EE 562: Artificial Intelligence for Engineers Assignment 3

Cheng-Yen Yang
Department of Electrical and Computer Engineering
University of Washington
cycyang@uw.edu

1 Introduction of Kalah

According to Wikipedia[1], Kalah, is a game invented by William Julius Champion Jr. that can be dated back to 1940. Kalah is played on a board of two rows, each consisting six holes that can hold a number of seeds, and a kalah on each side. And the ultimate goal of the games is to capture more seeds via the following listed rules as two player take turns performing action[2].

Figure 1: A Kalah game board in real life.[2]

Generally speaking, each action is determined by a player's decision of choosing one of his or hers own hole and then distribute the seeds one by one counterclockwisely in all the holes and only the current player's kalah. And depending on the position of the last seed we will have different result[2]:

- If the last seed is dropped into an opponent's hole or a non-empty hole of the player, the move ends.
- If the last seed falls into the player's kalah, he or she moves again.
- If the last seed is put into an empty hole owned by the player, he or she captures all seeds of the opposite hole together and puts them into the kalah.

And the game ends when one of the player no longer has any seed in his or hers row of holes.

2 Heuristic Function

Different heuristic functions provide different insights and information of the current board of Kalah and prompt the AI to move accordingly. In this section we will show two distinct strategies of heuristic function and will record the experimental results in the next section.

2.1 Naive Method

For the naive method, our heuristic function is designed as:

$$h(s) = self_k - oppo_k \tag{1}$$

where s represents the current board layout, $self_k$ and $oppo_k$ respectively represent the number of seeds currently each player is holding in his or hers kalah (or notated as a_{fin} and b_{fin}). This heuristic function simply prompt the AI to selected the move that can created the largest difference in terms of the current scoring.

Even-though this naive method is already pretty hard to beat from a human perspective since the algorithm can search as deep as $\tilde{1}0$ further moves ahead. The strategy did not take the current situation of the holes into accountant which ignores a pretty important factor that it is the player who decides the action to make and therefore the numbers of seeds in the player's holes actually can do the player a favor. Therefore we present the advanced method base on the observation.

2.2 Advanced Method

For the advanced method, our heuristic function is designed up top of the naive method by adding another term as:

$$h(s) = \alpha \cdot (self_k - oppo_k) + (1 - \alpha) \cdot (\sum self_i - \sum oppo_i) \tag{2}$$

where s represents the current board layout, $self_k$ and $oppo_k$ respectively represent the number of seeds currently each player is holding in his or hers kalah (or notated as a_{fin} and b_{fin}) and $\sum self_i$ and $\sum oppo_i$ respectively represent the total number of seeds holding in each player's row of holes. The parameter α is set to 0.8 after a set of experiments.

3 Implementation

Follow the providing skeleton code of *ai.py*, I will break my code implementation into three part *move()*, *getSuccessors()* and *minmaxSearch()*.

3.1 *move()*

Starting with *move()*, this member function of the class *ai* is the interface API that the *main.py* call upon when it is the AI's turn to make a move. It will pass in the current state of the board in the format of a, b, a_fin and b_fin and returns a move(integer) accordingly. Here since we want to make the code as clear as possible, the move directly call for the *minmaxSearch()* function and only deal with the timing part as experiments require.

3.2 *getSuccessors()*

To perform minmax search algorithm with aplha-beta pruning, we need to first implement the search algorithm that can returns all the successor states. Following the implementation of *updateLocalState()* function in *main.py* we also create a similiar member function *getNextState()* in the class *ai* to help determine what will the state be after a certain move and will the state be valid or not.

3.3 *minmaxSearch()*

Finally we implement the minmax search algorithm along with alpha-beta pruning[3] in this function, *minmaxSearch()* function will get called recursively, and when the given depth is reached, we will determine the heuristic value based on the $h(s)$ member function. Also note that the input arguments for this function are current state, current α value, current β value and current depth. The Boolean *maxx* also help identify if it is a max node or min node.

4 Result

4.1 Search Time

The experiment result on the search time of the algorithms are collected by repeatedly playing the game for 20 times. From the table, we can easily observe that the average search time is exponentially proportional to the search depth as expected. As there is also a 1000ms limitation of the turnament, we also observe that it will sometime surpass such limitation for depth = 10 case.

Search Depth	Average Search Time (ms)
depth = 1	0.156 ± 0.0624
depth = 2	0.883 ± 0.452
depth = 3	1.925 ± 0.516
depth = 4	4.448 ± 1.235
depth = 5	$11{,}841 \pm 0.346$
depth = 6	29.814 ± 1.203
depth = 7	$76{,}217 \pm 1.345$
depth = 8	171.032 ± 4.032
depth = 9	445.087 ± 55.002
depth = 10	741.822 ± 72.245

Note: All experiment results are collected from running the program locally on my laptop (2.3 GHz Quad-Core Intel Core i7).

4.2 Win Rate

The experiment result on the win rate of the algorithms are collected by repeatedly playing the game against by human (myself) and our baseline algorithm (depth = 1 with naive method). We play 5 games for each combination and recording the win rate and wining difference of the algorithm.

Algorithm	Against Human	Against Baseline AI
Naive Method (d=3)	40% (+1.2)	40% (+3.0)
Naive Method (d=5)	80% (+4.4)	80% (+4.6)
Naive Method (d=10)	100% (+7.8)	100% (+6.0)
Advanced Method (d=3)	40% (+2.6)	80% (+3.8)
Advanced Method (d=5)	100% (+5.8)	100% (+5.0)
Advanced Method(d=10)	80% (+10.2)	100% (+7.2)

We experiment the naive method first so we think that the 'Human' had developed a better strategy during the matches and therefore somehow perform better during the games with the advanced method. Overall, we can observe that as the search depth goes deeper, the algorithm has a better chance of securing a win.

References

[1] Kalah - wikipedia. https://en.wikipedia.org/wiki/Kalah, accessed: 2020-10-25

[2] Kalah | mancala world. https://mancala.fandom.com/wiki/Kalah, accessed: 2010-09-30

[3] Alpha–beta pruning - wikipedia. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning, accessed: 2020-10-25