Project Report Omega

Kevin Trebing (i6192338)
k.trebing@student.maastrichtuniversity.nl

October 26, 2018

1 Description of the game

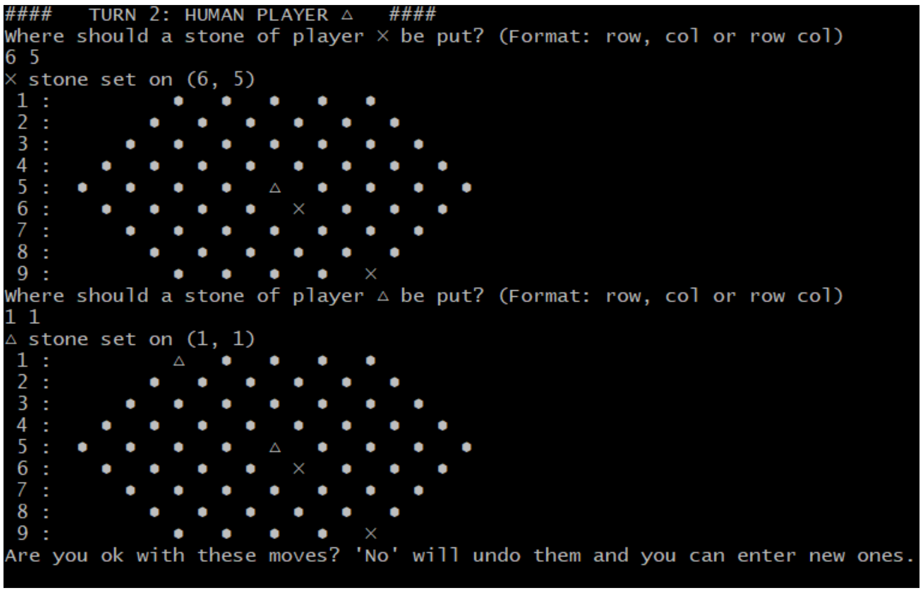
Omega is a game played by two to four players where the goal is to achieve the most points by the end of the game. The game is played on a hexagonal grid of varying size (and even shape). Points are calculated by multiplying the players groups of stones together. A stone belongs to a group of the same color when it is next to a stone of the same color. A turn of a player consists of playing a stone of his own color and of every other player. After every player has played a turn the board is checked whether another whole round can be played. This is done by counting the empty tiles on the board; they need to be at least the number of players squared, e.g.: two players need at least four free spaces and 3 players need at least 9 free spaces. If that is not the case the game ends and the scores will be calculated. The player with the highest score wins.

2 Choice of programming language

I chose the programming language Julia. Since Mark Winands told us that Python is "quite slow", the programming language I'm most comfortable with fell through and I had to choose another one. Julia is quite new (version 1.0 came out on 8th of August 2018) [1] and has a similar syntax like Python but was created for high performance computations. With Julia it is possible to have C-like speeds [2]. Although this is mainly useful for data analysis and machine learning I wanted to try it out for this project. Furthermore, it is possible to import Python and C packages which can be useful if there is no native Julia package released yet.

3 Usage of the program

In order to use my implementation Julia needs to be installed. To do this go to `https://julialang.org/downloads/` and download it. For easier use you can add Julia to your Path variable. Since my implementation uses packages that are not installed natively you need to run Julia and open the so called REPL (Read-eval-print-loop) and install them. Type 'using Pkg' (this is the package installer of Julia) followed by the following statements: 'Pkg.add("Printf")', 'Pkg.add("IterTools")' and finally 'Pkg.add("DataStructures")'.



