Implementing the *Best move or first robot* change case

Matthew Eden
Department of Electrical, Computer and Software Engineering
University of Auckland
Auckland, New Zealand
mede607@aucklanduni.ac.nz

***Abstract*—To evaluate the changeability of an implementation of Kalah (a.k.a Mancala), a change case was proposed. This involved replacing the second player in the game with a simple AI dubbed *Best move or first robot*. Instead of taking player inputs, this AI would determine moves based on whether it was possible to achieve an extra move or a capture. If neither were possible, it simply chose the first available house (i.e. first non-empty house). A change plan was created in anticipation of implementing the changes in code. During the execution of this plan, there were several issues encountered as a result of the complexity of the change case and the somewhat lacking changeability of the design requiring much new code to be written. The time taken to implement the change case was roughly 50 minutes. The overall impact of the change case was mild, as the base ruleset hasn't changed but the game is noticeably different.**

***Index Terms*—kalah, mancala, changeability, java**

I. Planning the Change Case

Before any code was actually written, a plan was devised describing the minimal amount of changes that would be required to achieve the desired change case. This plan is as follows.

1) Find the `GameConfig` class.
2) Change the value of `VERTICAL_OUTPUT` from `true` to `false`.
3) Find the `GameIO` class
4) Create a new method: `public static int computeMove(Board board, int playerNum, IO io)`.
5) Within this method, create a `List<Pit>` to contain the player's houses returned from `board.getPlayers().get(playerNum).getHouses()`.
6) Create three separate loops to iterate through each object in that list.
7) In the first loop, implement logic to determine if a move exists that leads to an extra move. Achieve this by comparing the number of seeds that would be sown and the distance to the player's store. If equal, that move leads to an extra move. Otherwise it does not.
8) In the second loop, implement logic to determine if a move exists that leads to a capture. Achieve this by calculating where the last seed sown will land and using this information to determine whether a capture occurs at that location.
9) In the third loop, implement logic to determine the legal move that exists with the lowest house number i.e. index.
10) Find the `Kalah` class.
11) Find the `Play` method.
12) Within the `while` loop, modify the assignment to the variable `selection` to read: `selection = (playerNum == 1) ? GameIO.computeMove(board, playerNum, io) : GameIO.getMove(board, playerNum, io);`.

II. Implementing the Change Case

The total amount of time taken to implement the change case was timed as being 48 minutes and 50 seconds. During the execution of the aforementioned plan, there were various issues encountered that arose due to the complexity of the change case. In particular, implementing the base logic for the BMF bot proved troublesome and the change plan did not convey in enough detail how this should be done; thereby requiring me as a developer to figure it out on the spot. Although there was a lot of new code to be written, a large majority of that code was constrained to a single method which made it easier to manage. A breakdown of each step in the change plan follows.

1) No issues in finding the `GameConfig` class.
2) No issues in changing the value of `VERTICAL_OUTPUT`.
3) No issues in finding the `GameIO` class.
4) No issues in creating the `computeMove` method.
5) No issues in creating a `List<Pit>` or invoking the required methods to obtain a value for it.
6) No issues in creating the loops.
7) In the first loop, some difficulties arose with implementing the comparison condition. Particularly with regard to moves that sow enough seeds to circle around the board, passing both player's houses.
8) In the second loop, it was difficult to calculate the position at which the last seed of a potential move would be sown. Particularly with regard to how the opposing player's store is skipped when sowing seeds that circle the board.
9) No issues in implementing the logic for the third loop.
10) No issues in finding the `Kalah` class.

11) No issues in finding the `Play` method.
12) No issues in modifying the assignment of `selection`.

III. Impact of the Change Case

On a scale from *0.0* to *1.0*, the impact of this change case is *0.3*. Although the rules of *Kalah* are preserved and the overall functionality of the game remains largely unchanged, the replacement of the second player with a simple AI changes the game from being multi-player to being single-player.

IV. Changeability Assessment

Implementing this change case required a large amount of new code to be written, and this exposed some gaps in the changeability of the design. This change case was implemented on top of a previous change case [2], and in that instance the design showed a good level of changeability. However, due to a lack of factorisation of functionality related to determining different move outcomes (i.e. gaining an extra move or executing a capture), this aspect of the design did not have good changeability and that made this change case more difficult to implement. If a future developer wishes to change the conditions that lead to an extra move or a capture they will need to significantly alter the design and doing so will not be a straightforward process, as I have discovered in implementing this change case. So, contrary to claims made previously [1], this design of Kalah only *partially* lends itself to changeability and could be improved to be more changeable.

References

[1] M. Eden "SOFTENG 701 - Assignment 3 - Submission", *Implementing Kalah with a Changeable Object-Oriented Design*

[2] M. Eden "SOFTENG 701 - Assignment 4 - Submission", *Implementing the Change Board Orientation change case*