Design and Software Implementation of Heuristic and Suboptimal Strategies for the Mancala/Kalah Game

Libor Pekař, Jiří Andrla, and Jan Dolinay

Faculty of Applied Informatics, Tomas Bata University in Zlín,
Nad Stráněmi 4511, 76005 Zlín, Czech Republic
pekar@utb.cz

Abstract. One of the oldest games worldwide – the Mancala game – is focused in this preliminary study. Namely, its the most popular version – the Kalah game – is considered. This contribution is aimed at the analysis of Kalah rules first. Further, based on these rules, some novel deterministic and suboptimal strategies are proposed. It is proved that the order of playing has a decisive impact on winning. The proposed strategies have been implemented via a simple C++/Qt application. By experiments, a human player, when playing as the second one, cannot defend the designed strategies in general. However, the same applies in reverse – when a human player begins, he/she can nearly always win. To sum up, the proposed software-based strategies are comparable to human opponents.

Keywords: Game theory, C++, Kalah, Mancala, suboptimal strategy, implementation

1 Introduction

Mancala games represent a wide family of strategic board games with a variety of rules yet with common characteristics [1],[2]. They are included among the oldest board games worldwide ever [3]. Players need to have a board with one or more rows of holes (or pits, houses) in which a defined number of counters (or seeds) are placed. Often there are two or four additional holes (also called stores or end-zones) with a special meaning. The games are usually played by two players. The game starts with a certain distribution of the counters over the pits. A move consists of selecting one of the holes and putting all counters inside the hole one-by-one in adjacent holes in a certain direction. This procedure is called sowing. The goal is to capture as many counters as possible. The captured counters remain in the stores. According to the position and state of the last hole to which a counter is placed), some types of Mancala games continue in the move. Sometimes the player is allowed to do another move [4].

Kalah is a game from the Mancala family introduced in the 40s [5] or 50s [4] of the last century. It has become very high popularity in the Western world. The game has two players who own 6 holes on each side of the board. These sides are usually called South and North. In each hole, 2 to 6 counters can be distributed at the beginning of the

© Springer International Publishing Switzerland 2018
…
…

game. Moreover, the board is equipped with two stores for each side (player). Detailed game rules are given to the reader in Section 2 (along with their analysis).

In this preliminary study, we consider the game with 4 counters in each hole and we attempt to provide the reader with a combination of a heuristic and a weak solution of the game [6],[7], i.e., we suggest a behavior using which the Player 1 (= South) wins or draws from the initial (and every possible) position when playing first, given arbitrary play [8] on North side (= Player 2). On the other hand, if South starts as the second one, it may lose.

As mentioned above, Kalah rules and their analysis are given in section 2. Section 3 introduces selected existing software solutions and briefly describes the authors' implementation in C++ and Qt. The evaluation of the proposed solution can be found in section 4.

2 Kalah Analysis

2.1 Kalah Rules

Let us describe the most common rules for the Kalah game [4],[5]. The two players (North and South) sit at each side of the board (see **Fig. 1**). Each player has a row of 6 holes and one store on his/her right-hand side. It is assumed that South is Player 1, yet not playing as first in general.

- At the beginning of the game, a given number of counters are placed in each house (usually 4).
- Players take turns sowing their counters. A player removes all counters from one of the holes and, moving counter-clockwise, drops the counters one-by-one in each house in turn, including the player's own store but not the opponent's one.
- If the last sown counter lands into an empty hole owned by the player, and the opposite hole contains any counter, the sown and the opposite counter are captured and placed into the player's store.
- If the last sown counter lands into the player's store, the player do an additional move (this step can be repeated).
- If any player has no counter in any of his/her houses, the opponent moves all remaining counters to his/her store and the game ends.
- The player with the most counters in his/her store wins.

The usual Kalah game notation is *Kalah*(m, n) where m stands for the number of holes for each player and n means the number of counters inside every single hole at the beginning of the game. As introduced above, the most common case is *Kalah*(6, n) with $n = 4$; however, a different n is possible. In this contribution, we consider *Kalah*(6, 4). Note that the so-called empty capture rule is not assumed here. This rule means that if the player inserts the last counter into an empty hole and, simultaneously, the opposite hole is empty, the counter goes to the store (i.e., it does not remain in the hole).

