Softwarepraktikum SS 2016

Assignment 5

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
Implementation of Aspiration Windows

The first task of this biweekly assignment was to implement aspiration windows. As we have already implemented a way to select different algorithm, we did the same with aspirational windows to make to sure that we are able to disable them at any point in time.

The implementation of the algorithm is based upon alpha-beta pruning with move sorting enabled. The first iteration of iterative deepening is the same as previously with values of $-\infty$ for *alpha* and $+\infty$ for *beta*.
We now use the score of the best move found and try to predict the a window between which we expect the score of the next move to be in. As we use a paranoid search strategy, we know that the score will most likely go down if one of our opponents will do the move that was not captured by the previous depth limit. So we set the *alpha* value to the score of the current best move.
Otherwise, if we do the last move, our score should go up. Therefore, in these cases, we set the beta value to the current score.
Our first intuition to set the second value was to scale it up by a constant factor. This was not working at all, mainly because we can have a very low score but big addends(because of positive and negative factor cancelling out). So we increase/decrease the score be a fixed value depending on size of the map. Another problem was that we thought we could predict if the score goes down or not. This assumption was not true, most likely because of the way we evaluate the special stones.

Finally we used the formula:
$alpha = previousScore - x * getAmountOfCells()$
$beta = previousScore + x * getAmountOfCells()$
where x was the variable we tried to find the optimal value for.

If we the value returned by the move search is not between the alpha and beta value we know that we failed, low or high. In that case we will try again with just one value changed, but by a bigger margin. If we fail high, we will increase beta. Otherwise we will decrease alpha.
If we still fail after those adjustments, we will use $-\infty$ and $+\infty$ again.

After trying a lot of values for x we concluded that the aspirational windows do not help improve our search. The values at which we start pruning away some

branches are very close to the ones where we fail to find a move at all. The range is sometimes lower than 1. This means that even if we manually set the x value we see only small improvements for a very small range for x. Meanwhile the x value we did find to work for different benchmarks ranged from 3 to 100.

We did not include graphics as there where no informations to be gained from them other than described above, and no nice way to display what we did. We will not use aspirational windows for the moment. Before the tournament we might try to active them again if our final evaluation functions seem more suitable.

Task 2
Optimisation of Bombing Phase

Compared to the first evaluation method we implemented for task 2 of assignment 2 our evaluation changed completely. Back then we used the formula

$$Ranking(p) = \frac{|p\ stones|^x}{|enemy\ bombs|^y * explosionradius^z * rank(p) * clustering}$$

for player p with x, y and z being different weights. By adjusting the weights we searched for the best overall results. One of the problems we encountered with this evaluation function was that we can not do anything about the clustering of our stones. Furthermore it takes quite some time to check the neighbours for each stone.

Now, we compare the amount of stones we own more directly with the other players' amount. Our new formula for the bombing evaluation is:

$$Ranking(p) = \begin{cases} (|p\ stones| * (|players| - 1)) - stonesAbove & \text{if } stonesAbove > 0 \\ (MAX\_WIDTH * MAX\_HEIGHT) + \\ (|p\ stones| * (|players| - 1)) - stonesBelow & \text{else} \end{cases}$$

We only have to iterate once over all existing stones to count the stones owned by each player, and then iterate once over all players to check if they have more stones than we do. Those are added up in 3 variables: stones owned by *ourself*, stones owned by *players above us*(=having more stones than we do), and stones owned by *players below us*(=having less stones than we do). Disqualified players get ignored for the number of stones players have above and below us. For a bomb move we want to maximise our amount of stones on the map while minimising the other players' stones. If there are players with more stones than we own, we will focus them.

As we mark a stone as destroyed(=non-existent) by setting its state value to the max-value, we have to iterate over all stones that once existed. For that reason the number of stones iterated through is constant. We see no efficient way to change this, because while creating the map we saved every existing stone into an array over which we iterate.

Moves that will improve the rank in the ladder of a player by bombing stones of a specific other player, will get preferred by the search tree. This is the case because if throw a bomb that will result in us moving up a place in the ranking, the count of the stones of players above us does not only go down by the amount of stones bombed, but by the total of stones owned by that player plus the stones of other players that we bombed. On the other hand if there is no move that will improve the rank, the search tree will chose a move that will bomb in a fashion that closes the gap between the number of stones from the player and the stones from players with more stones then him. To make sure that a state where we are in the leading position always gets recognized as better, we add a constant of $(MAX\_WIDTH * MAX\_WIDTH)$ to the score. This value is higher than any result of the first part of the evaluation function.

The method, which checks if a move in the bomb phase is valid, was changed while a prior assignment. It used the depth first search algorithm that marks all the stones as to be destroyed. After all stones were marked, we iterated over all stones and destroyed the stones which were marked for it.
The search algorithm we use now is a variation of depth first algorithm. We mark all stones to be destroyed and added how many steps were taken from the starting stones to get to the stone. If a stone was already marked to get destroyed, the search algorithm will only check that stone and all it's successors if the steps taken to get to that stone is lower in the current run of the search tree.
That way we save checking some stones multiple times.