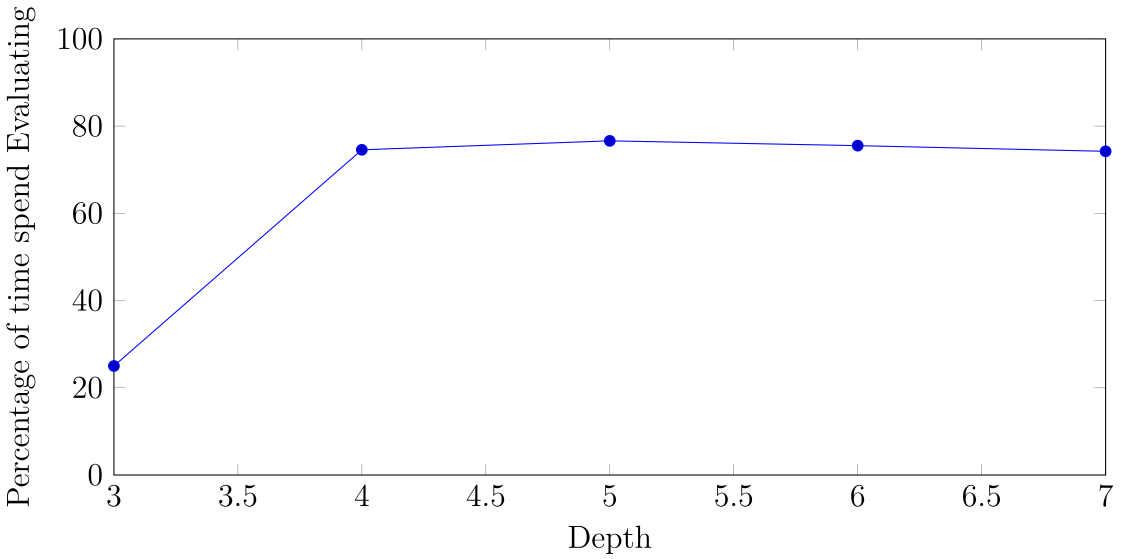
The following tests where executed on a ten by ten 2 player board with around 10 stones placed for each player. The OS used is an ubuntu distrubution run in an VM with 2 Gb of RAM and an i7 skylake processor.



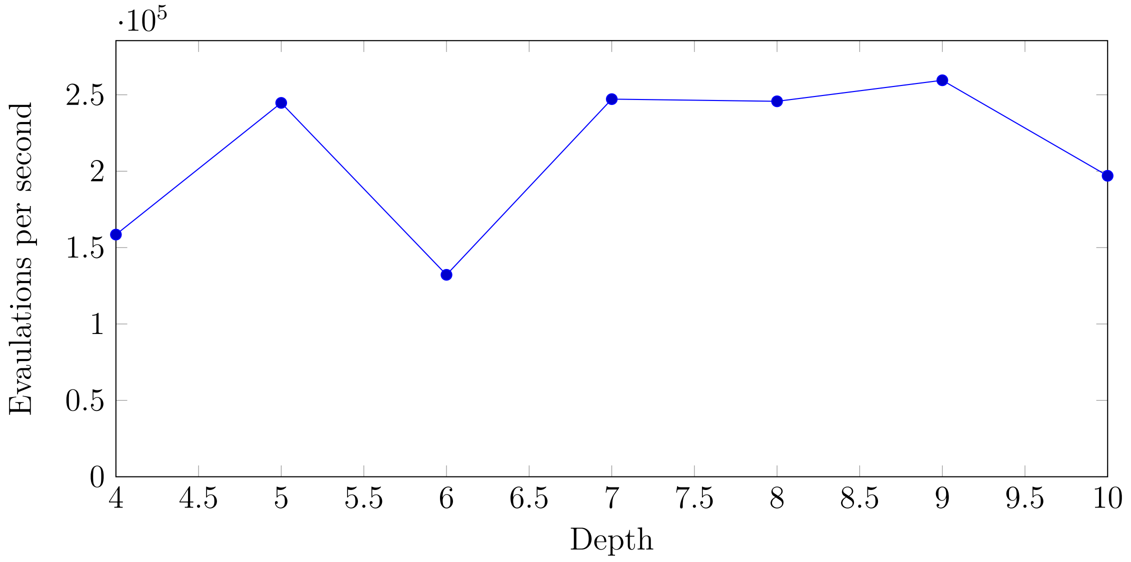


In the first two analysis we compare minimax with alphabeta-pruning. As the amount of nodes visited and the time take to execute the algorithms result in similar graph, we plotted the second one on a logscale.
In the first graph we see how much more efficient alphabeta-pruning is in compar-

ision to minimax. In the second graph that impression of superiority shrinks, but we can see much better that there is already a significant difference in small game trees.
These result were expected as minimax is in $O(b^d)$ and alphabeta-pruning's average case runtime is in $O(b^{\frac{3d}{4}})$.



We also found out that we spend about 75% of the time evaluating the leafs. How good or bad that value is remains to be seen.
We got this number down from 80% in just about an one our of pruning, and some more performance boosts should be expected.



This last graph shows something interesting. While the evaluations per second remain constant to a certain degree with increasing depth, as they should be, they seem to go up and down. It seems as if we do take more time analysing a board after we have done a move than if we analyse after our enemy has done so. This seems logical as we should have more stones on the board after our own turn, but the changes are quite significant. We might use this information when implementing iterative deepening, to skip some specific depth if we think we have enough time to do so and can risk it.