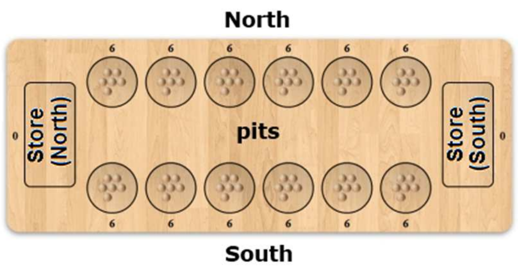

Fig. 1. The Kalah (6-counter) game board [9].

2.2 Rules Analysis

Various theoretical, as well as computer-aided analytic results on Kalah game complexity, have been obtained. Strong solutions based on a computer program that exhaustively compute all possible positions were obtained by Mark Rowlings in 2015 [8]. *Kalah*(6, 4) was proven win by 8, *Kalah*(6, 5) win by 10, and *Kalah*(6, 6) a win by 4 for the first player. By playing 100 games [7], it was shown that *Kalah*(6, 4) has an average game length of 30.75 with the average branching factor 4.08. However, full-depth game-tree searching strategies are time-consumptive. Many research results refer to the advantage for the first player. Some heuristic (experience-based) suggestions can also be found [10].

We do let provide the reader with a simple heuristic analysis that will be used for our suggested player's behavior combining heuristic and weak (low-depth game-tree search) strategies.

Ordinary (Unrepeated) Move. The goal is to save at least 25 counters. Counters can be saved only by sowing through the player's store. Usually, only one counter can be placed at the store; however, if the emptied hole includes more than 14 counters, two of them land to the store.

Repeated Move. Repeated moves mostly bring advantage for the player because of the increasing number of counters inside the store. Another positive impact is that the opponent's holes remain unaffected. It has been proved that the longest possible chain on such moves for the 6-hole board is 17 [5]. We have, however, found another (different from the cited source) chain of moves of the same length as follows:

- Let the holes be numbered consecutively from the left to the right (1 to 6). Assume that the numbers of counters inside them are 6, 5, …, 2, 1.
- The player starts the sowing by emptying hole 6. The only counter lands into the store.
- The player removes two counters from hole 2. Again, one counter is left in the store.
- As a third, hole 6 is to be emptied.
- Etc.

Such a scheme leads to 17 counters left inside the store and 4 counters remaining inside board holes within a single move. However, it is difficult to reach the initial scenario in practice. A good (even a partial) scenario is when there is the ascending (+1) number of counters inside holes 6, 5, 4, etc. starting from 1, on the board. Or, the same advantage appears when there is the descending (-1) scheme on the boar, starting from 6 counters inside hole 1. Then, the player should start the sowing from the right (i.e., from hole 6) or from the last possible hole when going through holes from the left.

Capturing the counters. This rule enables the player to gain more counters then when starting the game. Recall that the empty capture rule is not applied herein. On the other hand, the player should be aware of capturing his/her own counters as well.

2.3 Eventual Recommendations

Based on the preceding subsection, we can conclude the following rational suggestions (to be implemented in the software application):

- A repeated move is preferable. However, its advantage decreases with the distance of the particular hole from the store. If there are some empty holes near the store and there is a hole of a small number (i.e., on player's the left-hand side), the counters of which can ensure a repeated move, one has to be careful – If the player inserts a counter into the empty hole, he/she loses a chance to capture the opponent's counters.
- The player has to check a possible existence of empty opponent's holes, into which the last counter can land within the current move. This induces the necessity to analyze the opponent's move simultaneously. On the other hand, capturing represents the only possibility how to increase the number of player's counters.

3 Software Implementation

3.1 Existing Solutions

There can be found a lot of software solutions for the game on internet. Let us name just a few. Regarding mobile applications, the well-known Google Play service suggests AppOn Innovate Mancala [11] that enables to play the Mancala game in the on-line or the off-line modes and against a real or an artificial player. MobileFusion Apps Ltd Mancala Ultimate [12] has three modes – man-to-computer, man-to-man played on a single machine or via the internet. It enables to give the players names. Mancala by L. Priebe [13] has a stand-alone man-to-man or a man-to-machine mode. A short game statistics are displayed when the game is over. A very sophisticated and smart algorithm can be found at Mathplayground [14]. Its Kalah game can be played on-line only in the man-to-machine mode. Most of the above-referred implementations do not enable to determine the first player – which can, however, a crucial option [7], [8].

