Better AI for Hive: Mimicking human game-play strategies

1st Duncan Kampert
University of Amsterdam
The Netherlands
fjf@hotmail.nl

2nd Ana-Lucia Varbanescu
University of Amsterdam
The Netherlands
A.L.Varbanescu@uva.nl

3rd Matthias Müller-Brockhausen
Leiden University
The Netherlands
m.f.t.muller-brockhausen@liacs.leidenuniv.nl

4th Aske Plaat
Leiden University
The Netherlands
aske.plaat@gmail.com

***Abstract*—While Deep Blue and AlphaGo make it seem like board games have been solved, there are still plenty of games out there that are combinatorically similarly complex, yet the rules of the game make it too difficult for AI to mimic and best us in our game-play strategies. With Hive we present one such example. The current methods can barely beat a randomly playing agent. By applying state of the art methods and trying to improve them by baking in our human domain knowledge we attempt to improve upon the current dire state of Hive agents. While our AI still fails against actual Humans, it has still improved upon related work.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

Creating human-like game playing agents for the board-games or games we like to play has been done for generations [1, 2]. These game playing programs become increasingly complex and good at beating us, e.g. Deep Blue[3, 4], AlphaZero/Go [5, 6] etc.. However, there are only few games where human world champions have been beaten. Still many games remain that AI can not properly play, as it requires years of domain knowledge engineering or millions of dollars in compute resources (AlphaZero). The state of AI in Hive leaves a lot of room for improvement (Section II-D), hence we attempt to improve this by taking a look at the theoretical background of Hive (Section II). We then attempt to bake this knowledge into human-like agents via different methods (Heuristic, MiniMax, MCTS) and compare their performance.

A. Technical Context

Combinatorial games like chess or checkers are played by two players taking alternating turns playing their move Browne et al. [7]. Because both players have a free choice to make on what move to take, the state-space grows exponentially with the amount of moves. For a simple game like tic-tac-toe, this exponential growth is counteracted by the small amount of available moves at every turn, and as such its state-space can be fully explored. Even more complex games like checkers have had their optimal path fully explored, and as such can define the entire game as a draw [8]. However, when increasing the complexity of the game, this space can easily become too large to be explored within reasonable time.

For example, in chess, there are on average 40 moves available at every turn [9]. Adding on the average length of the game of approximately 80 turns [10], the tree search would have to explore $40^{80} \approx 10^{128}$ possible states. One can see this space is too large to fully explore. As a result, tree-searching algorithms have to work around searching the entire space somehow.

Instead of trying to solve the game, we apply Heuristics, that attempt to make a best-estimation of what the most likely outcome is of a board-state. Examples of algorithms using such estimations are Minimax or Monte-Carlo Tree Search, which both have a different approach to reaching these estimations. A third, fundamentally different approach uses machine-learning to reach these estimations. Due to the increased amount of available computational power, machine-learning is increasingly being used. As a result, it has also joined the subject of game-playing programs [5, 6]. To further boost the performance of these machine-learning approaches, a mix of tree exploration and machine-learning is typically used. For example, AlphaZero uses Monte-Carlo Tree Search, where their neural network decides which branches to take, rather than it being a fully random choice.

B. Objective

Instead of chess, we will be looking into a different game *combinatorial* game, called Hive. While not a lot of research has been performed on Hive specifically, we can make use of the existing body of knowledge for chess, and attempt to apply it to Hive. Creating a game-playing program in itself is not a challenging problem, but making a program play *well* is. However, the term *well* cannot be easily converted into a metric. For example, if a new chess-playing program were to be created, its *playing performance* can be compared to existing programs. As Hive does not have many existing programs, and none that plays it well, this comparison is much more difficult to make.

In this research, we will explain the thought process behind creating heuristics for new games. Additionally, we will compare the application of these heuristics to MCTS and Minimax. Next, we will define metrics which can be used to rank different game-playing algorithms. Having such a ranking in place would allow us to compare different algorithms without having to play every individual version against each other.

Fig. 1: The Hive game after many moves, the two stacks in the back are yet to be played.

II. Hive

Hive is, similar to chess, a turn-based two-player full-information game. This means that, in theory, the same algorithms that are applicable to chess can be applied to Hive. However, there are some unique properties to Hive which make it an interesting game to explore further. The first and most obvious difference is that Hive is not played on a board. Instead, the tiles with which are used are placed next to each other, thus creating a dynamically changing playing field. Additionally, not all the tiles are *in* the game when it begins. Both players have a pool of (eleven) tiles, and a turn consists of either of two actions:

1) Placing a new tile on the board
2) Moving an existing tile to a new location

In total, there are five unique classes of tiles, with each their unique rules to movement (the rules can be read in more detail here [1]). The ultimate goal of the game is to surround the *queen* tile of the opponent. The first person to accomplish this will win the game.

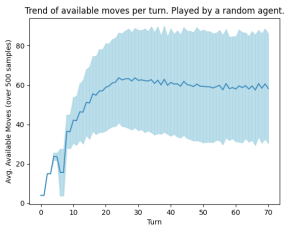
For our research, we will be using Beekeeper [2] as our game-engine. This means Beekeeper will keep track of all board states, generate possible moves, and free all memory whenever it is done being used.

