Problem Set 6

Due by 11:59 pm on Wednesday, November 6, 2019.

1 Assignment Goals

1. Extend the Kalah game from Problem Set 5 to allow for both human and machine players.
2. Learn how to declare and use polymorphic classes.
3. Learn how to use `const` to allow read-only access to the data in `Game` and related classes.
4. Experience issues that arise in refactoring code to accommodate new features.

2 Problem

In Problem Set 5, you implemented a Kalah game app that allowed two people to play a game of Kalah electronically, without the need for a physical board. The implementation used five types: `Player`, `Board`, `Pit`, `Game`, and `Referee`.

In this problem set, you will extend your implementation to allow for either or both players to be played by machine instead of by humans. To do this, you will implement the notion of an `agent`, which is the entity that controls a player (South or North) and chooses its moves. In PS5, the only agent is a human, and the code that interfaces with the human is embedded in the `Referee` class.

In this assignment, we have two different kinds of agents: human and machine. The human agent does the same thing as in PS5, but the code involved in interacting with the human to obtain the next move must be moved from the `Referee` class to a new `HumanAgent` class. The machine agent runs a not-very-smart AI algorithm to choose the next move. That code should reside in a new `MachineAgent` class.

The two agent classes should be derived from a common `Agent` base class that has a pure virtual function `int chooseMove(Player pl)`. The meaning is that the agent (human or machine) controlling player `pl` is to choose the next move when it is `pl`’s turn to move.

To invoke the proper agent, the referee will have separate `Agent*` pointers, one for `South` and one for `North`. The referee sets them to point to the agents controlling `South` and `North`, respectively, when it creates the two agents.

To know which kinds of agents to create, we add two more arguments to the command line. These arguments are strings describing the two agents playing the game. The first string describes the agent controlling `South`; the second describes the agent controlling `North`. The two legal string arguments are `human` and `machine`.

When it is player `pl`’s turn to play, the referee will find the `Agent*` pointer for player `pl`. Call it `ap`. It will then call `ap->chooseMove(pl)`. Because `Agent::chooseMove(Player)` is virtual, this will automatically dispatch the call to the appropriate `chooseMove()` function.

For this assignment, the machine should always choose the smallest-numbered valid move. For example, if the machine is playing `North`, then the agent would choose the first non-empty house
in the sequence N1, N2, ... If all its houses are empty, then the game is over and the machine should not be asked to choose a next move.

The referee must be modified in several ways. First, it should create the agents for playing South and North. Second, the code for obtaining the next move from a human should be moved into the new HumanAgent class, as described above. Third, the referee should call the polymorphic function chooseMove() as described above.

For the machine to move, it will need to query the game to find a non-empty pit. More intelligent future AI's might want to take much more information into account such as how many seeds are in each of the pits, how many houses are there, and so forth.

To give the agents access to such information, you should do two things. First, have the referee create the agents after creating the Game object so that it can pass a const Game reference to the selected agent constructor. The agent can then store the const Game reference in a data member which its chooseMove() function can access. By making the reference a const, this preclude an agent from cheating by changing the board or game progress.

Secondly, you should add information functions to Game for supplying the information that the agents might need. For example, you might add a function bool Game::isEmpty(Player pl, int num) that tests whether pl's pit num is empty or not. The class that has the knowledge to answer this question is Board, so Game::isEmpty() can simply delegate the task to a corresponding isEmpty() function in Board.

The information functions you add to Game will all need to be const functions in order to be accessible using the const Game reference provided to the agents. Getting const in the right places to avoid compiler errors can be tricky. Please ask for help if you are having problems with this.

You may discover other changes and additions to the existing PS5 code are needed in order to make this all work properly. As usual, such design decisions should be documented in notes.txt.

3 Testing

You should test your program to make sure that it follows the specification above. You should repeat the tests from PS4 and PS5 to make sure that in modifying your code you haven’t broken any of the parts that used to work. In addition, you should consider carefully what an adequate test of the new parts should be, and you should implement test code to verify that the code passes them. For example, you should test the four cases of the program with human vs. human, human vs. machine, machine vs. human, and machine vs. machine agent combinations. You should also verify one way or another that the machine agent is following the simple-minded algorithm described above.

4 Grading Rubric

Your assignment will be graded according to the scale given in Figure 1.
<table>
<thead>
<tr>
<th>#</th>
<th>Pts.</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>All relevant standards from previous problem sets are followed regarding good coding style, submission, identification of authorship on all files, and so forth.</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>All of the instructions in section 2 regarding code structure and functionality are followed.</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>You should perform all required tests from PS4 and PS5 that are still relevant to your code and supply the output that shows those tests are still passed.</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Running <code>make</code> successfully compiles and links the project and results in an executable file <code>kalah</code> that takes four command line arguments.</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td><code>kalah</code> produces correct output when run with the four different combinations of agents controlling <code>South</code> and <code>North</code>.</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td><code>valgrind</code> shows no memory leaks and no memory management errors.</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>A written file <code>notes.txt</code> explains briefly what each function in each class does, and why it belongs in its class. No need to duplicate functions from PS5 that have not changed. Any additions or modifications to the classes from PS5 are described and justified. Other observations about the chosen code structure also go here.</td>
</tr>
</tbody>
</table>

| 20 | Total points.                                                                                               |

Figure 1: Grading rubric.