3.2 Used Tools

The well-known universal and object-oriented C++ language, one of the most prevelant programming languages worldwide [15], is used herein. Since one of the main goals is to design a graphical user interface (GUI) application, it is desirable to use an appropriate framework (i.e., a set of library modules and other programming tools) or a software development kit (SDK). Our software application of proposed Kalah strategies is designed in Qt [16], which is a complex SDK that includes all the necessary tools for application design and deployment. It is characterized by the usage of signals and slots. Roughly speaking, a signal is sent by an object (e.g., a widget) when some event appears, whereas a slot represents a function (e.g., of the C++ language) that returns a reaction to the signal.

3.3 Important Programmed Methods

Prior to a concise description of the proposed game strategy behavior and corresponding C++ methods (functions), we do let introduce key user-defined management (handling) functions.

`sowing` - This method performs a single move of the game for the given combination of a player and a hole. It returns *true* if the last counter lands to the store. If the opponent's counters should be captured, the `lostStones` function is called.

`lostStones` – It has the same parameters as sowing, i.e., the player number and the number of the hole. If the opposite hole is not empty as well, all the counters are placed into the store.

The application enables us to choose if Player 1 (a human being) or Player 2 (the computer) starts to play. The Player 1 move is handled by the `sowing` member function by using signal&slot tools.

3.4 Suggested Strategies

As introduced above, heuristic (deterministic) and weak (tree-search) strategies are proposed herein.

Deterministic strategy. Based on the game rules analysis and given recommendations (see subsections 2.2 and 2.3), a heuristic and deterministic player's behavior can be designed. This strategy simply repeats a finite set of methods with a given hierarchy based on the current game state. If the game move – as a result of the member function on a particular level – is evaluated as a preferable one, it is made. The code for the hierarchic order of conditions and methods inside method `PCplayer` is as follows:

```
bool mancala::PCplayer(unsigned short pl){
  if (whichLastStoneMancala(pl)){
    return true;
  }else if(stoneToEmptyHouse(pl)){
    return false;
  }else if (defense(pl)){
    return false;
  }else if(firstHouse(pl)){
  return false;
}
```

As can be seen, all the member functions inside `PCplayer` have the same argument *pl* that means the player number. Their concise description follows.

`whichLastStoneMancala` – This function checks whether it is possible to place the last counter from a hole into the store. In the positive case, it returns *true*.

`stoneToEmptyHouse` – It checks whether there exists a hole with the given number of counters inside, such that sowing ensures that some opponent's counters can be captured. In other words, an empty hole with a non-empty number of counters in the opposite hole must exist. Besides a simple move within the single row, a possible round-about move finishing in the starting or another empty move is tested.

`defense` – It works similarly to `stoneToEmptyHouse`, yet from the opponent's point of view. It means that a hole which is the most threatened by the capturing is sowed. However, such a move may lead to a loss of the player's chance to capture the opponent's counters during the next round.

`firstHouse` – This method search for the first non-empty hole (from the right).

Tree-search strategy. The second strategy implements a modified game-tree search for a given number *STOP_ITERATION* of layers. Note that the completed (full) tree search – that represents a brute force procedure – is not evaluated herein. Such a strategy or game solution is included in the family of weak solutions of a game [7],[8]. Apparently, the higher the value of *STOP_ITERATION* is, the slower the move calculation for Player 2 is. It is worth noting that the so-called game-tree pathology can appear [17]. It means that a deeper game-tree search may lead to worse play. Our strategy, however, combines a "slavish" searching with a heuristic behavior that prefers such a player's move that does not cause any loss for the player. The algorithm for this combined strategy and a single move can simply be expressed as follows.

```
Algorithm: Modified tree-search (current state of the game, player number,
                                 depth of the tree-search, moves made from
                                 the initial state)
  If the player has any counters for the state
    For all holes of the player
      If it is possible to do a strictly positive move
```

```
            Do the move and update the moves
            If the move can be repeated
                Update the state and Modified tree-search (state, player, depth,
                moves)
            Else
                Return the saved moves
        Else
            Return the saved moves
    If a strictly positive move has not been indicated for any of the holes (= moves is
    the empty set)
        Standard tree search (state, player, depth, moves)
        Return the saved moves
```

where

```
Standard tree search (state, player, depth, moves)
    If depth >= STOP_ITERATION or any of the players has no counters
        Evaluate the reached state and return the moves
    Else
        For all holes of the player
            Sow the hole, update the moves and the state for this branch, change the player,
            and increase the depth
            Standard tree search (state, player, depth, moves)
```

Let us now introduce some important steps of the algorithm in detail. The meaning of the strictly positive moves is that such moves can be done by the player without any fear of loss. Namely, this situation primarily appears when the move ensures a repeated move (i.e., it is similar to `whichLastStoneMancala`). However, a capturing of the opponent's counters is positive as well (see `stoneToEmptyHouse`); therefore, it is further tested whether the player can do such a move. If none of these two cases suggests itself, the standard tree search is performed. This function saves the computation time by avoiding the computation of less valuable moves.

