Solitaire Mancala

Help Center

Overview

Mancala consists of a family of games in which the objective of a player is to move sets of objects around a board. One variant of Mancala that is played frequently in the United States is the game Kalah. (For the sake of simplicity, we will just refer to Kalah as "Mancala".) In this variant, the Mancala board consists of two rows of six pits called *houses* with one row facing each player and one larger pit at each end of the board called a *store*. The store on the player's right end of the board is considered to belong to the player.

At the start of a game, each house contains some number of *seeds* (usually three). On a player's turn, a player chooses one of the houses on his side of the board, removes the seeds from that house, and places one seed in each neighboring house or store in a counter clockwise direction. Since seeds cannot be removed from a store, all of the seeds will eventually accumulate in the two stores at the ends of the board. The player whose store contains the most seeds at the end of the game wins. To get a better feel for how Mancala works, try searching for "Mancala online" and you will find lots of sites that allow you to play online against a computer.

For the two player variant, there are several other rules designed to encourage more complex game play. However, our computing skills are not quite robust enough to tackle building a computer algorithm to play a reasonable game of two-player Mancala at this point. Instead, we will focus on a simpler one-player version of Mancala called Tchoukaillon. Since this name is rather unwieldy, we refer to this game as *Solitaire Mancala*.

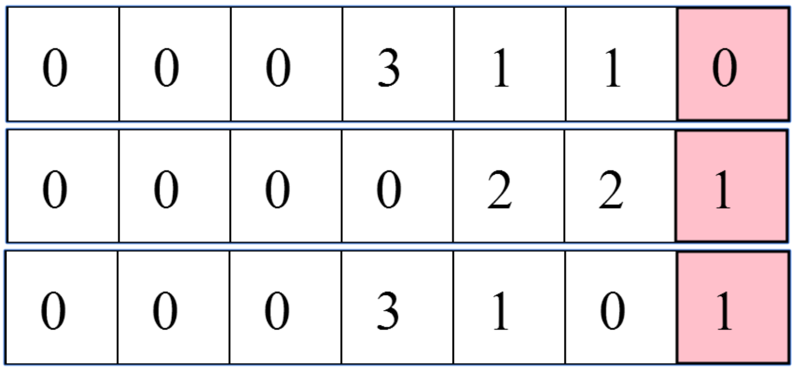
In Solitaire Mancala, the player has six houses and a store. At the start of a game, a variable number of seeds are placed in each house (as opposed to the three seeds per house in two-player Mancala). As in two-player Mancala, the player may select a house and gather all of the seeds in that house. The player then places the seeds one at time in the houses to the right of the selected house. In Solitaire Mancala, the last seed must be placed in the store. Other moves that either don't place a seed in store or end up with seeds left over are not allowed and are *illegal*. Note that the restriction that the last seed in a move must always be placed in the store substantially limits the choice of legal moves available to the player. If we index the store and houses from right to left starting at zero for the store, moving the seeds from a particular house is legal exactly when the number of seeds in the house matches the house's index.

Analyzing Solitaire Mancala

The player's objective in Solitaire Mancala is to choose a sequence of moves that transfers all of the player's seeds to the store. A *configuration* of the Mancala board corresponds to the current number of seeds in each house and the store. Due to the fact that most moves are illegal, a game of Solitaire Mancala can not be won from most starting configurations. However, some starting configurations are *winnable*. i.e; there is sequence of moves that transfers all of the seeds to the store.

However, even in winnable configurations, the player must choose carefully when faced with a choice of possible legal moves. Given the choice of two legal moves, the player should always choose the move whose house is closer to the store. Why? Note that if we choose the move for the house farther away from the store, the result of that move will cause the closer house to contain more seeds than its index. In this situation, there will then never again be a legal move available for this house.

For example, consider the top configuration shown below. The store (colored pink) contains no seed. To its left, house one contains one seed while house three contains three seeds. Moving the seeds from either of these houses is legal. However, the middle configuration show the results of moving the seeds from house three first. House one now contains two seeds. As a result, there will never be a legal move that can clear house one and, as a result, this middle configuration is not winnable. The bottom configuration show the configuration that results from moving the seeds in house one first. Note that this configuration is winnable since we can then move the seeds from house three.



