Kalah Computer Tournaments for Teaching Students Adversarial Search

Case Study and Advice

Tobias Völk <tobias.voelk@fau.de>

Abstract

Following the tradition of hosting Kalah computer tournaments for teaching students about adversarial search as part of the „Artificial Intelligence I“ course at the University of Erlangen-Nuremberg, a tournament between teams of university students was hosted with, in contrast to earlier tournaments, the choice of programming language being left to the students and the option of testing programs against programs of other teams on a website prior to the actual tournament. Code submissions indicate that a reasonable number of teams were highly motivated, although the overall playing strength was far lower than expected, based on the concepts taught in the course. Most teams opted for a minmax approach with most of the minmax programs using alpha-beta pruning. A few teams resorted to simple heuristics instead of adversarial search. Only one team submitted a program using Monte-Carlo Tree Search. The approaches of the students as well as the setting of the tournament are discussed and advice for future practitioners and participants is given.

Contents

1 Kalah

The game of Kalah, oftentimes incorrectly referred to as „Mancala“, was invented by William Julius Champion in 1940 and is part of the Mancala game family. It is a discrete, deterministic two-player zero-sum board game with perfect information.

A board consists of a circle of two types of holes: „houses“, also called „pits“, and „stores“. A hole contains a non-negative number of „seeds“, also called „counters“. While the original game of Kalah uses a board of size 6 with 4 initial seeds in each house, it has been generalized to different board sizes with different numbers of initial seeds per house, with „(m, n)-Kalah“ referring to a board size of m houses per player with n initial seeds in each house.

Figure 1 The initial position of (6, 4)-Kalah, each house contains 4 seeds, both stores are empty. The large empty hole to the right is the southern store, the six small southern holes are the southern houses. Seeds are moved counter-clockwise. [9]

The player that goes first will be referred to as „South“ and the other player as „North“. For each player there are m consecutive houses followed by the store. The order of the holes in the circle is: southern houses, southern store, northern houses, northern store, making $2m + 2$ holes in total. The game is won by the player that has more seeds in their store at the end of the game. In the event of both stores containing an equal number of seeds at the end of the game, the result is a draw. A move consists of choosing one of one's own houses containing at least one seed, removing all of them and dropping them one by one, starting with the next hole, following the circle, skipping the opponent's store, but not the own store. A move can end in three ways:

(1) The last seed is dropped into one's own store. In this case, the player moves again. The move is referred to as a „bonus move“ (figures 2 and 3).

(2) The last seed is dropped into an own empty house and the corresponding opponent's house (same distance to the southern store if the circle was undirected) contains at least one seed. In this case, the seeds of these two houses are moved to the store of the player that made the move (figures 5 and 6).

(3) Neither of the two cases above, the move ends (figures 2 and 4).

In the event of all houses of the player that is to move being empty, the remaining seeds of all houses are moved to the store of the other player and the game ends. It is worth noting that the game can be terminated early if the store of a player contains more than half of all seeds as no sequence of moves can alter the outcome.

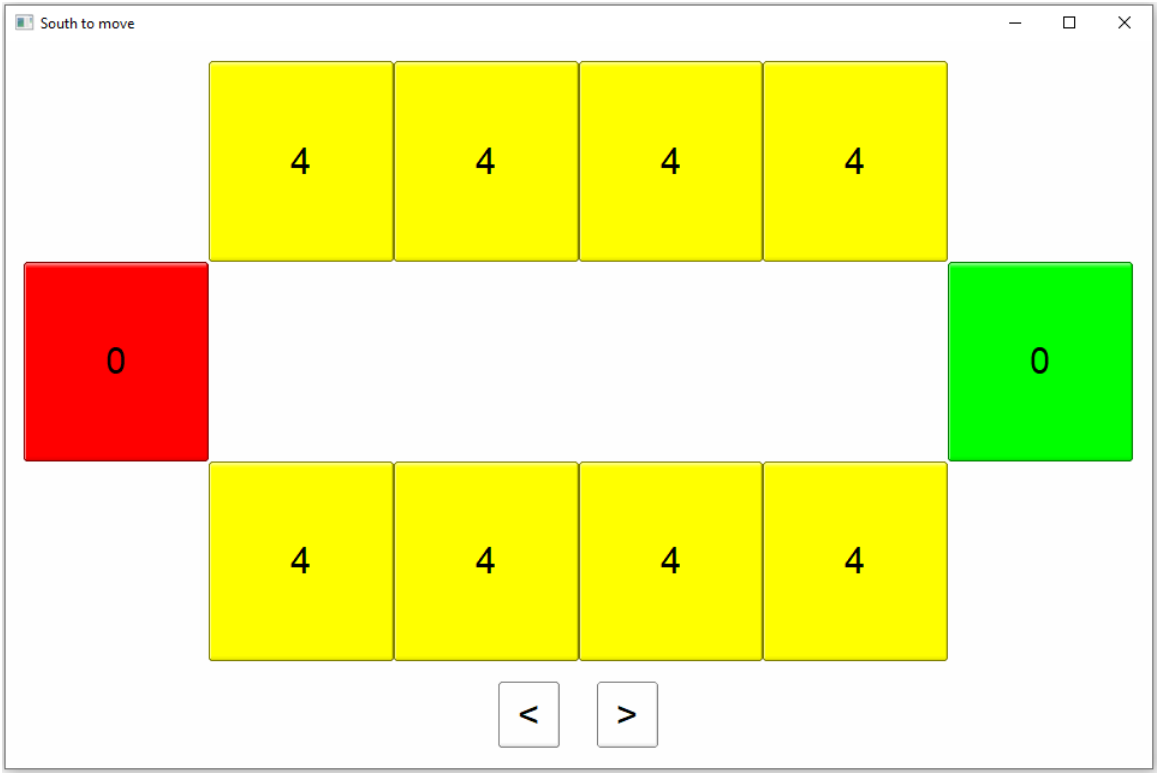

Figure 2 Cases 1 and 3: The initial position of (4, 4)-Kalah, it's South to move. Depending on the chosen move, we get either case 1 or case 3.