The standard (raw) tree search implements the habitual iterative approach. Every single turn can be evaluated after moves are finished for both the players. The evaluation can be made when the maximum prescribed depth is reached or the game is over (i.e., in the leaf nodes). Sowing is naturally not possible for empty holes. A repeated move is checked by `sowing` function. If `Modified tree-search` is called for Player 1, the evaluation seeks the maximum number of stored counters for this player and the minimum of them for Player 2. Hence, the basic criterion for the move quality is the difference between the changes in positions of the stores for Player 1 and Player 2. However, a repeated move ought to be benefited as well. Moreover, if the hole nearest to the store is empty, this state implies an advantage for the next moves (due to a very high chance of a future repeated move). Hence, we have suggested the following cost function for Player 1 (p_1)

$$F_c\left(p_1\right) = a\left[\left(S_{1,end} - S_{1,start}\right) - \left(S_{2,end} - S_{2,start}\right)\right] + b + c \qquad (1)$$

where $S_{1,end}, S_{1,start}$ are the final and the initial number of stored counters for Player 1, respectively, and $S_{2,end}, S_{2,start}$ are those for Player 2. The indicated difference is multiplied by the scaling factor *a*. The value of *b* represents the number of repeated moves during the computed turn for Player 1. The value of *c* is non-zero if hole 6 is empty at the final state; otherwise, it is zero. The nominal values have been set to $a = 10,\ b = 1,\ c = 5$.

3.5 GUI

We do let concisely introduce the simple GUI programmed for verification of the designed algorithms. When starting the application, a window enabling the decision whether a human-being or the computer starts is released. This is very important feature that may significantly contribute to the game result. Then, the player can decide what strategy is to be used. There are 5 options here; namely, the deterministic algorithm or the modified tree-search strategy with the value of *STOP_ITERATION* from 1 to 4. After this selection, the main window (see **Fig. 2**). The human-being player (Player 1) is always situated on the Southside. When the game is over, a short summary of the game is displayed (who won, the eventual numbers of stored counters).

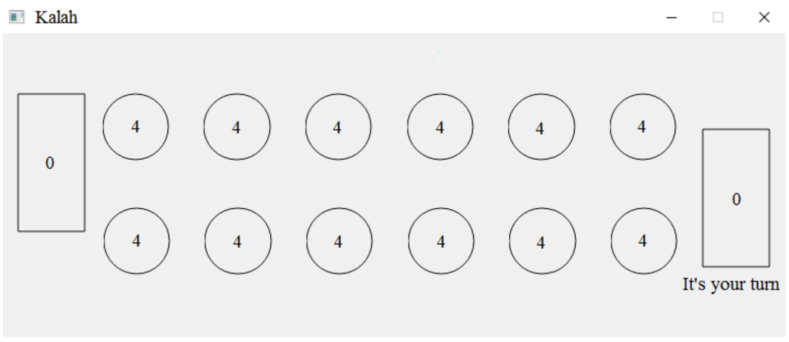

Fig. 2. The application GUI main window.

4 Evaluation

This subsection is aimed at a concise evaluation of both the proposed algorithms. Besides the cost function (1), the total running time (speed) of computer moves is taken as the quality measure.

All 5 above-mentioned options were evaluated by playing against an experienced human player. Let the human being be Player 1, whereas the computer be Player 2. The number of ten games have been played for every single strategy, except for the tree-search algorithm with *STOP_ITERATION* = 4 due to its slow computation speed. All the results are given to the reader in **Table 1**.