When doing a tree-search, it is beneficial to know some properties which heavily influence the search time.

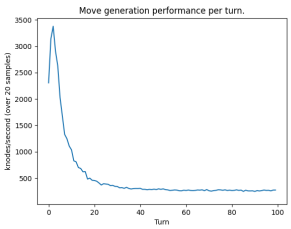
A. Branching factor

Doing a branching-factor analysis gives us one factor which influences the complexity of our tree-search. To do this, we can generate random moves for n turns, and plot the average amount of available moves at each turn. This shows us a trend of the branching factor during the span of the game.

Figure 2a shows the average branching factor throughout the game, with the standard deviation at every move. This indicates that, during later stages of the game, the branching factor stays stable at 60. Thus, the exponential growth of Hive is approximately twice larger than that of chess, which is around 30 [9].



(a) Average branching factor per turn. Blue area is the standard deviation.



(b) Performance of the move generation per turn. Average taken over 100 turns.

B. Generator Performance

A second large influence of the search time is the actual time spent by the move generator. The trend of time-per move can be seen in Figure 2b. For Hive, a highly optimized engine can only generate around 400k nodes per second during later stages of the game. Combining this with an average branching factor of 60, doing a full search to depth 4 will take $60^4/400.000 = 32.4s$. This means that rather than being able to do a full search to a very high depth, making good choices on what branches to explore is much more important.

C. Creating Heuristics

For many of the classical games being played by computers, like chess or checkers, there has been created a large body of work spanning many years. For a new game like Hive, this same body of work does not yet exist. There are some resources available on how to play Hive, specifically catered to human play [11]. The issue with these strategies are that these concepts are not directly transferable to an efficient algorithm identifying them. To name an example, placing two beetles on top of the Hive near your own queen is good when defending. However, identifying *when* you are defending is not something that can be counted as easily.

As we have no Hive-related work to build upon for creating heuristics, we have to identify (and implement) them ourselves. To this end, we adopted an iterative process, empirically evaluating the play of the engine and finding out a way to improve upon it. While we later on used our heuristics for both Minimax as well as MCTS, we evaluated our heuristics exclusively using Minimax. We did this because Minimax will more clearly show the tendencies of the heuristic, as it will have no other information to base its decisions on, whereas MCTS will have the added information of the random playouts.

Our initial heuristic is simply based on the actual rules of the game: winning gives you $+\infty$, and losing $-\infty$. The issue with this was that it would not converge towards winning states. Essentially, it would play random moves, until it would by sheer luck find a forced win in ≈ 4 turns. To attempt to combat this, we added a simple convergence component to our

[1] https://www.ultraboardgames.com/hive/game-rules.php
[2] https://github.com/fjf/hive_engine

heuristic: tiles around the queen. This is likely on of the most straight-forward heuristic enhancements that can be made to a Hive engine. It is so straight-forward, that we found other open-source Hive implementations using exactly this for their Hive agent [3].

The issue we found does not occur when only bots play against each other. Instead, when faring up against humans the problems of this agent quickly surface. After playing a single game against this agent, the (human) player would know the agent plays very aggressively, and will place tiles around the opponents queen as soon as it can. The issue with this was: it does not identify it is behind in progress of placing tiles around the queen. This means that, if there are 4 tiles around the agent's queen, and two are ready to fill the remaining two spots, and the agent has the option to either *block* the movement of a tile, or *place* the tile next to the opponent's queen, it will always pick the latter. While this behaviour is obvious when looking at the heuristic is playing with, it is still difficult to preemptively assess the playing strength of such an agent.

The next step to attempt to make the agent play more like a human was to make it prioritize placing new tiles on the board. It would get a higher positive value the more tiles would end up on the board. This is easily countable, and as such easily implemented in the agent.

To attempt to make the engine play a more *elegant* game, rather than throwing tiles at the opponents queen until the game ends, we added the first heuristic component based on strategy. This component was based *blocking* of tiles, based on their relative value. While this relative value has not yet been completely defined, there are some indications of what the strongest players think is more important. The queen is the most important piece, so blocking this is (of course) helpful. After this, the ant is the most mobile tile which allows for the blocking of tiles. For the other three tiles, there is no clear ordering of strength. Instead, they have advantages and disadvantages based on the state of the Hive, and what movement options they have. This makes it very hard to give the other three tiles relative values which are an indication of their actual importance in the game. In the end, we set the relative values to $5, 3, 1$, for the queen, ant, and rest of the tiles. Now we have three parts in our heuristic function, we have to combine them into a single value. For this, we use the following formula;

$$h = \sum_{n=0}^{P} c_p * f_p(board) \tag{1}$$

Where h is the resulting heuristic value, P is the amount of *parts* of our heuristic, c is a vector of weights, and f is a vector of functions which extract a value from the board.