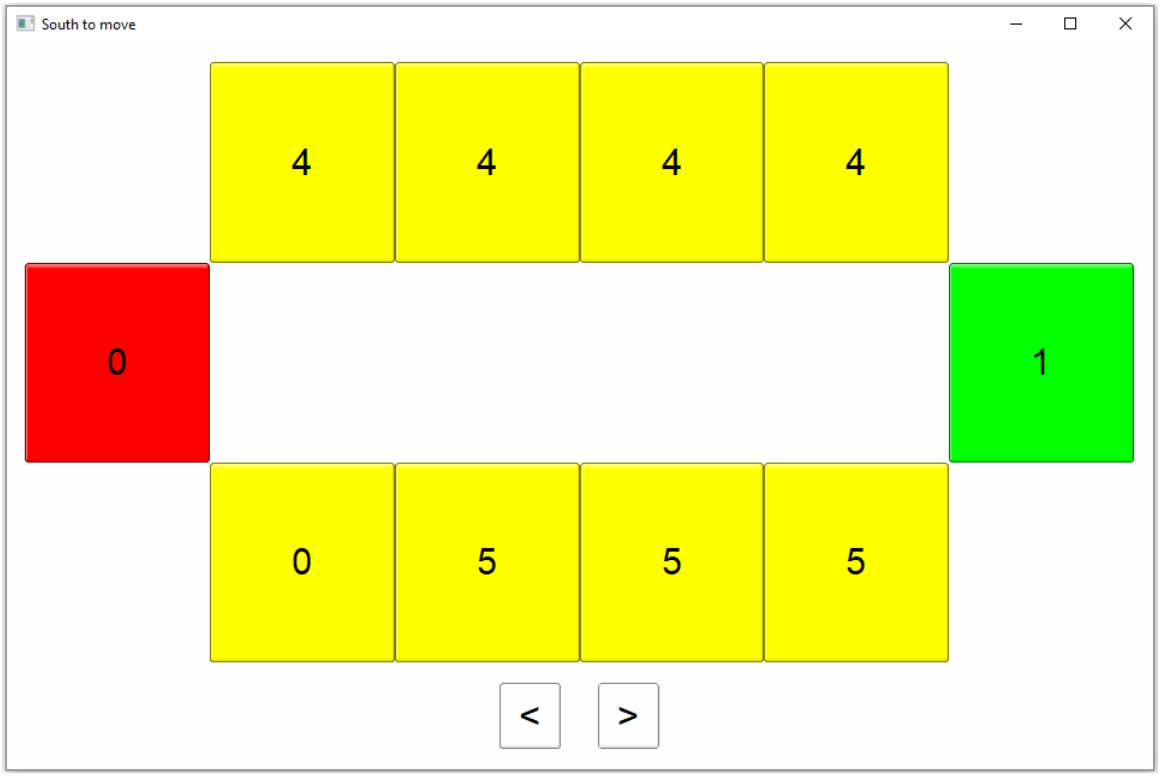

Figure 3 Case 1: South chose the leftmost house. Because the last seed ended up in South's store, it is South's turn afterwards.

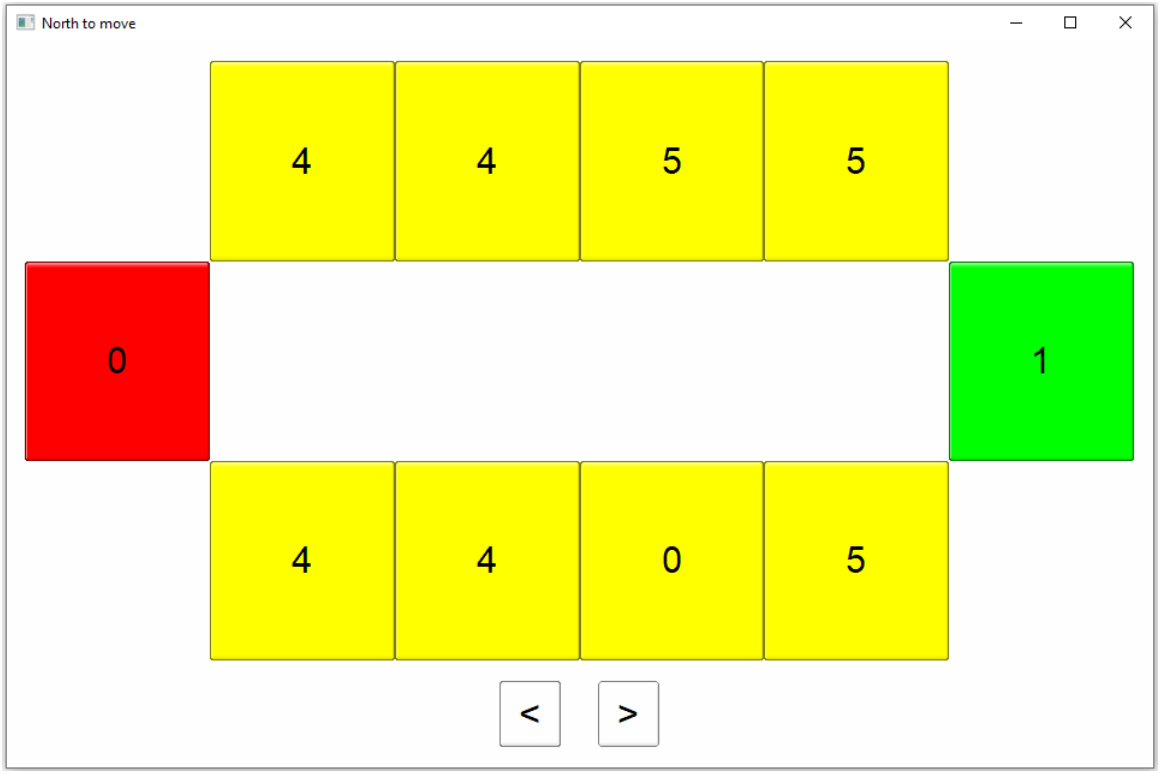

Figure 4 Case 3: South chose the third house from the left. It is North's turn afterwards.

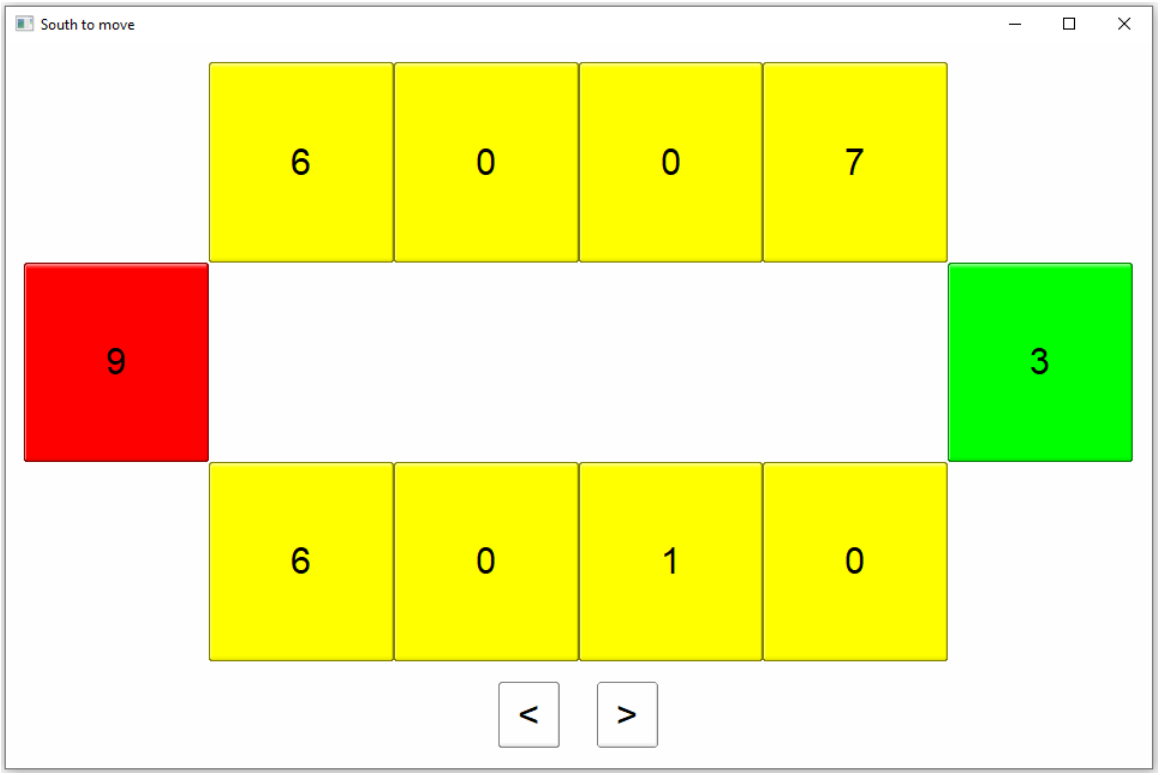

Figure 5 Case 2: In this position South can make a capture by choosing the third house from the left.

