# Project Report Omega

k.trebing@student.maastrichtuniversity.nl

October 7, 2018

## 1 Description of the game

## 2 Choice of programming language

I chose the programming language Julia. Since Mark 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. There 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.

## 3 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.

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 groupsizes together and not a scan of the whole board is needed.

## 4 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.

### 4.1 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.

### 4.2 Move ordering

Move ordering aims at investigating promising moves first as they may lead to additional pruning, making the tree search more effective. 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.

### 4.3 Heuristic

The heuristic the AI uses 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. 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 has the effect that even later in the game the AI tries to form groups of two and three.

### 4.4 Early Moves

When the board is empty the 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. That is the reason that for the first five turns the AI puts their own stones in the corner of the board and groups the opponent's stones in the middle of the board.

### 4.5 Null move

### 4.6 Multi-cut

### 4.7 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 about 15 seconds to find a terminal state. The AI then uses the calculated time window to do a search. At the end of such a search the elapsed time of the search is subtracted from the remaining time resulting in new search windows for the next search.

## 5 Problems

### 5.1 Heuristic

Finding a good heuristic was a key problem. As my old AI-teacher always said: "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.

### 5.2 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.

## 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 micro-benchmarks. `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.