While Formula 1 is not expensive to compute, the problem comes from assessing the *best* values for c. The values of these weights then also need to be optimized, as this is what strongly influences the eventual playing strength of our agent [12]. We will set our initial weights to 10, 1, and 5 respectively, values chosen by relative importance.

D. Related Work

We are not the first to try to build a Hive-playing program, nor the first to document it. Multiple projects have attempted to create a good Hive agent, using various game-playing algorithms. For example, Barbara Konz [13] created a Hive program using MCTS. They reported an 80% win rate of their MCTS implementation against random, where the last 20% of games ended in early terminations due to turn limit or draws. They found their game-playing strength to be best when the UCT exploration factor was set to 100. Lastly, their engine takes approximately 30 seconds per turn. This makes their tests very computationally intensive, and, as such, their UCT parameter is likely good specifically for their low amount of generated MCTS playouts.

Another group attempted to use reinforcement learning to play Hive [14]. They had a similar issue to the implementation described above: running simulations to let the neural network learn from took too much time. To give an example, to play 200 games, their experiment ran for 14 hours. Of course, with a complex game like Hive, 200 games is not a large enough sample size to let a neural network learn how to play.

Tamás Bunth [15] also used reinforcement learning to play Hive. They used the same strategy as was used in AlphaZero to train the network how to play: using MCTS simulations and a pool of players iteratively replacing the worst one. As they had too many hyperparameters to search, they picked values that were comparable to that of AlphaZero, and lowered the amount of MCTS simulations to speed up the training process. This resulted in their final version to achieve a 71% win rate against random, where the remaining 29% of games were draws.

Connor Michael McGuile attempted to create a Swarm AI for Hive [16]. They compared their AI to a fully random agent and a true MCTS agent. To speed up their experiments, they did make some alterations to the game, by using used randomly initialized board-states, instead of starting from an empty board. While this alteration might make sense in the context of a swarm AI, it does remove some complexity in the game. Additionally, they only allowed for 40 turns to be played after this initialization. When looking at the average game analytics for game length, as shown previously in Figure 2a, this 40 turn limit (or 20 moves), seems to be on the low end. Additionally, for their MCTS agent, they limited the playout depth to 13, reducing the amount of time spent generating MCTS playout samples. Their swarm AI performed equally well compared to a true MCTS agent, with an approximate 50% split between wins and draws.

Lastly, an AlphaZero-like approach was taken by Danilo de Goede [17]. They quickly ran into the problem that it was very computationally expensive to explore all the factors that come with using deep RNNs. As such, they simplified the game-rules, and managed to barely outperform the fully

[3]https://github.com/shaw3257/hive/blob/76ed671366f3d844266fd06209d992010bea0b63/app/public/ai.js#L89

random agent after four hours of training. In comparison to all previous work, they did use Elo ratings for a more accurate representation of the playing strength of their algorithm.

Compared to previous research, our work highlights the advantages and disadvantages of available metrics for playing strength, and decide which ones are best to use for our purpose. We will use Beekeeper [4] as our Hive engine, which is the best performing engine currently available. Lastly, we compare a larger set of algorithms to each other and attempt to order them by playing strength.

III. Evaluation

This section presents the evaluation of two different tree-search algorithms, MCTS and Minimax. We will compare the playing strength of the algorithms to each other, and also investigate what the impact of some enhancements are on the playing strength or their respective algorithms. All experiments discussed below are run on a DAS-5 node [18], featuring dual 8-core 2.4GHz (Intel Haswell E5-2630-v3) processors, with 64GB memory.

A. Metric for Playing Strength

Evaluating the *playing* performance of a tree-search algorithm is difficult to do, because there are no proven objective metrics that can be easily measured. For long-standing established games, like chess, there exist algorithms against which a *new* algorithm can be compared. However, as Hive does not yet have this luxury, we have to propose a different metric to assess playing strength.

One possible objective metric is the *win-rate versus a fully random agent*: the higher the win-rate, the better the algorithm would be. This metric has been proposed and used before as an indication of Hive playing strength [13, 15]. The disadvantage of this approach is that as soon as a version of an algorithm starts performing reasonably well, it will win all of its games against the fully random agent. Especially in Hive, there are very specific moves that need to be made from a given position to result in a win. This makes it very difficult for a fully random agent to generate any kind of advantage, as it has to - by chance - pick several correct moves in a row. Consequently, if an algorithms only looks one move ahead, ensuring that it will not lose immediately, the random agent will likely never win.

Thus, as an extension on the win-rate, we propose to also take into account the *average amount of moves required to win*. This is useful information, because we expect a better algorithm to converge quicker towards winning positions on the board. As a result, having a lower average amount of moves required to win would correlate to having a stronger algorithm. This same metric has been used several times before in Hive implementations [13, 16], allowing us to also compare our algorithm to other implementations.