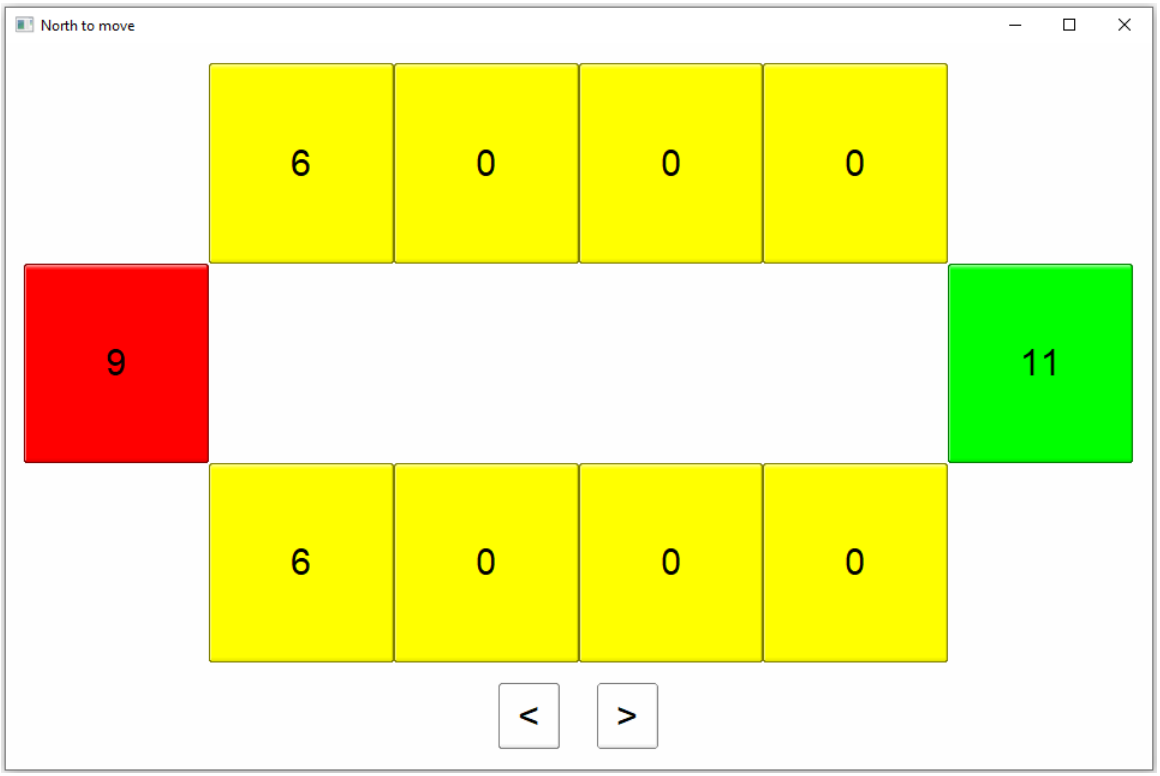

Figure 6 Case 2: South made a capture by choosing the third house from the left, capturing the 7 seeds in the corresponding northern house. It is North's turn afterwards.

2 Approaches and Advice

In this chapter, we briefly explain some important concepts while also giving advice to future participants, based on our own experience with writing Kalah programs. Some of those concepts were taught in the course and are subject of Artificial Intelligence courses at other universities as well. See the appendix for more detailed sources on these concepts.

The game of Kalah can be tackled using a variety of adversarial search algorithms such as Monte-Carlo Tree Search (MCTS), minmax-based algorithms, hybrids of MCTS and minmax-based algorithms, as well as other approaches [3] [5] [6] [7]. Note that, in our experience, heuristics and modifications to the basic search algorithm have a greater influence on the playing strength than the choice of the underlying search algorithm. Furthermore, some of the concepts listed here can be modified, extended or combined in creative ways. In any case, major modifications to one's algorithm should be tested by having the new program play against the old one using different positions of varying board sizes from both sides.

2.1 Evaluation Function

An evaluation function assigns scores to positions. It attempts to assign positive scores to favorable positions and negative scores to positions which are favorable for the opponent. Ideally, a position is more favorable than another position if and only if its assigned score is larger. The performance of most adversarial search algorithms heavily depends on the quality of the evaluation function. Even the most sophisticated search algorithm cannot perform well, if it has no good idea about what kind of positions to strive for. Computing the evaluation function can be expensive and leave less time for the actual search, which in turn can make up for a bad evaluation function. Thus, a balance between evaluation accuracy and computational cost has to be found. An intuitive and simple evaluation function for Kalah is the difference between the number of seeds in one's own store and the opponent's store, but in our experience, this evaluation function can be improved upon without mentionable increases in computational cost. Attempts at creating linear evaluation functions based on predefined features and weights, computed by a genetic algorithm, can be found in the literature [4].

2.2 Minmax Algorithm

The minmax algorithm considers the tree of possible game continuations from the current position, with nodes representing positions and directed arcs representing moves leading from one position to another. The outcome of a position can be determined at the nodes

at which the game is over. The algorithm then computes the outcome of the remaining nodes by assuming perfect play from both sides, meaning that, for a given node, it chooses the move which leads to the most favorable outcome for the player that can choose the move in that position, and assigns the outcome of that child node to the parent node. In practice it is, with the exception of small board sizes and later phases of the game, mostly infeasible to do this computation. Thus, in practice, the search is stopped according to a certain criterium, for example based on the search depth i.e. the height of the search tree, and non-terminal positions are assigned heuristic values using an evaluation function, with positive values being favorable for one player and negative values being favorable for the other player. Thus, the first player picks the move which maximizes the score while the second player picks the move which minimizes the score, hence the name minmax algorithm. It is a common misconception that the minmax algorithm has to alternate between minimizing and maximizing, which is not the case. The minmax algorithm can also be used for games in which players may move repeatedly in certain situations, which corresponds to picking the maximum/minimum score reapeatedly. The minmax algorithm works especially well during the late stages of a game when terminal nodes are frequent [7]. It is worth noting, that the tree does not have to be stored in memory, but can be computed on the fly, by visiting nodes in a depth-first manner. The minmax algorithm is typically used together with iterative deepening, meaning that multiple minmax searches with increasing resource usage are executed until a resource limit, for example a search depth or time limit, is reached.

2.3 M & N Procedure

Minmax search is a special case of the M & N procedure [3]. Instead of taking on the score of the child node with the largest (or smallest) score, the score of the parent node is computed by a function taking the M largest (or N smallest) scores of the child nodes as input, hence the name. This function can be chosen in a way which causes the search to strive for positions in which there are multiple good moves available. Depending on the function used, other concepts applicable to minmax search, like alpha-beta pruning, cannot be used or have to be adapted accordingly.

2.4 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) works by repeatedly sampling games. It does so, by building a tree of possible game continuations in memory with nodes representing positions and arcs representing moves leading from one position to another. Starting with the root node of the current position, the tree is updated according to the outcomes of the sampled games. The sampling in turn, is guided by the tree. This first part of the sampling should

cause more promising moves and under-sampled moves to be chosen more often. The second part of the sampling, also called „playout“ or „roll-out“, is randomized, usually choosing moves according to a suitable probability distribution, for example uniformly at random. The outcome of the game is then mapped to a score, with wins being mapped to 1, draws to 0.5 and looses to 0. Each node stores the sum of the outcomes as well as the number of sampled games that used this node during sampling. At the end of a rollout, the first position of the sampled game which was not yet part of the search tree is added and the outcome of the sampled game assigned to it. This information is then backpropagated through the tree, meaning that all ancestors of that new node are updated accordingly. At the end of the algorithm, the move leading to the node that has been sampled most often is chosen.

