Comp 424 - Project Specification

Course Instructor: Joelle Pineau (jpineau@cs.mcgill.ca)
Project TA: Eric Crawford (eric.crawford@mail.mcgill.ca)

Code due: March 30, 2015
Report due: April 1, 2015

Goal

The main goal of the project for this course is to give you a chance to play around with some of the AI algorithms discussed in class, in the context of a fun, large-scale problem. This year we will be working on a game called Omweso. Eric Crawford is the TA in charge of the project and should be the first contact about any bugs in the provided code. General questions should be posted in the project section in mycourses.

Rules

Omweso is a two-player competitive game that is part of the *mancala* family of board games, which generally involve placing seeds in pits, and stealing or capturing opponents' seeds. In Omweso, each player controls 16 pits arranged in a 2x8 grid. To start the game, each player strategically arranges their 32 initial seeds in their controlled pits. Once both players have arranged their seeds, the players take turns. An Omweso turn proceeds as follows. The player selects one of the pits they control that contains more than 1 seed, and scoops all the seeds in that pit. The player then begins *sowing* the scooped seeds, moving counter clockwise from the scooped pit, placing one seed in each pit until running out of seeds. Note that players only ever place seeds in their own pits. At the end of a sowing, 3 things can happen depending on the contents of the pit in which the last seed was placed:

1. If the pit was previously empty, then the player's turn is over.
2. If the pit was occupied, and it in the player's second row of pits, and both of the opponent's pits directly opposite are occupied, then a *capture* takes place. The player scoops the seeds from both of the opponent's pits, and begins sowing those seeds from the same pit where the previous sowing began from.
3. If the pit was occupied but the conditions for capture are not met, then the player scoops the seeds from the ending pit, and begins sowing again from there. This is called relay sowing.

The player's turn continues until a sowing ends in an empty pit (i.e., option 1 happens). Note that neither player may capture on the first turn, and if a situation arises where a player would have captured, relay sowing is performed instead. A player wins when it is their opponent's turn to play, but their opponent has no more valid moves (so all their pits contain either 0 or 1 seeds). A draw is declared if neither player has won after 5000 turns. Complete rules can be found here: http://en.wikipedia.org/wiki/Omweso. Note however, that we make some modifications to the standard rules:

- No reverse capture.
- No alternate victory conditions. The only way to win is to leave your opponent without a legal move when it is their turn to play.

- Simultaneous setup: both players will place their initial seeds *without knowledge of how their opponent is setting up*. However, the players *will* know whether they will sow first or second, allowing them to adapt their initial configuration accordingly.

Implementation

Competition Constraints

We will hold a competition between all the programs submitted by students in the class, with every submitted program playing one match against every other program. Each match will consist of 2 omweso games, giving both programs the opportunity to play first.

During the competition, your player will be given no more than 2 seconds per normal move and no more than 60 seconds to choose its initial seed configuration. This initial 60 second period should also be used to set up your agent (e.g. loading data from files). If your player does not choose a move within the allotted time, a random move will be chosen instead. If your agent exceeds these time limits drastically (for example, if it gets stuck in an infinite loop) then you will suffer an automatic loss. Your agent will run in its own process and will not be allowed to exceed 500 mb of RAM. The code submission should not be more than 10 mb in size. Going over these memory limits will result in an automatic loss/disqualification.

To deal with the possibility of infinite moves, we will simulate a given move for at most 200 iterations (where an iteration is defined as scooping a pit and sowing its seeds). If a move is played which has not ended by 200 iterations, the game will be cancelled and restarted. If a player is responsible for 3 such cancellations in a single game, they will suffer an automatic loss.

You are free to implement any method you wish as long as your program runs within the constraints and is well documented in both the write-up and the code. Documentation is an important part of software development, so we expect well-commented code. All implementation must be your own. You are not allowed to use non-standard libraries. If you are unsure if a particular library is allowed, feel free to ask on mycourses where the project TA will clarify.

Submission Constraints

We provide code for the board game logic as well as an interface for running and testing your player. You are only required to extend the "boardgame.Player" class, overriding the chooseMove method. An example submission is provided. We will enforce the following naming/structure constraints (replacing XXXXXXXXX with your student number):

1. The player class you want us to test must be named "sXXXXXXXXXPlayer".
2. The player class you want us to test must have a default constructor which calls super("XXXXXXXXX"), **and does nothing else**. In particular, code for setting-up your agent should be in the chooseMove method, not in the constructor.
3. The player class you want us to test must have the createBoard method copied from the example submission.
4. All your classes must be in the same parent package which must be named "sXXXXXXXXX". Sub-packages are allowed (e.g., "sXXXXXXXXX.mytools").
5. The player class you want us to test must be in the parent package "sXXXXXXXXX".
6. You must submit your source code with the package structure intact.
7. Do not include the provided code.

You are free to reuse any provided code in your code (must be documented). We plan on running several thousand games and cannot afford to change any of your submissions. Any deviations from these requirements will have you disqualified, resulting in part marks.

You are expected to submit working code to receive a passing grade. If your code does not compile or throws an exception, it will not be considered working code, so be sure to test it thoroughly before submitting. If your code runs on the trottier machines with the (unaltered) provided code, your code will run without issues in the competition. The provided code includes a README file on how to start a game. We will allow programming languages other than java, but we wont support them. It will be up to you to write the socket based message passing to communicate with the server. You will have to provide a shell script to launch your player as well as a make file. It will need to compile/run on the trottier machines. If you choose not to use java, **please let the project TA know** and include a detailed README file explaining how to run your shell script. For non-java languages, which libraries are allowed will be determined on a case-by-case basis in order to avoid unfair advantages.

Write-up

You are required to write a report with a detailed explanation of your approach and reasoning. The report must be a typed .pdf file, and should be free of spelling and grammar errors. The suggested length is between 4 and 5 pages (at ∼300 words per page), but the most important constraint is that the report be clear and concise. The report must include the following required components:

1. An explanation of how your program works, and a motivation for your approach.
2. A brief description of the theoretical basis of the approach (about a half-page in most cases); references to the text of other documents, such as the textbook, are appropriate but not absolutely necessary. If you use algorithms from other sources, briefly describe the algorithm and be sure to cite your source.
3. A summary of the advantages and disadvantages of your approach, expected failure modes, or weaknesses of your program.
4. If you tried other approaches during the course of the project, summarize them briefly and discuss how they compared to your final approach.
5. A brief description (max. half page) of how you would go about improving your player (e.g. by introducing other AI techniques, changing internal representation etc.).

Academic Integrity

This is an individual project. The exchange of ideas regarding the game is encouraged, but sharing of code and reports is forbidden and will be treated as cheating. We will be using document and code comparison tools to verify that the submitted materials are the work of the author only. Please see the syllabus and www.mcgill.ca/integrity for more information.