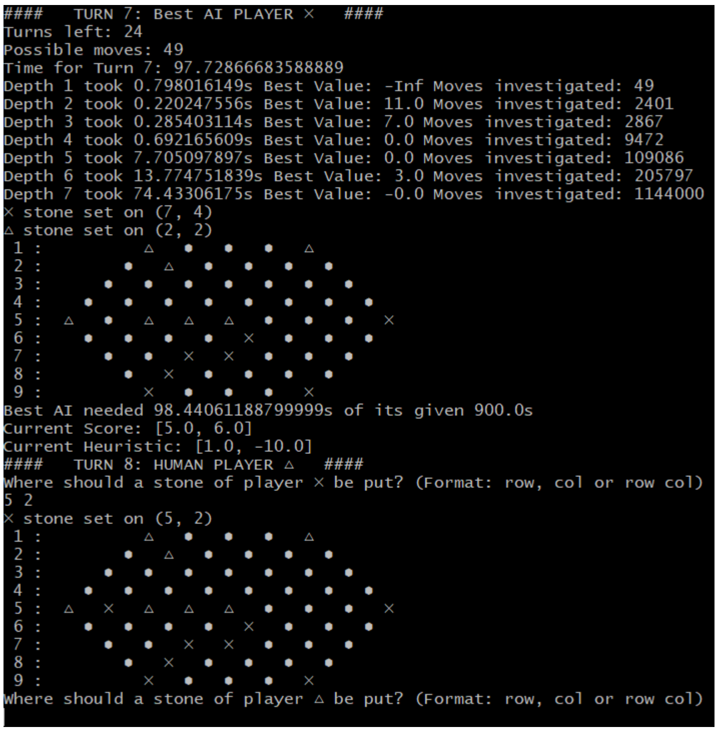
```
####   TURN 2: HUMAN PLAYER △   ####
Where should a stone of player × be put? (Format: row, col or row col)
6 5
× stone set on (6, 5)
 1 :         ● ● ● ● ●
 2 :        ● ● ● ● ● ●
 3 :       ● ● ● ● ● ● ●
 4 :      ● ● ● ● ● ● ● ●
 5 :     ● ● ● ● △ ● ● ● ●
 6 :      ● ● ● ● × ● ● ●
 7 :       ● ● ● ● ● ● ●
 8 :        ● ● ● ● ● ●
 9 :         ● ● ● ● ×
Where should a stone of player △ be put? (Format: row, col or row col)
1 1
△ stone set on (1, 1)
 1 :         △ ● ● ● ●
 2 :        ● ● ● ● ● ●
 3 :       ● ● ● ● ● ● ●
 4 :      ● ● ● ● ● ● ● ●
 5 :     ● ● ● ● △ ● ● ● ●
 6 :      ● ● ● ● × ● ● ●
 7 :       ● ● ● ● ● ● ●
 8 :        ● ● ● ● ● ●
 9 :         ● ● ● ● ×
Are you ok with these moves? 'No' will undo them and you can enter new ones.
```

Figure 1: An example of a human turn

With Julia correctly set up you can go to the folder of my implementation (you can get it here: `https://github.com/HansBambel/Omega`). In order to play a game of Omega you have to type in bash 'julia main.jl'. You will be prompted to type in parameters for the kind of game you want to play, such as board size and amount of players (currently only two players are supported for AI). It is also possible to start the program without the need to enter the parameters every time by using keyword arguments: 'julia main.jl <boardsize> <players>'. Boardsize is the dimension of the Omega board, e.g. 5 or 7. Players is a composite of two letters, h (Human), a (AI), r (random) or d (dumber AI), e.g. 'ah' would be AI as white playing against a human player as black. The sequence the players are entered also defines who is white and who is black. A complete call would be for example 'julia main.jl 5 ah'.

When playing the game the current player is displayed, when the current player is a human he is asked for the position to put a stone. The position is a 2-tuple, consisting of the row and the column. After entering the first position the player is asked for the next stone coordinates. If a mistake was done in either of these, he will be asked again. At the end of his turn he is able to undo the entered moves (if he wants to correct himself) and can restart his turn (see Figure 1).