The first stage of the sampling has to sample more promising nodes, i.e. nodes with higher scores, more often, in order to focus the search on the „promising“ game continuations as well as to eventually have nodes take on the score of their „best“ child. Additionally, it must not neglect nodes with few sampled games, as a score based on few samples is unlikely to be accurate. A typical way of balancing these conflicting objectives is the UCT formula (Upper Confidence Bound 1 applied to trees) [2].

While choosing moves uniformly at random is a simple probability distribution to use during rollouts, different distributions can improve playing strength, even including distributions biased towards „bad“ moves. It is possible to do multiple rollouts at a time, but using one rollout only seems to be best for Kalah [7]. It can make sense to trade sampling accuracy for a higher number of samples by terminating rollouts early and guessing the result, for example if one player has a far larger number of seeds in their store. It is possible to replace the rollouts with an evaluation function or even a low-depth minmax search in which case the first part of the sampling should be adapted accordingly. It is also possible to combine rollouts and evaluation functions, for example by randomly or deterministically terminating rollouts and using the score of an evaluation function as the „outcome“ of the sampled game.

MCTS seems to play better during the beginning of a Kalah game while a minmax search seems to play better towards the end of the game, when terminal nodes become more frequent. A hybrid approach seems to have the best performance. [7]

2.5 Alpha-beta Pruning

Alpha-beta pruning extends the minmax algorithm by ignoring branches which cannot influence the result at the root node. To put it simply, the minmax algorithm with alpha-beta pruning does not waste time computing how good or bad a position is, when

it already knows, that the opponent or player would not allow this position to happen in the first place, due to better choices being available at earlier positions. It does so, by remembering a lower bound „alpha“ and an upper bound „beta“ for the heuristic value/outcome, assuming that the true value resides within that window. If the score of a position is within the bounds the search was started with, then the score is correct. This is of course always the case, if the initial bounds are set to the lowest and highest possible scores of the evaluation function. When the score is outside of the bounds, a score outside of the bounds is returned, indicating that the true score is either below or above the given window. Thus, one may start the search with a small window, not knowing whether this window contains the actual score, and repeat the search with a larger window afterwards, if necessary. Searching good moves first narrows the search window early and thus speeds up the search for the remaining moves. It is for this reason that alpha-beta pruning heavily benefits from a good move ordering, which in practice oftentimes speeds up the search by several orders of magnitude. In the worst case, considering a search tree in which every non-leaf node has the same number of children, the extended algorithm searches the same nodes as the pure minmax algorithm and in the theoretical best case, twice as deep while only considering the same number of nodes. Thus in practice, the minmax algorithm is always used with alpha-beta pruning and a great effort is made to find a good move ordering based on heuristics and/or information from previous searches. However, the cost of the move ordering should not be so great as to slow down the entire search despite visiting fewer nodes. It makes sense to use move ordering based on more expensive heuristics at earlier depths, since more time is saved in the event of a cut-off. This could include low-depth searches. Note that information from previous searches can be saved and used for improved move ordering at low computational cost.

2.6 Zero-Window Search

A zero-window search is a minmax search using alpha-beta pruning with the upper bound of the alpha-beta window being larger than the lower bound by 1, thus leaving a gap of one. The evaluation score returned by a zero-window search only reveals whether the value used as bound is a lower or upper bound to the true score, but the search is usually much faster than a normal search. Zero-window searches are part of many other search algorithms, oftentimes used as a quick way to disprove less promising moves, followed up by a normal search if this proof fails and the exact value needs to be known.

2.7 Principal Variation Search

The name Principal Variation Search (PVS) comes from the sequence of best moves sometimes being reffered to as the „principal variation“. The move that is guessed to

be the best move, for example because it was the best move according to the previous search, is searched first using the minmax algorithm with alpha-beta pruning. Its resulting evaluation score is then used to search the remaining moves with a zero-window search in order to quickly prove that the remaining moves are not better. In the, ideally rare, case that one of the remaining moves is better, the search of this move is repeated with a full window in order to compute the exact score. In this case, it would have been faster to do a full-window search immediately and time has been lost.

2.8 MTD-f

The MTD-f algorithm (Memory-enhanced Test Driver with first guess 'f') determines the value of a minmax tree using repeated zero-window searches, each of them improving the lower or upper bound until upper and lower bound meet. The performance of MTD-f heavily depends on a good initial guess, which of course can be based on previous searches, and the way of choosing new guesses based on new bounds. It does not perform well for fine-grained evaluation functions because the large number of different evaluation scores usually results in many zero window searches until lower and upper bound become exactly equal. In the case of Kalah, it is easy to come up with sensible non-fine-grained evaluation functions, for example, based on seed counts. When using a fine-grained evaluation function, one can switch to using the minmax algorithm with alpha-beta pruning once the bounds get close.

2.9 Pruning and Focusing the Search

The decision to ignore a part of the tree of game continuations due to some criteria, is called „pruning“. There are speculative, unsafe ways of pruning as well as safe ones. Alpha-beta pruning is an example of safe pruning, that is, pruning which does not change the result of the search. Another example of safe pruning is Futility pruning, which uses game specific upper and/or lower bounds on the score during an alpha-beta search to improve the bounds used by the alpha-beta search, for example by realizing that the player cannot reach the lower bound of the search window, even if all remaining seeds, meaning seeds that are not in a store, would end up in their own store.

Speculative or unsafe pruning bears the risk of pruning branches that would have changed the outcome of the search. It can also be interpreted as a focus of the search effort on certain parts of the tree. Note that it is extremely beneficial to the playing strength of a program to focus its efforts on promising parts of the search tree while neglecting others. A simple example of this would be to search positions, in which there is no capture or

bonus move available, with lower depth. This is of course equivalent to searching positions with a bonus move or capture with higher depth.

2.10 Traps and Playing in Lost Positions

A minmax-based algorithm usually assumes that the opponent plays the best move according to the tree of game continuations searched. However, this assumption does not always make sense. In lost positions, especially against weaker opponents, it might make sense to choose a bad move which is good under the assumption, that the opponent's search algorithm will not play the complicated refutation of that move. Striving for traps or positions in which a weak opponent is more likely to go wrong might make more sense than trying to loose more slowly or by a smaller margin. A common mistake of strong programs, when they have proven a position and thus all available moves to be lost, is to play random moves instead of putting up resistance, even though the opponent might not have realized yet that the position is won and thus might not play the objectively best moves.