0	0	0	3	1	1	0
0	0	0	0	2	2	1
0	0	0	3	1	0	1

Mini-project description

In this mini-project, we will implement a small Python program that models a game of Solitaire Mancala using an object-oriented approach. In particular, your task is to implement a `SolitaireMancala` class that models the current configuration of a game of Solitaire Mancala and provide several methods for analyzing and updating the game. You are welcome to start from this provided program template.

We suggest that you represent the configuration of the board as a list of integers with the first entry in the list being the current number of seeds in the store. In general, the *i*th entry of the list should represent the number of seeds in the *i*th house where the houses are indexed in ascending order from right to left. In the top configuration shown above, this list would be `[0, 1, 1, 3, 0, 0, 0]`.

Your `SolitaireMancala` class should support the following methods:

- `__init__(self)`: Create a `SolitaireMancala` object whose configuration consists of a board with an empty store and no houses (i.e; the configuration `[0]`).
- `set_board(self, configuration)`: Set the board to be a copy of the supplied configuration (to avoiding referencing issues). The configuration will be a list of integers in the form described above.
- `__str__(self)`: Return a string corresponding to the current configuration of the Mancala board. This string is formatted as a list with the store appearing in the rightmost (last) entry. Consecutive entries should be separated by a comma and blank (as done by Python when converting a list to a string).
- `get_num_seeds(self, house_num)`: Return the number of seeds in the house with index `house_num`. Note that house `0` corresponds to the store.
- `is_legal_move(self, house_num)`: Return `True` if moving the seeds from house `house_num` is legal. Otherwise, return `False`. If `house_num` is zero, `is_legal_move` should return `False`.
- `apply_move(self, house_num)`: Apply a legal move for house `house_num` to the board.
- `choose_move(self)`: Return the index for the legal move whose house is closest to the store. If no legal move is available, return `0`.
- `is_game_won(self)`: Return `True` if all houses contain no seeds. Return `False` otherwise.
- `plan_moves(self)`: Given a Mancala game, return a sequence (list) of legal moves based on the following heuristic: after each move, move the seeds in the house closest to the store when

given a choice of legal moves. Note that this method should not update the current configuration of the game.

Observe that the method `plan_moves` should return a sequence of moves that wins the game if the initial configuration is winnable. If the configuration is not winnable, `plan_moves` returns a sequence of moves that leaves the board in a configuration with no legal moves.

Testing your mini-project

One of the concepts that we will emphasize in this class is thoroughly testing your code. For Solitaire Mancala, we recommend a three-phased approach to testing.

1. Build your own tests that check correctness of your implementation of each method in the class. The provided template includes a short snippet of testing code at the end of the program. Observe that this code is encapsulated in a function `test_mancala` that is called after the testing code. Structuring your test code in this manner allows the tests to easily be disabled by commenting out the function call. (In week two, we will build test suites that allow for complete separation of the class definition and the test code.)

   As you implement each method in the `SolitaireMancala` class, you should add tests to `test_mancala` that checks whether each newly implemented method works as intended. While building such tests may seem laborious, the time spent debugging code built in the absence of such tests is often much more substantial.

2. Import and run this GUI code. In particular, add the following two statements to your implementation of the `SolitaireMancala` class.

   ```
   import poc_mancala_gui
   poc_mancala_gui.run_gui(SolitaireMancala())
   ```

   Run in CodeSkulptor, these two statements will import and run a GUI built with SimpleGUI that allows you to interact with your game and test it further. Note that this GUI is designed so that it only interacts with the `SolitaireMancala` object via the methods provided above.
3. Finally, submit your code (with the four lines of test code added above commented out) to the following OwlTest page. This page will automatically score your mini-project. Note that trying to debug your mini-project versus the tests in OwlTest will be very tedious since they are slow and give limited feedback. Instead, we *strongly* suggest that you test your program against your own tests and the provided GUI first. Programs that work correctly versus these tests are much more likely to run correctly against OwlTest.

Remember that OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will results in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums. If you get stuck or need a nudge forward, you are welcome to review our solution to this mini-project.

Created Tue 22 Apr 2014 11:36 AM PDT

Last Modified Sat 14 Feb 2015 7:45 AM PST