Lastly, we can use *relative playing strength* as a metric. Instead of comparing all algorithms using some baseline, we can make them play against each other, and use wins against each other as an indication of strength. One issue with this approach could be that the so-called "rock-paper-scissors situations" may arise. For example, assume algorithm A employs strategy 1, which works very well against strategy 2, used by algorithm B. If strategy 2 works very well against strategy 3 by algorithm C, and strategy 3 works well against strategy 1, no objective *best* algorithm can be declared. Thus, to be able to determine which algorithm *is* best, one has to always play every version of every algorithm against all the others.

However, as this all-against-all approach is very time-intensive, alternative ways are strongly preferred, to ensure timely analysis on algorithm variants. Thus, to reduce the required evaluation time, and still use relative playing strength as a metric, we propose to assess the algorithms in *a tournament-setting*. Specifically, the algorithms play each other, and have a rating assigned to them. In our case, as Hive is a two-player zero-sum game, we can simply use Elo rating [19]. Elo rating allows for wins, losses, and draws to influence the rating of an agent. This is beneficial for us, as we can use a fully random agent as a control-group in the tournament pool. As we expect the fully random agent to never win a game, its rating will be lower than all other agents in the pool. However, if a higher rated agent draws against the random agent, the higher rated agent will lose rating points, as it was expected to win.

Because it is the simplest and most straightforward metric to estimate playing strength, we will use the win-rate against random, combined with the average amount of moves required to win. However, if this metric turns out to be not sufficient to still distinguish between different algorithms' playing strengths, we will use Elo-rating instead.

B. Experiments

Our initial experiment matched MCTS up against Minimax [20], using the win-rate against random as a metric for success. MCTS had UCB1 implemented, and we tested two different values for the exploration constant: 1.4 as the theoretical optimum [21, 22], and 100 as the empirically found optimum for Hive [13]. For Minimax we implemented $\alpha - \beta$ pruning [23], iterative deepening [24, 25], and transposition tables [26, 27]. The results of this experiment can be found in table I.

Algorithm variant	Win-rate	Average turns to-win
True MCTS	0.7	60.26
UCB1 c=1.4	0.8	57.5
UCB1 c=100	0.66	53.17
Minimax	0.66	47.0

TABLE I: Win-rate and average game-length for various algorithm variants. Allotted time set to 1 second per move.

While this initial result would indicate Minimax is much more promising to explore, MCTS does not have any heuristics to base its decisions on. As the performance of our engine does not allow for many MCTS playouts to be completed, there is not a lot of information it can base its moves on. Thus, we

[4]https://github.com/Fjf/hive_engine/tree/master/c

implemented two additional enhancements to MCTS: random playout guidance, and First-move urgency [28].

While the win-rate against a random agent does show very large differences in performance, it struggles to show smaller differences. This becomes more and more clear as the average game-length gets lower, as there is a fixed lower bound for the game-length. To be able to more clearly identify the differences in playing strength, we opted into using an Elo-system for our next experiment with more different agents.

However, such an evaluation has a much higher cost compared to playing against random. For most versions, a game against random take 50 turns on average, which means there are 25 turns for the algorithm. With one second per turn, we only need to wait 25 seconds per sample. By comparison, to compute an accurate Elo rating, we need to play multiple games for each algorithm. Ideally, we want to:

1) Match-up all pairs of algorithms.
2) Let each algorithm play both sides of a match-up. While the analysis on Hive games has shown that there is no statistical advantage on playing white or black, we are not sure if this is the case for (all) our algorithms. It could be that a certain strategy employed by Minimax strongly benefits having the first, or the second move.
3) Play all match-ups multiple times, because Elo ratings become more defined after more games.

This as the pairing of algorithms creates an exponentially growing set of pairs, this will very quickly become too big a space to fully explore. For us, as we have 5 base versions (2 MM and 3 MCTS), and a random agent, that already gives us $15 * 5 * 2 * 100 = 15000$ [5] seconds. If we double the amount of versions from 5 to 10, it goes to $55 * 5 * 2 * 100 = 55000$ seconds, a little over 15 hours. Additionally, this assumes every move will take one second, while we would prefer the agents to have more time to *think*. So evaluating a large amount of versions quickly becomes very expensive, especially if we want also increase the allotted time.

We initialize Elo to 1500, which is the theoretical initial seeding value. For this test, we picked the best versions of our previous tests, and added the two new MCTS enhanced versions. Meaning, MCTS uses the exploration constant of 1. Additionally, we gave minimax various amounts of time per turn, ranging from 0.01 seconds to 1 second.

Initially, the minimax results were very surprising. We expected the minimax versions to converge towards higher ratings for versions with higher allocated time per turn. The actual results showed 0.1s as the highest rated, followed by 0.01s, with the 1s-minimax rated third. We believe this was caused by shared access to the transposition table, combined with some randomness of move selection. After removing the transposition table cross-accesses, we re-ran the experiment to ensure no interference existed between versions anymore.

The Elo rating results are presented in Figure 3.

[5] 15 total pairs of 6 versions, playing each matchup 5 times, and every matchup gets played from both white as black perspective (2). Assuming every game takes up all possible moves, it will go for 100 moves.