2.11 Memory

It is oftentimes, although not always, worthwhile to store information coming from the current search in memory and reuse it for later searches. For example, in the game of Kalah, it is possible to „transposition“ into a position, which means to reach a position, that can also be reached using a different sequence of moves. If the score of a position is saved during a search, transpositions into that position can be recognized and recomputations avoided. However, in our experience, the use of a transposition table slows down the search, as the overhead is computationally more expensive than the time saved by not re-searching positions. Storing information about positions is beneficial nonetheless, if the information is put to good use. For example, saving move orderings based on previous searches can be used to improve the move ordering of future searches. When using zero-window searches for disproving certain moves, the bounds on the true scores proven by these searches can be saved in order to narrow the alpha-beta window of future searches with larger windows. Storing bounds on the score of a position or even the exact score, possibly including the resources, like search depth, used to compute that score, can be used to guide the search. For example, this information could be used to decide which moves to search first, which to search with a full window and which to search with a smaller or even a zero-window for quick refutation. In the case of Monte-Carlo Tree Search, it is obvious but important not to delete the entire tree built in memory, but to replace it with a subtree of the existing tree accordingly. This can even be achieved when multiple moves have been played inbetween, for example, by using a breadth-first-search, which requires much less computation time

than the „recomputation“ of this subtree. Since Monte-Carlo Tree Search stores the tree in memory anyway, transpositions can be avoided without using additional memory by storing the tree in a hash map instead of a tree-like data structure. It should also be noted, that one can use a 64-bit hash value of the position as key to the hash map, since, in practice, it is sufficiently unlikely for two 64-bit hash values to have the same value.

2.12 Perfect Play and Databases

The game of Kalah has been solved for all starting positions of board size $1 \leq m \leq 6$ with $1 \leq n \leq 6$ initial seeds per house [1] [10]. For larger board sizes it is possible to create databases for all positions with at most a certain number of remaining seeds, that is, seeds that are not in a store. This information can then, ideally in combination with a minmax search, be used to achieve perfect play towards the end of the game. These types of game-databases are referred to as „endgame databases“ oftentimes. All N positions of the same board size and number of remaining seeds can be assigned different integers from 0 to $N - 1$ using simple combinatorics („Stars and bars“). Thus only the scores of the positions need to be saved in an array indexed by these integers. Therefore, kalah endgame databases do not waste space and have constant access time in the number of positions stored. Since the contents of the stores do not affect the optimality of a move, they are assumed to be zero for all positions in the database and added to the looked-up score afterwards. There are $\binom{m+r-1}{r-1}$ positions with board size m and r remaining seeds. Table 1 gives a rough idea about how feasible endgame databases are for various board sizes. Of course, this can be improved by using the database in combination with a minmax-based algorithm. Our experience with writing kalah-playing programs is, that improvements in the search algorithm or evaluation function result in larger gains in playing strength than the use of endgame databases, with the obvious exception of small board sizes and boards with very small numbers of remaining seeds. There also is a danger to using endgame databases: If a program knows that the position is lost with perfect play from both sides, but the opponent does not, they might not distinguish between the moves and thus play random moves instead of putting up resistance. Databases can also be created for the early phase of the game, for example by letting the program precompute results for positions that are guaranteed or likely to be encountered at the beginning of a game. This type of database is oftentimes referred to as an „opening database“. It might make sense to prohibit the use of opening databases as well as endgame databases, since they are not directly related to adversarial search.

Board size	Number of seeds
4	46
5	30
6	23
7	19
8	16
9	15
10	13
11	12
12	11

Table 1 Number of seeds s.t. all positions with at most that number of remaining seeds fit into 1 GB of memory, using 1 byte for each position.

2.13 Horizon Effect

The Horizon Effect refers to the difficulty of finding a good evaluation function for non-terminal nodes in a minmax search. In the case of Kalah, this is because of its highly tactical nature. While an evaluation function might consider a position to be favorable, the opponent might be able to win a large number of seeds in the next few moves, for example by means of a forced sequence of bonus moves followed by a capture or zugzwang (from German „Zugzwang“), which means to be forced to move despite not moving being preferable. Continuing the search as long as bonus moves or capture moves are available is oftentimes unfeasible and the detection of zugzwang situations is, in our experience, very difficult. Zugzwang is a common phenomenon in Kalah, since emptying all of one's houses results in all remaining seeds, oftentimes a large, game-deciding number, being moved to the opponent's store. Thus, most of the time, one would not want to make the move which empties one's last house. While this by itself would be easy to recognize, this forced emptying of houses can stretch across multiple moves, with the opponent repeatedly making moves which do not add seeds to one's houses, oftentimes even permitting captures in order to empty one's houses faster or adding some seeds to prevent capture moves by turning them into non-capture moves or capture moves that win less seeds. These highly tactical move sequences become a problem when they are „behind the horizon“ of the search, meaning that the search either decided not to search further or was not able or willing to spend the computational resources to search further, hence the term „Horizon Effect “. Reliable recognition or inclusion these dynamics into the evaluation function at acceptable computational cost appears to be difficult [8]. The strength of the Horizon Effect in Kalah could be due to the fact, that a single move can change the number of seeds in multiple houses, thus changing the behaviour of the moves associated with those houses and thus changing the characteristics of the position. We had no difficulties finding

positions, in which even slight modifications to the number of seeds in the houses changed the outcome with perfect play from both sides.

Some approaches for tackling the Horizon Effect are:

- Increasing the search depth in hope of recognizing bad estimations resulting from the Horizon Effect long before the position is reached and being able to avoid them due to a large number of alternative moves, including good moves.
- Trying to end the search at nodes which are less likely to suffer from the Horizon Effect, for example by not ending the search on the opponent's turn with multiple moves being available to the opponent.
- Trying to incorporate this uncertainty into the adversarial search algorithm, for example by backing up scores in a certain way and/or backing up uncertainty as well as the score and considering this backed-up uncertainty when making the decision about which move to play.

One has to be careful not to continue the search in a way that prevents it from terminating. While this problem might not appear for small board sizes, it oftentimes does for larger board sizes, due to the greater number of moves, for example, when continuing the search as long as there is a certain type of move available. It is a common mistake to try to continue the search too often, resulting in a search that is not going to terminate in a reasonable amount of time. Note that the computational cost of trying to handle the Horizon Effect should match the risk and not slow down the search too much, especially considering that a deeper search makes misevaluations due to the Horizon Effect less likely and notices misevaluations earlier, thus providing more chances of encountering a possibility to avoid reaching the position misevaluated earlier.

3 Tournament and Background

The use of Kalah computer tournaments for teaching adversarial search in a fun and motivating way has shown promise in the past [11]. Encouraged by these results and our experiences with hosting Kalah computer tournaments, we attempted to improve upon the tradition of hosting a Kalah tournament as part of the „Artificial Intelligence I“ course at the university of Erlangen-Nuremberg, by letting students use any programming language and providing the option of testing their programs against other programs prior to the actual tournament, neither of which has been the case in previous tournaments. The tournament was part of a voluntary homework for which the students could receive extra credit. The students mostly worked in teams of three or four students and were given several weeks for the development of their kalah-playing programs. There were no restrictions on what algorithm to use, even primitive heuristics and the use of opening and endgame databases were permitted. We truthfully announced that the programs would be given one CPU core only, in order to make the students focus on improving and learning about the actual algorithm instead of parallelization. The tournament was based on the KGP (Kalah Game Protocol) [12] which was developed for this and future tournaments. In the KGP, clients receive positions from the server to which they reply with moves until the server notifies the client, that it is no longer interested in receiving moves for a certain position, for example because a time limit has been reached. After that, the last move which reached the server in time is played. Libraries implementing the KGP as well as game logic were provided for the programming languages Java and Python, but students could use other programming languages, provided their code conformed to the KGP. Template programs using the minmax algorithm were made available for Java and Python in hope of relieving students with weaker programming skills of the burden of implementing this algorithm and enabling them to spend more time improving upon the template in creative ways, with more skilled students possibly ignoring the template and opting for an entirely different algorithm, such as Monte Carlo Tree Search.