Table 1. Computer vs. Human Player Scores

Algorithm	Starting	Score	Number of the game									
			1	2	3	4	5	6	7	8	9	10
Deterministic	Player 1	Player 1	41	27	39	41	36	40	26	40	31	35
		Player 2	7	21	9	7	12	8	22	8	17	12
	Player 2	Player 1	19	13	19	19	19	15	20	16	23	16
		Player 2	29	35	29	29	29	33	28	32	25	32
Tree (depth 1)	Player 1	Player 1	35	34	22	32	20	29	35	22	16	35
		Player 2	13	14	26	16	28	19	13	26	32	13
	Player 2	Player 1	18	16	7	16	12	15	10	25	22	26
		Player 2	30	32	41	32	36	33	38	23	26	22
Tree (depth 2)	Player 1	Player 1	23	23	28	25	25	25	10	29	24	32
		Player 2	25	25	20	23	23	23	38	19	24	16
	Player 2	Player 1	17	18	12	13	23	19	13	15	7	13
		Player 2	31	30	36	35	25	29	35	33	41	34
Tree (depth 3)	Player 1	Player 1	35	26	27	28	21	20	27	26	21	26
		Player 2	12	22	21	20	27	28	21	22	27	22
	Player 2	Player 1	13	13	18	23	23	18	15	12	18	16
		Player 2	35	35	30	25	25	30	33	26	30	32
Tree (depth 4)	Player 1	Player 1	22	16	31	19	20	x	x	x	x	x
		Player 2	26	32	17	29	28	x	x	x	x	x
	Player 2	Player 1	5	9	6	5	5	x	x	x	x	x
		Player 2	40	39	41	35	33	x	x	x	x	x

As can be seen from the table, the starting position has a decisive impact to the game result. This fact has already been reported by some studies [5],[7],[8]. For the deterministic behavior strategy, a human opponent can always win when playing first. However, in some cases, he/she can win even if starting as second. As expected, the success of the tree-search strategy increases as the depth of the search increases. On the other hand, an increasing computation time can be observed. The computer managed to win when starting as second for a small number of games. To sum up, we can conclude that the proposed strategies are comparable to the behavior of a mid-experienced human player.

5 Conclusion

In this contribution, two computer game suboptimal strategies for the Kalah game have been proposed; namely, a deterministic and a modified tree-search one. The proposed algorithms have been verified by using a simple GUI application. It has been proofed by experiments that the programmed player behaviors give similar winning success to human-player movement decisions. However, the order of player is crucial.

Acknowledgments. We do have to express our gratitude to Dr. Petr Dostálek for his patience during preparing this contribution.

References

1. Murray, H.J.R.: A History of Board Games other than Chess. Oxford at the Clarendon Press, London (1952)
2. Russ, L.: The Complete Mancala Games Book. Marlow & Company, New York (2000)
3. Pankhurst, R.: Gäbäṭa. In: Uhlig, S. (Ed.) Encyclopaedia Aethiopica: D–Ha. Harrassowitz Verlag: Wiesbaden (2005)
4. Irving, G., Donkers, J., Uiterwijk, J.W.H.M.: Solving Kalah. ICGA J. **23**(3), 139-147 (2003)
5. Kalah. Wikipedia: A Free Encyclopedia. https://en.wikipedia.org/wiki/Kalah/ (Accessed October 24, 2019)
6. Allis, L.V.: Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis, Department of Computer Science, Rijksuniversiteit Limburg, Maastricht, The Netherlands (1994)
7. van den Herik, H.J., Uiterwijk, J.W.H.M., van Rijswijck, J.: Games solved: Now and in the future, Artif. Intel. **134**, 277–311 (2002)
8. Solved Game. Wikipedia: A Free Encyclopedia. https://en.m.wikipedi0.org/wiki/Solved_game/ (Accessed October 24, 2019)
9. Carstensen, A.K.: Solving (6,6)-Kalaha. http://kalaha.krus.dk/ (2011)
10. Brown, S.: Basic Strategy for Mancala. https://www.thesprucecrafts.com/ (Accessed October 29, 2019)
11. AppOn Innovate: Mancala [Mobile application software]. https://play.google.com/store/apps/details (Accessed November 1, 2019)
12. MobileFusion Apps Ltd.: Mancala Ultimate [Mobile application software]. http://mobilefusionapps.com/ (Accessed November 1, 2019)
13. Priebe, L.: Mancala [Computer software]. https://www.superhry.cz/games/240/ (Accessed November 1, 2019)
14. Math Playground LLC: Mancala [Java-based game]. https://www.mathplayground.com/mancala.html (Accessed November 1, 2019)
15. Stroustroup, B.: Programming: Principles and Practice Using C++. 2nd ed. Addison-Wesley Professional, Boston, MA (2014)
16. Eng, L.Z.: Qt5 C++ GUI Programming Cookbook: Design and Build a Functional, Appealing, and User-Friendly Graphical User Interface. Packt, Birmingham, UK (2016)
17. Zuckerman, I., Wilson, B., Nau, D.S.: Avoiding game-tree pathology in 2-player adversarial search. Comput. Intell. **34**(2), 542-561 (2018)