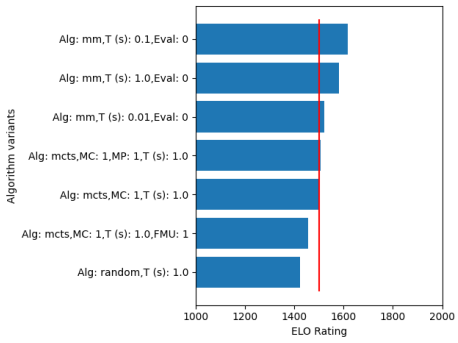


Fig. 3: Elo rating for different algorithms and different allocated time-shares. Red line shows the initial rating for every algorithm.

We observe that the minimax versions are ordered slightly more logically, but the 0.1s version is still rated higher than the 1s version. Investigating this further, we discovered that the individual game results show that the 0.1s version drew less games against the random player during this specific tournament. As drawing against a significantly lower player also results in a loss of rating, these results penalized the 1s version more, resulting in this peculiar ordering of algorithms.

Despite this rather odd ranking, we believe Elo-ratings to be a better method of comparing a large set of algorithm versions. However, the exact placement of these versions should not be taken at face-value. Instead, the rating can be taken as an approximation of strength. Moreover, we believe that, as the algorithms strength increases, the variance in the results decreases, and the Elo rating will become more accurate, as we expect stronger algorithms to never draw against a random agent.

C. Optimizing the evaluation function

As described in Section II-C, our initial heuristic components are unlikely to be optimal. So, to increase the performance of our initial version, we need to do a parameter sweep to find the best configuration of weights. There are a few problems with finding these best values for c in Equation 1. Firstly, to do a parameter sweep, we need to define a lower and an upper bound. Luckily, as the total heuristic value can be scaled by a constant, we only need to bother with the ratio between the different parts. As we can use floating point values, we do not need to set a large bound, and can instead use as many intermediate values as we want.

To then find these values, we can define some values within our bounds, and simply brute-force over all combinations. However, this will quickly grow to become a large volume of permutations. In combination with the expensive evaluation of our agent this becomes infeasible to do within reasonable time. Instead, we can use a smarter method of parameter optimization. Our parameter optimization is similar to the hyperparameter optimization employed for neural-network architectures for two reasons: we have a large exploration space,

and the game-results are non-deterministic. Thus, as we will be using the python framework Optuna [29].

*a) **Using Optuna**:* As we have three components to optimize for, which are represented as floats, we have to tell Optuna what the search space is for each of our parameters. The range chosen has to be large enough such that it does not get influenced by the random initialization of the evaluation value. As this randomization is in the hundreds of a digit, a sufficiently large enough range should not have meaningful random influence. Initially, we simply the range from 0 to 10. Because the values have relative importance, Optuna should be able to find a good configuration in here.

To measure the efficacy of any new configuration, we compare it to our base-line configuration. Our initial configuration set the weight values to: $UnusedTiles = 1$, $Movement = 5$, and $Queen = 10$.

For Optuna to distinguish between the *goodness* of two difference configurations, we will use the win-rate of our agents, which is computed as follows:

$$f = \sum_{n=0}^{N} \frac{r}{N}$$

Where $r = 0, 0.5, 1$ for a loss, draw, or win respectively, and N the total number of games.

After running Optuna for 100 trials, we can perform a rerun against these newly accrued values, and let the optimizer attempt to find an optimum again. This process should identify if there is a rock-paper-scissors scenario in the parameters, where it might be impossible to find an optimum.

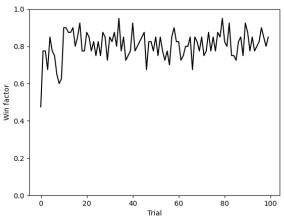
As Optuna naturally has very spiky parameter values, we smoothed the parameter values (Figure 4(c,d)) using a 1d convolution with a 3-wide Laplace stencil $[0.25, 0.50, 0.25]$. This procedure obfuscates the actual values per trial, but it illustrates better the overall trend of the values, which is more important. The win-factor is not smoothed in any way.

Figure 4 shows the Optuna results. To interpret them, we look for correlations between different parameters combinations with the win factor. Thus, we see in Figure 4c and Figure 4d that the $Queen$ component of the evaluation function should not be heavily factored into the eventual evaluation function. Especially in the rerun, we can see that when the queen is factored in more (i.e., the parameter value is high in Figure 4c and Figure 4d), the win factor drops significantly. This indicates that, for a more effective evaluation function, we should not use the amount of tiles around a queen as an indication of board *goodness*.

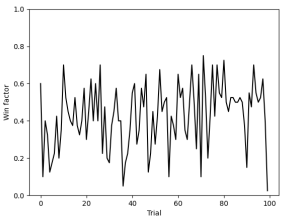
Looking back at our other experiments, specifically the evaluation of MCTS enhancements (Section III-B), the degradation of playing-strength after implementing these enhancements makes sense. Those enhancements specifically looked at prioritizing moves to surround the opponent's queen, but this is apparently not the best strategy to employ.