In an attempt to further increase motivation, the students had the option of having their program connect to a training server to be matched against other programs, prior to the actual tournament. In order to keep track of a program inbetween reconnections, the programs identified themselves to the server using secret tokens, which were chosen by the students. Furthermore, the students could choose program name, author and description, which were displayed on the website, thus being able to participate anonymously or give additional details about their program, if desired. In order to give students the possibility of testing their programs at any time, as well as getting an idea of the playing strength of their programs, we connected several programs of ours to the training server, which were always online. Some of our programs chose their moves either uniformly at random from

the available moves while others performed a minmax search with a fixed low depth in the range 1 to 5, using the store difference as the evaluation function. The algorithms of our programs were revealed through the program names on the website, including search depth. The strength of all programs was measured using the Elo system and this ranking was displayed on the website. The matching was performed uniformly at random among the programs which were not currently playing, with large rating differences being avoided. For each matching, the board size and number of initial seeds per house were chosen randomly and two games were played with each player moving first in exactly one game, in order to eliminate the advantage of moving first.

Finally, the programs participated in the actual tournament which was run on a single machine, thus removing any advantages due to hardware. The submitted programs had to reliably beat a program which chooses moves uniformly at random in order to qualify for the tournament. Extra credit was awarded for a successful qualification with further extra credit being awarded depending on the placement in the tournament. Both the qualification test and the tournament were played with 5 seconds of computation time per move. The tournament consisted of multiple round-robin rounds with increasing board size and number of initial seeds per house. Each program played every other program twice, once as South and once as North, in order to eliminate the advantage of moving first. One point was awarded for a win, half a point for a draw and zero points for a loss. After each round, the points were reset, and the worst programs were eliminated according to the availability of time and computational resources for the execution of the remaining tournament. The programs were restarted between games but could store information in memory across moves. To save computing resources and run more matches in parallel, the programs were only given CPU time when it was their turn.

4 Results and Discussion

After removing heavily bugged and plagiarized submissions, 30 programs remained, 25 of which qualified for the actual tournament by defeating a program that chooses moves uniformly at random. This section discusses these 30 programs only, since the remaining programs cannot be considered serious attempts at implementing and submitting strong programs. Teams that were affected by bugs were contacted and had the opportunity to fix their submission and ask for help. In later considerations, only the top 15 programs are considered. This was necessary due to some programs crashing during the tournament and thereby manipulating the score in a way that does not reflect the performance of the programs which these considerations rely on. Table Table 3 shows various properties of these top 15 programs and will be used repeatedly throughout this section.

4.1 Training Server Usage

At the end of the tournament the leaderboard of the website contained 326 entries, greatly exceeding the number of teams. The program names used on the leaderboard indicate that most entries refer to the same program or slightly different versions of the same program. The number of entries on the leader board also exceeded the number of actively playing programs by far, causing large parts of the leaderboard to not reflect the actual playing strength due to outdated ratings. While some students chose to identify themselves on the training server using their name, university ID or alike, the majority of students chose nonsensical program names and author names and thus remained anonymous. Almost no student sent a description of their program to the training server and when they did, they sent nonsense. Furthermore, the rating of the programs fluctuated heavily and stabilized only after a large number of games. This would have required the students to run their program for many hours, which most of them did not do, despite the availability of a machine with high core count for that purpose. Hence the ratings were only a rough indicator with new programs needing many hours in order to get a serious estimation of their playing strength. Observation of training server activity and code submissions as well as post-tournament verbal communication with some participants revealed that at least four teams were actively using the training server for testing different approaches and versions of their programs, considering the rating displayed on the website to be a sufficiently accurate indicator of playing strength for that purpose. Assuming that the students used similar names for their programs in the actual tournament, the training server placement roughly reflected the tournament placement with 12 of the tournament's top 15 programs being found in the training server's top 50 under the same or similar names.

4.2 Programming Languages

With different programming languages being permitted, one might be concerned about the performance gaps between them. Among the top 15 programs, only the second place was written in C++, called by Python code, seven programs were written in Python without resorting to other languages and the remaining seven programs were written in Java. However, only one Python program became part of the top 5, using a low-effort but clean implementation of the minmax algorithm with alpha-beta pruning without move ordering and the store difference as evaluation function, ending up one place below a Java program with the exact same properties. This might indicate, that the choice of programming language does not make that much of a difference. The remaining Python programs ended up in the lower half of the top 15. Whether this disadvantage is due to the programming language or, for example, motivated students preferring different programming languages, cannot be said. The distribution of programming languages used can be found in Table 2.

4.3 Library Usage and Self-Implemented Game Logic

Every team used either the provided Java or Python library. When asked about this, students gave high implementation effort, risk of implementation errors and the unwillingness to invest time reimplementing the game logic, as reasons. Thus, in practice, the choice of programming language was always restricted to the possibility of using an existing library, either be using the language the library was written in (Python, Java), a related language (Kotlin) or by using the language the library was written in and calling code written in another language (Python calling C++ or C).

Out of the five teams which implemented the game logic themselves, two of them being forced to do so due to their choice of programming language, all but one ended up in the top 6, hence self-implementation of the game logic was strongly correlated with better tournament performance. Observation of the training server activity revealed that all but one of those five teams spent large amounts of time testing their programs on the training server, possibly indicating high motivation in general.

Programming Language	Count
Python	16
Java	11
Kotlin	1
Python calling C++ code	1
Python calling C code	1

Table 2 The distribution of programming languages used among the 30 submissions.

4.4 Algorithms

Out of the 30 programs, four teams did not implement adversarial search but simple, computationally inexpensive heuristics. Examples include choosing the move with the largest number of seeds in the house corresponding to the move as well as choosing the move which maximizes the number of seeds in the own store, looking one move ahead. Minmax-based approaches were chosen by 25 teams with four teams implementing the minmax algorithm without alpha-beta pruning, 20 teams implementing the minmax algorithm with alpha-beta pruning and one team implementing Principal Variation Search (PVS) which makes use of the minmax algorithm with alpha-beta pruning. One of the teams implementing the minmax algorithm with alpha-beta pruning also, according to code remains and comments submitted by that team, implemented MTD-f („Memory-enhanced Test Driver with first guess f“) as well as the minmax algorithm with alpha-beta pruning, storing positions and their scores in memory in order to avoid recomputations, but considered both of these approaches to be worse and thus did not submit them. It is worth mentioning that neither PVS nor MTD-f were part of the course. These two teams and seven other teams used move ordering, the remaining eleven teams did not. There was only one submission using what could be considered Monte Carlo Tree Search (MCTS), despite MCTS being discussed in the lecture. Only two teams, places 1 and 10, implemented pruning techniques, although the latter one likely was based on the wrong assumption that bonus moves in Kalah are always preferable. Finally, there is no evidence of attempts at using machine learning approaches such as artificial neural networks or decision trees, neither as algorithm, nor as evaluation function of a search or otherwise. No team used databases or any other precomputed results, neither for the initial moves, nor for improved play towards the end of a game.

4.5 Understanding and Implementation of the Minmax Algorithm

Some teams that implemented the minmax algorithm did so by duplicating their code into two functions called „min“ and „max“. Since this is error-prone and results in more

maintenance work, it might indicate that those students did not understand this algorithm well enough and thus did not feel confident enough to negate the evaluation scores correctly in a single function. This is especially irritating, since the provided templates do this correctly already. Furthermore, a number of students attempted to implement alpha-beta pruning on top of the template in a way that indicates, that they did not understand alpha-beta pruning, for example, by not negating the evaluation score and/or bounds correctly after making a move or bonus move.