Furthermore, it is possible to vary the amount of time the AI has to search. This is done by changing the variable *totalTurnTime* in main.jl. By default, the total time the AI gets is at 15 minutes, resulting in about 87 seconds per turn. In order to have a more timely game time I recommend 2-4 minutes. This reduces the time per turn to about 14 seconds.



```
####   TURN 7: Best AI PLAYER ×   ####
Turns left: 24
Possible moves: 49
Time for Turn 7: 97.72866683588889
Depth 1 took 0.798016149s Best Value: -Inf Moves investigated: 49
Depth 2 took 0.220247556s Best Value: 11.0 Moves investigated: 2401
Depth 3 took 0.285403114s Best Value: 7.0 Moves investigated: 2867
Depth 4 took 0.692165609s Best Value: 0.0 Moves investigated: 9472
Depth 5 took 7.705097897s Best Value: 0.0 Moves investigated: 109086
Depth 6 took 13.774751839s Best Value: 3.0 Moves investigated: 205797
Depth 7 took 74.43306175s Best Value: -0.0 Moves investigated: 1144000
× stone set on (7, 4)
△ stone set on (2, 2)
 1 :          △   ●   ●   ●   △
 2 :        ●   △   ●   ●   ●   ●
 3 :      ●   ●   ●   ●   ●   ●   ●
 4 :    ●   ●   ●   ●   ●   ●   ●   ●
 5 :  △   ●   △   △   △   ●   ●   ●   ×
 6 :    ●   ●   ●   ●   ×   ●   ●   ●
 7 :      ●   ●   ×   ×   ●   ●   ●
 8 :        ●   ×   ●   ●   ●   ●
 9 :          ×   ●   ●   ●   ×
Best AI needed 98.44061188799999s of its given 900.0s
Current Score: [5.0, 6.0]
Current Heuristic: [1.0, -10.0]
####   TURN 8: HUMAN PLAYER △   ####
Where should a stone of player × be put? (Format: row, col or row col)
5 2
× stone set on (5, 2)
 1 :          △   ●   ●   ●   △
 2 :        ●   △   ●   ●   ●   ●
 3 :      ●   ●   ●   ●   ●   ●   ●
 4 :    ●   ●   ●   ●   ●   ●   ●   ●
 5 :  △   ×   △   △   △   ●   ●   ●   ×
 6 :    ●   ●   ●   ●   ×   ●   ●   ●
 7 :      ●   ●   ×   ×   ●   ●   ●
 8 :        ●   ×   ●   ●   ●   ●
 9 :          ×   ●   ●   ●   ×
Where should a stone of player △ be put? (Format: row, col or row col)
```

Figure 2: Interface

4 Details of implementation

I chose to implement Omega on the command-line because Julia has no native graphical user interface library and it seemed to me a bit too much work to learn to program a GUI in Julia using another library additionally to learning Julia in the first place. Using unicode symbols still makes the graphical board representation appealing and useful (see Figure 2).

The grid has a hashing function for the later use of a transpositiontable. The hashing is done using zobrist hashing [4] to be computationally efficient. A random 64-bit Integer is created when the grid is initialized to represent the empty board. Then, for every hexfield three 64-bit Integer values are generated representing that this field can be either free or have a white or a black stone on it. When a stone is set on the grid, the corresponding Integer of the previous occupying stone is removed from the old hashvalue using the XOR operation. Then, the new stone is inserted to the hashcode using the XOR operation with the corresponding Integer of the new stone.

The hexgrid, additionally to having the current boardstate stored, has two lists of groups and groupsizes stored. This enables a fast calculation of the current score. When putting a player stone on the board a new hashcode is calculated and also a union-find algorithm [3] with path-splitting is used to update potential new groups. There is a difference to this when a stone of a player is removed (a field is freed up): additionally to storing the current board state, groups and group-sizes I also store the past groups and group-sizes. This makes it easy to revert to a previous state where the latest player stone was not set. This is done because a de-unification and thus updating all the groups is inefficient and requires more overhead.

When calculating the scores it is only required to multiply all values of the player's group-sizes together and not a scan of the whole board is needed.

5 Implementation of AI

The AI is implemented using a negamax search with iterative deepening and using a transposition table. To reduce the branching factor negamax is looking at a single move at each search depth and not a whole turn, which consists of two moves. This reduces the branching factor significantly: On a 6-by-6 boardgame it reduces the branching factor from 8190 to only 91 on the first move. Furthermore, with this division into two separate moves it allows negamax to prune even more resulting in less subtrees that are searched resulting in a deeper search. In order to make the search more efficient I use two move-ordering heuristics. Move ordering aims at investigating promising moves first as they may lead to additional pruning. In the following I will describe different techniques I use that resulted in an improved performance.