1) Refining the components:

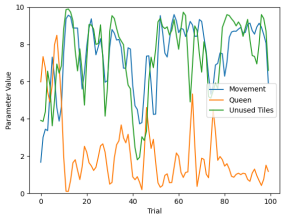
*a) **Refinement per tile**:* We can build upon this experiment by further dividing the evaluation function components



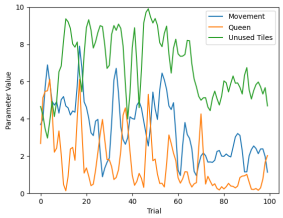
(a) Win factor vs. the random agent.



(b) Win factor of rerun vs. previous best.



(c) Parameters vs. the random agent



(d) Parameters of rerun vs. the previous best

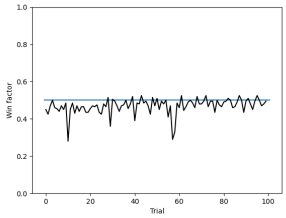
Fig. 4: Example results from an Optuna run attempting to optimize the three parameters for the Optuna-3 evaluation function.

per each specific tile. For example, instead of the amount of tiles around a queen, we count the amount of *ants* around a queen. Such finer-grain components might allow a better evaluation function than the 3-parameters one (which we will call Optuna-3), because the evaluation function can prioritize certain tiles with different importance per tile.

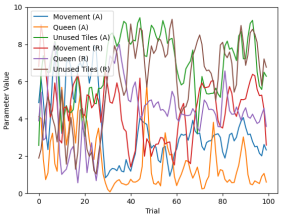
However, while Optuna can likely optimize for 15 parameters, it does mean we would have to significantly reduce the variability in the results. Where before we ran 20 game simulations, we would need more for the 15 parameters, as the optimizer would have difficulty finding out which of the parameters influence the win factor if the win factor is very inconsistent. Additionally, we would have to run more trials, as our small amount of trials (100), would likely not allow for Optuna to find the optimal configuration of values.

We do propose a second experiment, where every component is split into two, rather than five parts. Specifically, we make separate components for the ants, because they are the most mobile of all tiles, and are generally accepted to be the most powerful tile nearing the end of the game. We rename this evaluation function Optuna-6.

In Figure 4c, we see a very spiky win-factor for Optuna-3, which is what we would like to reduce for Optuna-6, where we have more parameters. To accommodate for our added three parameters, we also increased the amount of games to base the win-factor on. Instead of 20 games, we now set $TotalGames = 100$, which should reduce the variability in the results. The amount of trials is kept the same. The results are presented in Figure 5. Specifically, Figure 5a shows the win-rate against the best performing parameter configuration from Optuna-3.

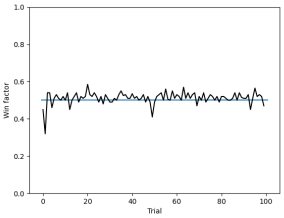


(a) Win factor vs. Optuna-3. Because the win factors are very similar together, the blue line indicates 0.5.

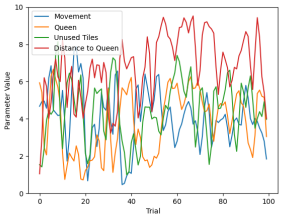


(b) Parameters of new run vs. Optuna-3.

Fig. 5: Result of an Optuna run attempting to optimize the six parameters for the Optuna-6 evaluation function.



(a) Win factor vs. Optuna-3. Because the win factors are close together, the blue line indicates 0.5.



(b) Parameters of added parameter run vs. Optuna-3.

Fig. 6: Result of an Optuna run attempting to optimize the four parameters for the Optuna-4 evaluation function. All games played against the best Optuna-3 configuration.

We observe in Figure 5a a huge reduction in the win-factor variance. Thus, the increased amount of games played to compute the win-factor is likely useful to optimize the configuration parameters. Maybe using an even higher count of games to base the win-factor on would reduce the variability further, but this comes with the disadvantage of having to run more samples.

Sadly, we do not see a significant improvement in the win-factor when splitting our evaluation function components into two parts. Additionally, from empirically evaluating the play of the agent, we noticed our best configuration of weights has a very specific strategy in play: it attempts to block out the opponents tiles repeatedly until the opponent has no free tiles left. After this, it would not necessarily converge towards a win, or attempt to move its free tiles closer to the opponent's queen. While this could be a good strategy to increase ones win-rate when playing hundreds of games against a computer, this is not necessarily very interesting to play against for a human.

*b) **A convergence parameter**:* To attempt to combat this newly-learned strategy, we added a convergence parameter to Optuna-3. This parameter measures the sum of distances between all the tiles of one player and the opponent's queen. This new parameter would hopefully push the minimax implementation to slowly move its tiles closer to the opponent's queen. The results for this new function (i.e., Optuna-4) are presented in Figure 6.

Similar to the previous results, the parameters in Figure 6b do not seem to clearly converge towards certain values. While this is somewhat expected behaviour due to Optuna's Gaussian optimization method, we do think that, given more trials, the values would eventually find an optimum. More interesting is the win-factor, as shown in Figure 6a. The addition of this additional *distance* component to the evaluation function very quickly results in a positive win-factor against the previously best version.