On the other hand, many teams did not implement move ordering for the minmax algorithm with alpha-beta pruning either. One might speculate that, despite it being mentioned in the lecture, the students were not aware of the importance of move ordering when using alpha-beta pruning, since it does not take much effort to implement a simple move ordering based on heuristics.

Finally, several teams limited the search depth of their minmax algorithm despite using iterative deepening. We have no explanation for why they did this, since it is a well-known fact that larger search depth results in better performance. A search depth limitation is especially disadvantageous towards the end of the game, when the average branching factor of the search tree is smaller and large search depths become feasible.

4.6 Understanding and Implementation of Monte-Carlo Tree Search

The only team which submitted, what could be considered to be a Monte-Carlo Tree Search (MCTS) program did not seem to understand the concept of MCTS. Starting with a tree consisting of the root node, their program adds randomly played games to the tree repeatedly, the score of a node being the average score of its child nodes. They did not use the tree they built to bias the sampling in any way. Thus, given enough time, their program chooses the move which leads to the subtree with the best average over the terminal nodes of that subtree. This sampling of random games from the root position is known as „Pure Monte Carlo Search“ and it is not an effective method for the game of Kalah. The complexity of their code suggests that they have invested a large amount of time. The unnecessary creation of a search tree might suggest that they did not realize what their code was doing.

4.7 Evaluation Functions

We'll refer to the difference between the sum of the seeds in one's own houses and the sum of the seeds in the opponent's houses as „house difference“ and the difference between the

number of seeds in the own store and the number of seeds in the opponent's store as store „difference".

The store difference was used by 16 teams although one team did not implement adversarial search and only looked one move ahead instead. The code of one of these teams strongly suggests that they intended to use a linear combination of two features instead but canceled out the second feature by accident. One team came up with an unusual evaluation function, modifying its value throughout the search algorithm: The score was multiplied by 1.1 every time a bonus move was played and by 1.05 every time a capture was made, thus having their evaluation grow exponentially with the search depth, reinforcing good and bad scores alike. Their evaluation function consisted of a polynomial of degree 5 based on the ratio of the number of seeds in all houses to the number of total seeds, which was then used to weigh the store difference and house difference in a way that decreases the weight of the store difference and increase the weight of the house difference towards the endgame. This is a sensible choice, since the event of one player running out of moves and thus the opponent winning all remaining seeds, is far more likely in the endgame. Another team used a quadratic function of the stores, houses and the number of bonus moves since the root of the search tree and the fact whether the rightmost own pit was empty, indicated by a value of one or zero.

Ten teams used linear combinations of features, most often based on the stores and houses, two of them seem to have used features and weights from the literature [4] [6]. Common features were the number of seeds in the stores and the number of seeds in the houses. Other used features include whether the rightmost pit was empty, indicated by a value of zero or one, the number of bonus moves since the root position and the right to move. The weights used in the linear function were almost always rough-grained, using one decimal place at best. This might indicate, that these evaluation functions were not derived using an automatic approach but via a trial-and-error procedure. The code of at least three teams indicates that they have experimented with other and more complex evaluation functions but ultimately chose to use the store difference again. It should be noted that this is not a bad choice, as it is somewhat accurate and very cheap to compute, although, in our experience, it can be improved upon with almost no additional computational cost. One team reported improved performance when using random move ordering, something that we were not able to reproduce. In our experience, even primitive move orderings like ordering bonus moves and captures first vastly outperform random move orderings. The team that based most of its evaluation function on the literature placed 8th in the tournament, beating multiple programs using alpha-beta pruning with move ordering, despite using the minmax algorithm and thus having lower search depth. This is a clear example of a more accurate and computationally expensive evaluation function compensating for a worse search algorithm.

4.8 Performance and Motivation

In order to get an idea of the playing strength of the top 15 programs, we made them play against a simple java program in a trial-and-error procedure using various board sizes and numbers of initial seeds per house. The program uses the minmax algorithm with alpha-beta pruning with move ordering and the store difference as evaluation function. Its move ordering causes the program to search bonus moves before non-bonus moves, captures before non-bonus-non-capture moves, with moves closer to the own store being searched first in the event of a tie. With the exception of the top 3 programs, all programs performed worse. We expected this program to have average performance among the programs submitted by the students before hosting the tournament as it is easy to implement and only uses classic concepts presented in the lecture, namely minmax search with alpha-beta pruning with move ordering. This expectation was not met. Despite this slight lack of performance, the observation of the training server activity, submitted code and comments indicate that at least ten teams had their programs play on the training server regularly, indicating high motivation and time investment, which payed off. In contrast, places four, five and seven did not invest much effort. While similarily named programs could be observed playing on the training server from time to time, all three only implemented minmax search with alpha-beta pruning without move ordering and the store difference as evaluation function, yet their programs outperformed most other programs in the tournament. Curiously, this includes places 9, 10 and 12 which are Python programs which only differ in the fact that they use move ordering, which should make them play better. A look at the code revealed that place 9 limited the search depth to 6. Place 10, perhaps on purpose as a kind of pruning, misimplemented the minmax search by only considering one bonus move when at least one bonus move was available, ignoring the other moves. Perhaps this was based on a wrong understanding of bonus moves in Kalah. While, in small experiments of ours, bonus moves were the best moves the majority of the time, assuming perfect play from both sides, there are regular exceptions. While this kind of pruning enables the program to search deeper, it also causes it to overlook these regular exceptions as well as other, potentially even better, bonus moves. Unfortunately, this team also limited the search depth of its program to 12. Finally, place 12 built the search tree in memory without reusing it between searches, slowing down the search considerably and resulting in worse performance. This is a case of a superior algorithm performing worse than another algorithm written in the same language, namely place 5, due to a vastly slower implementation.

Place	Algorithm	Ordering	Language	Evaluation Function	Own Logic	Effort
1	Alpha-beta	Yes	Java	Stores	Yes	High
2	Alpha-beta	Yes	Python, C++	Stores	Yes	High
3	Alpha-beta	Yes	Java	Stores, houses and bonus moves	Yes	High
4	Alpha-beta	No	Java	Stores	No	Low
5	Alpha-beta	No	Python	Stores	No	Low
6	PVS	Yes	Java	Stores and houses	Yes	High
7	Alpha-beta	No	Java	Stores	No	Low
8	Minmax	-	Python	Seven features, six weights and features taken from literature	No	High
8	Alpha-beta	Yes	Java	Polynomial of degree 5 of stores, houses, ratio of sum of seeds in all houses to total seeds previous captures, bonus moves	No	High
9	Alpha-beta	Yes	Python	Stores	No	High
10	Alpha-beta	Yes	Python	Stores	No	Low
11	Minmax	-	Java	Stores	No	High
12	Alpha-beta	Yes	Python	Stores	No	Medium
13	Minmax	-	Python	Stores, houses, bonus moves and captures	No	Medium
14	Alpha-beta	No	Python	Stores	No	Low

Table 3 Properties of programs ordered by tournament performance. The underlying algorithm is given, with „Alpha-beta“ referring to a minmax algorithm using alpha-beta pruning and „PVS“ being short for Principal Variation Search. If not indicated otherwise, the evaluation function is a linear function of features of the properties listed. The second-last column indicates whether the students used the game logic of the library or implemented their own game logic. Finally, based on our experience with writing Kalah programs, we attempted to estimate the amount of effort spent by reading the submitted code and considering the training server activity. „Low“ refers to an estimated time usage of 1-2 hours, „Medium“ to an estimated time usage of 2-10 hours and „High“ to any estimated time usage beyond that.