5.1 Transposition table

Using a transposition table enables iterative deepening to work, since without it the previous search is not used at all and only extends the search space. Furthermore, using a transposition table applies move ordering thus making iterative deepening more effective. As previously already mentioned, the transposition table is implemented using Zobrist-hashing [3]. The transposition table is reset after the AI finished its turn. This way it does not use outdated knowledge from previous searches.

5.2 Killer moves

In my implementation, the AI investigates first the move that was saved in the transposition table, then it will look into up to two killer-moves. A killer-move is a move that resulted in a cut-off in a previous search at that search depth. A newer killer move will substitute an older one.

5.3 Heuristic

The heuristic the AI uses consists of two scores. The first only takes groups of two and three into consideration. Those groups are then checked for their free neighbouring hexagons. Every free hexagon of a group-hex serves as a punishment, encouraging *safe* groups (groups that cannot be extended). All other, bigger groups, are disregarded until a terminal state is reached. The second score is the current board score. The final heuristic score consists of a weighting of these scores. The further the game progresses the more the current score will be taken into account and less the first score.

When a terminal state is reached, the complete score will be calculated, because this score can not be an overestimate since nothing changes any more.

This weighting of both scores has the effect that even later in the game the AI tries to form groups of two and three.

5.4 Opening Book

When the board is empty the amount of possible moves are very high, but as a human good opening moves are quite easy to spot. A good idea is to put your own stone in the corner of the board while trying to put the other player's stones in the middle. This results in safer groups of your own and unsafe and easily extendible groups of the opponent. In my implementation the AI will put a stone of its own in the corner when it is free and the opponent's stone in the center of the board.

5.5 Null-move

Using a null-move should improve the search depth even further. It assumes that when passing your turn, you are still in a good spot later on, you can do even better when actually making a move. Since the search depth when doing a null-move is more shallow, it requires less time to return a value. A null-move should not follow a previous null-move, otherwise people would just skip all the time. In order to avoid this the null-move flag is disabled after executing a null-move. Since a null-move should not occur after setting only one stone of a player, it is only enabled after a whole turn was done.

5.6 Multi-cut

Another enhancement which the AI uses is multi-cut. Multi-cut assumes that when there are a certain amount of prunings already at the beginning of a (shallower) search then there will be even more later on. This is a form of forward pruning and may be more aggressive resulting in incorrect or bad prunings. The AI prunes where there are at least three children out of ten where a cut-off occurred.

5.7 Time management

What the AI does for time management is the following: it gets the remaining time and calculates how much time it maximally is allowed to use when every turn receives the same amount of search time. Disregarded are the last three turns, which require in total less than 15 seconds to find a terminal state. The AI then uses the calculated time window to do a search. When searching, only every 1000 searches the amount of time left is checked, resulting in small deviations from the allocated time for the search and the actual time it needed. At the end of such a search the actual elapsed time of the search is subtracted from the remaining time resulting in new search windows for the next search.

6 Problems

6.1 Heuristic

Finding a good heuristic was a key problem. As my old AI-teacher always said in what felt like every lecture: “An admissible heuristics never overestimates.” Therefore, using the current score is not suitable because having a high score now can change quickly in the next move when multiple groups are combined to a single one. In the earlier state of the game the current score is not that important, making another heuristic more useful for that part of the game. Due to the fact that my heuristic does not take into account bigger groups of the players, which could occur over the mid and late-game it will become less suitable for later stages of the game. In order to compensate for that I decided to also take the current score for those phases of the game into account. The current score is multiplied with a discount factor that decreases the further the game progresses.