IV. Discussion

In our attempt to build a highly-competitive Hive agent, we have tried many different algorithms, with a multitude of enhancements for each. In this chapter we analyse the current status of our best agent, and highlight some of the challenges to be faced to further improve it. We further analyse how the current agent fares up against humans.

A. The status of the current engine

Currently, the best version of an engine, by a small margin, is minimax, with the Optuna-4 weight configuration. However, the playing-strength of our algorithm implementations did not manage to consistently beat the best existing engines. The existing engines likely [6] use minimax with a more fine-tuned evaluation function.

Finally, we evaluated the engine against human players. We found that our agent consistently outperforms new players, and seems not to draw as often as our experiments might indicate. However, when pitted up against stronger players, the weaknesses of our engine quickly get highlighted, and it will consistently lose. Whenever a player employs long-term strategy to the placement of their tiles, the engine will not be able to recognize this strategy until it is too late. This places the strength of our best agent somewhere between a new and an advanced human player.

1) Minimax and its challenges: In general, an engine based on the minimax approach is only as strong as its evaluation function.

Our evaluation function - derived from the game rules and from strategies discussed in "Play Hive like a Champion" [11] - assumed that the amount of tiles around a queen correlates with higher winning chances. However, our in-depth analysis into the evaluation function used in minimax (see III-A) showed that our intuition does not hold.

We have further shown that, with more hand-picked properties extracted from the current board-state, evaluation functions can be further improved (see section III-C1). These preliminary results indicate that there are many possible ways to combine the current set of extracted properties of a board state in a more intelligent way.

[6]The best Hive-engines are closed source programs.

For example, we ran into the issue that some extracted values (e.g., how many tiles are not yet placed) are not equally important throughout the game. However, to be able to have this (variable) importance shine through in the evaluation function, we need scaling weights. As doing this for multiple types of scaling and multiple weights for both the overall value, and the scaling of the value is a lot of fine-tuning, it is not suited for our current method of parameter optimization.

Given a large enough set of properties to which we need to assign variable weights, we believe this might be a good place to insert a lightweight neural-network. This neural network can then mix-and-match the weights to find some optimal configuration. An added benefit is that the neural network is not limited to linear addition of weights, and can instead make any arbitrary combination.

2) MCTS and its challenges: The lower-than-expected impact of the number of tiles around the queen as a parameter also has implications on the working and success of MCTS. Specifically, it is exactly this game-state property we were using to base our MCTS enhancements on. For example, the move prioritization strategy was solely based on how many tiles are around a queen in a certain board.

This means that, while we did do experiments on First Move Urgency and move prioritization (see Section III-B), the current heuristic is not suitable to increase the playing strength of MCTS. Ideally, we should have used a better prioritization strategy, which is cheap to compute, and actually increases the winning chances. However, to the best of our knowledge, there is no cheap way to compute a game-state property which can improve the winning chances.

Until such a strategy is found, MCTS enhancements are difficult to use for two reasons: simulating playouts is expensive, the current (good) heuristics for Hive are expensive. In our experience, we have very little playouts to work with in MCTS, which means the enhancements we have to use either have to increase the quality of the playouts, or increase the amount of playouts. Furthermore, the enhancement may degrade the performance of the playout simulation too heavily. While this would increase the quality of some samples, the reduction of total samples would not result in better playing-strength of MCTS.

V. Future Work

We believe the main improvement towards creating a stronger agent is the addition of more separate optimizable components in the heuristic function.

We currently have a small set of variables which were optimized for, but many other components can be added to the evaluation function. For example, specific structures which are shown to be good [11] can be identified and counted (e.g., a c-shape with the queen in the center). Alternatively, using the current turn to change the weights depending on game progression may benefit the evaluation function (e.g., placing tiles is more important in the first 20 moves). As the optimization strategy will find a close-to-optimal weight configuration for any amount of parameters, increasing the amount of ways for the evaluation function could be a benefit to the evaluation function. Additionally, it might be useful to use a small neural network which gets fed the current components of our evaluation function. This neural network can take a few pre-extracted board values, and output the final evaluation. Compared to the AlphaZero approach, this is significantly less computationally expensive, and is expected to improve the performance of the minimax evaluation function.

VI. Conclusion

Letting artificial agents mimic human behaviour to improve their game-playing strategies has proven a difficult task.

The lack of simple-to-compute moves makes it rather difficult to create an efficient move-generator. For Go, doing a move is as simple as setting a bit to 1, and a move is valid if this bit was set to 0. Where for Hive this is a lot more expensive.

Compared to MCTS implementations for different games, we are at least a factor 50 slower in terms of playout sample generation. Combining this with the high branching factor inherent to Hive, creating a version of MCTS which works well becomes very difficult. The same applies to Minimax, although there the performance limits the depth of the search.