4.9 Description of Top 3 Programs

Uncommented code of the java program that won first place indicates, that the corresponding team experimented with more complicated, non-linear evaluation functions but ended up using the store difference in the end. They implemented their own game logic, using a single array containing all holes of a Kalah board, which enabled them to undo moves. This is in contrast to the provided library which requires the boards to be copied instead, since undoing moves in Kalah is non-trivial and requires information about the move played before, as well as the number of seeds captured. They used the minmax algorithm with alpha-beta pruning with move ordering, which considered bonus moves before other moves, ordering non-bonus moves in decreasing order of the number of seeds captured. The move ordering was saved and reused using a hashmap, thus saving the time to compute and sort the list of legal moves while also profiting from the information gained from the previous search. Finally, they distributed the search effort based on their move ordering: Given ordered moves $m_0, m_1, m_2, \ldots, m_k$ of a position with m_0 being searched first, their program searches move m_i with depth $N - i$ instead of N, thus spending less time investigating less promising moves.

The 2nd place was taken by a python program calling C++ code. The corresponding team chose to use the minmax algorithm with alpha-beta pruning with the stone difference as evaluation, using a move ordering based on previous searches. They saved the best move after each search in order to search it first in future searches, in order to achieve more cutoffs. However, they accidentally searched this move twice, even though, due to alpha-beta pruning, the best possible lower bound was used during the second search. When saving the best move, they did not take into account the search depth used to determine the optimality of this move, resulting in lower-depth searches overwriting the results of deeper searches. This might have been done on purpose since, in our experience, the move ordering based on a search with slightly lower depth is a better indicator of the ideal move ordering than the move ordering produced by a far deeper search.

The team that placed 3rd implemented alpha-beta pruning in Java. Their program was the only one which started the iterative deepening at a depth dependent on the board size used. Their evaluation function is the sevenfold of the store difference, with 2 being added or subtracted depending on whether it is one's own turn or not. Their code also adds the difference between the sum of the southern houses and the sum of the southern houses to this evaluation function, which is always zero of course. It is unknown whether this is intentional. The moves were ordered according to this evaluation function.

5 Suggestions for Future Tournaments

There still exist major problems with the tournament in its current form, which prevent it from making the students learn about adversarial search in an effective way. This section suggests ideas for fixing these problems.

5.1 Measuring Playing Strength

The Elo system proved to be very inaccurate for determining the playing strength as the students' programs were not online frequently enough to play a large number of games. The rating did not reflect the playing strength and the Elo system should thus be replaced.

Test suites could be a possible idea, where a program is presented with a position and has to find the best move, assuming perfect play from both players. These tests can be generated using a strong minmax-based algorithm, possibly in combination with an endgame database as described in an earlier chapter. The number of positions solved correctly could be used as a measure of playing strength. In contrast to gameplay, which might require playing hundreds of moves for large board sizes, test suites only require one move per test position. Limited by the strength of the program used to create the test position, test suites can only use small board sizes or late-game positions for larger board sizes as the provably best move has to be computed. Since minmax-based algorithms perform better than Monte-Carlo Tree Search towards the end of a game [7], this might make students prefer minmax-based algorithms. Finally, test suites do not necessarily reflect the playing strength at the beginning of the game.

Another way of determining playing strength would be to have the programs, using various board sizes and numbers of initial seeds, play against a set of predefined programs and compute their rating as $2w + d$ with w being the number of wins and d being the number of draws. This approach cannot compare very weak or very strong programs of course, as they would beat the same number of programs.

5.2 Turning Motivation into Performance

A considerable number of students invested large amounts of time into the development of their programs, yet only three programs exceeded the playing strength of a simple minmax search using alpha-beta pruning, move sorting and the store difference as evaluation function. Since these and other more advanced concepts are not difficult to implement, yet greatly improve playing strength, this might indicate a lack of knowledge of the concepts, or a lack of knowledge of the importance of these concepts, both of which could be fixed

by, for example, making suitable resources, such as the „Approaches and Advice“ chapter of this work, available to the students.

5.3 Encouraging the Use of Monte-Carlo Tree Search

Only one team submitted a version of what could be considered a nonsensical version of Monte-Carlo Tree Search (MCTS), as they built a search tree in memory, but did not use it to bias the game sampling in any way. It is known that at least one other team experimented with MCTS but chose not to submit it. The availability of minimalistic template programs using minmax search might have made the students prefer minmax-based approaches. Some students later on revealed, that they considered the release of these templates as a hint, that the implementation of a minmax-based algorithm was expected and that these approaches were superior. Hence, it might be a good idea not to provide these templates and announce that minmax-based approaches are not per se superior to MCTS and that MCTS seems to play better during the beginning of the game. [7]

6 References

1 Irving, Geoffrey, Jeroen Donkers, and Jos Uiterwijk. "Solving kalah." Icga Journal 23.3 (2000): 139-147.

2 Kocsis, Levente, and Csaba Szepesvári. "Bandit based monte-carlo planning." European conference on machine learning. Springer, Berlin, Heidelberg, 2006.

3 Slagle, James R., and John K. Dixon. "Experiments with the M & N tree-searching program." Communications of the ACM 13.3 (1970): 147-154.

4 Divilly, Colin, Colm O'Riordan, and Seamus Hill. "Exploration and analysis of the evolution of strategies for mancala variants." 2013 IEEE Conference on Computational Intelligence in Games (CIG). IEEE, 2013.

5 Lanctot, Marc, et al. "Monte Carlo tree search with heuristic evaluations using implicit minimax backups." 2014 IEEE Conference on Computational Intelligence and Games. IEEE, 2014.

6 Hunter, Trevon J. "The Exploration and Analysis of Mancala from an AI Perspective." (2021).

7 Ramanujan, Raghuram, and Bart Selman. "Trade-offs in sampling-based adversarial planning." Twenty-First International Conference on Automated Planning and Scheduling. 2011.

8 Kaindl, Hermann. "Minimaxing: Theory and practice." AI Magazine 9.3 (1988): 69-69.

9 https://commons.wikimedia.org/wiki/File:Wooden_Mancala_board.jpg

10 http://kalaha.krus.dk/

11 McGovern, Amy, Zachery Tidwell, and Derek Rushing. "Teaching introductory artificial intelligence through java-based games." Second AAAI Symposium on Educational Advances in Artificial Intelligence. 2011.

12 Kaluderic. "On the Designing of a Text Protocol for the Game of Kalah."

Appendices

A Adversarial Search Sources

Interesting literature, including some other approaches: [3], [5], [6] and [7]

An incomplete list of short and easy-to-read articles from the chess programming community:

https://www.chessprogramming.org/Minimax
https://www.chessprogramming.org/Alpha-Beta
https://www.chessprogramming.org/Best-First_Minimax_Search
https://www.chessprogramming.org/MTD(f)
https://www.chessprogramming.org/Principal_Variation_Search
https://www.chessprogramming.org/ProbCut
https://www.chessprogramming.org/Late_Move_Reductions
https://www.chessprogramming.org/Null_Window
https://www.chessprogramming.org/Aspiration_Windows

https://www.chessprogramming.org/Monte-Carlo_Tree_Search
https://www.chessprogramming.org/UCT
https://www.chessprogramming.org/Jakob_Erdmann#UCT
https://www.chessprogramming.org/MC$\alpha\beta$