6.2 Side-effects

For efficiently calculating scores and evaluating the heuristic I use a union-find algorithm [3]. Every time a stone of a player is set on the field the union-find algorithm will unify it with its neighbouring groups if there are any. Unfortunately, separating a group is complicated / computationally inefficient and thus, I save the current groups in a stack. If a separation is necessary, I pop the last entry (where the latest unification occurred) of the stack and use the one before. This way I circumvent a delete function of the union-find algorithm. A problem I had was that the scores were not calculated correctly when the AI-player was searching. Even trying reproducing the exact same moves by hand did not reproduce the same bug. After 10 mind-boggling hours I found out that I needed to create a deepcopy of the latest entry to not have side-effects later on.

6.3 Usefulness of enhancements

I’m still not quite sure how useful (if at all) some of the enhancements are. Even after testing different combinations of enhancements (see Chapter 7) some of the enhancements usefulness can be questioned. What appears to be very useful is the killer-moves heuristic. Although other enhancements (null-move and multi-cut) manage to increase search-depth by one or two ply I can not say for sure how big of an advancement this gives since I do not have an appropriate opponent to test it with. In the lecture it was said that (at least in chess) searching just one ply deeper gave a huge improve in performance. That is why I am optimistic that these enhancements boost my AI’s performance. In the end, the AI still loses against a human player even with all enhancements activated.

7 Experiments

In this section I will compare the different enhancements and how much of a difference they make. The time for the AI was 15 minutes across all experiments on a board size 5. For this I let the fully enhanced AI play two games against the lesser enhanced AI. Once as white and another time as black. In order to

Only TT	NM	KM	NM+MC	MC+KM	NM+KM	NM+MC+KM
Win	Win	Draw	Win	Draw	Draw	Draw

Table 1: Results of matches of fully enhanced AI versus differently enhanced AIs.

see which enhancement was the most effective I let different versions of the AI play against itself. After every run I added another enhancement. Since there is (almost) no randomization (the killer-moves get randomly exchanged) we can consider all results deterministic and thus do not need multiple runs. All the versions have at least a transposition table and use iterative deepening.

The first comparison was against an AI that just uses a transposition table. The fully enhanced AI won both games and was able to search at almost every stage of the game 2-ply deeper than the transposition table-AI. Next, I compared against an AI with transposition table and a null-move heuristic. Here the fully enhanced AI won again both games. The difference in search depth here was about 1-ply in favour of the fully enhanced AI. For the next comparison I added a multi-cut heuristic. After two games the AI was able to win both of them. Although in the mid-game the search-depth was similar, in later stages of the game the fully enhanced AI was able to search two and sometimes even three ply deeper than the less enhanced AI. This may be a reason for it to win both of the games. Lastly, adding killer-moves resulted in both AI being equally potent, thus resulting in a draw, i.e. each won once.

Combining killer-moves with multi-cut resulted in the full-AI winning only one of the games. This could mean that these two enhancements have a good effect on the tree search. To further investigate how much of a benefit these enhancements give I enabled null-moves and the killer-moves and let it play with a fully enhanced AI. The result was that the fully enhanced only won one of two games. That is why I activated only the killer-moves and found out that the AI again won only one of two games. This indicates that killer-moves are a very effective method for search optimization.

In table 1 I noted what the outcomes of the differently enhanced AIs vs the fully enhanced AI was.

8 Tournament

In the tournament it became clear that one of the major issues were crashing games. Since my implementation only fails if there is a hashtable lookup crash (and the chances of that happening are low) I already had an advantage over other players. In the first round of the tournament my opponent's game crashed three times, she was able to recover one of these, but it crashed almost immediately after. Furthermore, many used the current score as evaluation score, which made my AI play way better in the start of the game because the opponent build big groups at the start to get a big score at the start. Sadly, I had a bug that made me lose one game. The bug was due to the opening book that selected an illegal move. I fixed it afterwards. The next three rounds were won 2:0. In the end I lost another game against another player who had a pretty sophisticated opening book. But in the end I ended up in first place. I did not expected that.

References

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia 1.0. `https://julialang.org/blog/2018/08/one-point-zero`, 2018. [Online; accessed 2018-09-30].

[2] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia microbenchmarks. `https://julialang.org/benchmarks/`, 2018. [Online; accessed 2018-09-30].

[3] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[4] A. L. Zobrist. A new hashing method with application for game playing. *ICCA journal*, 13(2):69–73, 1970.