We have also compared multiple metrics for playing-strength evaluation, and shown the advantages and disadvantages for each of them. While the win-rate and average amount of turns against the random agent is quick to use as a comparison, it is not suitable when comparing two different *good* agents. Even if one agent wins 80% of its games against another, they might both have the same win-rate and average turns against the random agent. This makes it very difficult to compare all agents to each other using this baseline.

Instead, an Elo-based system should be used, where the strength of an agent is based on which opponents it wins against. The better the opponent it wins against is, the more rating points it gets. This is significantly more expensive in terms of computing power, but it does allow us to create an objective comparison between a large set of agents. We have shown here that even though the win-rate against random was equal for some agents, the rating difference was 200+ points. Thus, when comparing agents for Hive, win-rate against random should be used cautiously, as it might indicate two agents being equal strength while this is not actually the case.

We have also evaluated the engine against an engine trained using the AlphaZero approach (based on reinforcement learning and self-play). Although the AlphaZero variant implementes a slightly simpler version of the game, our version still outperforms it [17]. However, we expect that more training and tweaking of the machine-learning variant will also lead to improvements.

References

[1] R. Snodgrass, "LANL unable to release history report," 2008, http://lanl-the-rest-of-the-story.blogspot.com/2008/07/lanl-unable-to-release-history-report.html.

[2] R. B. Lazarus, E. A. Voorhees, M. B. Wells, and W. J. Worlton, "Computing at LASL in the 1940s and 1950s," 1978. [Online]. Available: https://permalink.lanl.gov/object/tr?what=info:lanl-repo/lareport/LA-06943-H

[3] A. J. H. J. Feng-Hsiung Hsu, Murray Campbell, "Deep blue," 2002. [Online]. Available: https://core.ac.uk/download/pdf/82416379.pdf

[4] F.-H. Hsu, *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton and Oxford: Princeton University Press, 2002.

[5] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[6] ——, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[7] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[8] J. Schaeffer, N. Burch, Y. Björnsson, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Checkers is solved," *science*, vol. 317, no. 5844, pp. 1518–1522, 2007.

[9] "What is the average number of legal moves per turn?" https://chess.stackexchange.com/questions/23135/what-is-the-average-number-of-legal-moves-per-turn, 2018.

[10] "What is the average length of a game of chess?" https://chess.stackexchange.com/questions/2506/what-is-the-average-length-of-a-game-of-chess, 2013, accessed: 20/06/2021.

[11] R. M. Ingersoll, *Play Hive Like a Champion*, 2nd ed. 5808 Southport Drive, Port Orange FL 32127: CreateSpace Independent Publishing Platform, February 2015.

[12] K. Hoki and T. Kaneko, "Large-scale optimization for evaluation functions with minimax search," *Journal of Artificial Intelligence Research*, vol. 49, pp. 527–568, 2014.

[13] B. U. Konz, "Applying monte carlo tree search to the strategic game hive," 2012.

[14] A. Y. Rikard Blixt, "Reinforcement learning ai to hive," 2013. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:668701/FULLTEXT01.pdf

[15] T. Bunth, "Solving hive board game with deep reinforcement learning," 2019. [Online]. Available: http://tdk.bme.hu/VIK/DownloadPaper/Mely-megerositeses-tanulas-alapu-Hive-jatekos

[16] C. M. McGuile, "Swarm artificial intelligence in hive," 2020. [Online]. Available: https://info.cs.st-andrews.ac.uk/student-handbook/files/project-library/cs5099/cmm40-Final_report.pdf

[17] D. de Goede, "Enhancing a hive playing engine with reinforcement learning," July 2021.

[18] H. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F. Seinstra, C. Snoek, and H. Wijshoff, "A medium-scale distributed system for computer science research: Infrastructure for the long term," *Computer*, vol. 49, no. 05, pp. 54–63, may 2016.

[19] "Elo rating system," July 2021, accessed: 11/07/2021. [Online]. Available: https://en.wikipedia.org/wiki/Elo_rating_system

[20] J. v. Neumann, "Zur theorie der gesellschaftsspiele," *Mathematische annalen*, vol. 100, no. 1, pp. 295–320, 1928.

[21] H. Baier and M. H. Winands, "Mcts-minimax hybrids," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, no. 2, pp. 167–179, 2014.

[22] ——, "Mcts-minimax hybrids with state evaluations," *Journal of Artificial Intelligence Research*, vol. 62, pp. 193–231, 2018.

[23] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

[24] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[25] A. Reinefeld and T. A. Marsland, "Enhanced iterative-deepening search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, no. 7, pp. 701–710, 1994.

[26] T. A. Marsland and M. Campbell, "Parallel search of strongly ordered game trees," *ACM Computing Surveys (CSUR)*, vol. 14, no. 4, pp. 533–551, 1982.

[27] D. Breuker, J. Uiterwijk, and H. Van Den Herik, "Information in transposition tables," *Advances in Computer Chess*, vol. 8, pp. 199–211, 1997.

[28] S. Gelly and Y. Wang, "Exploration exploitation in go: Uct for monte-carlo go," in *NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop*, 2006.

[29] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2623–2631.