Java PahTum: Know How

Karol Górnicki

September 21, 2012

This document provides short introduction of how to use Java PahTum and addresses few basic questions in order to help in further development of the application.

1 Structure of The Application

The broad overview of how the classes are organized and how they co-operate with each other is given in my dissertation. The names are pretty intuitive so there is no need to read it, though.

1.1 AI Package

Not all classes that you can find out there were used in the end in the project. At one point or another I made use of each of them. However, some were followed by new implementation or different concept–nonetheless, they have not been deleted. Superfluous packages are: minimax, mchboltzmann, random (AI which plays moves at random).

1.2 Core Package

Some methods in the Board class are denoted as *deprecated*. They are not used in any agent, only weren't deleted.

2 Images in GUI

In order to display images properly (stones and *black holes*) adjust their paths (only in GUI class). The images are stored in img folder.

3 How The Game Is Organized?

In order to find out how one single game is organized go to the *GUI* class in the *ui* package. Line 151 starts a single game. The while loop checks whether number of moves made is smaller than number of all empty fields at the beginning of the

game. In order to find out who's turn it is now the players are organized in an array and we track the *currentIndex* variable which is adjusted after the move is finished ($currentIndex = (currentIndex+1)\%2$;). Number of conducted moves is also incremented by 1. Player object holds information regarding the type of the AI, which we check at line 153, 180 and so on. At this point we focus only on one case (lines 153-179), all the other are copy-paste with changed names of the AIs. At first we create a new object of an AI engine. As a parameter we give

- **Board**: At this level making a deep-copy isn't necessary, it's just a good practice to not allow other objects to manipulate with the board object. Let them have a deep-copy of it with which they can do whatever they please, without any worries that some changes performed on the board, even unintentional, can affect the game.
- **Color** : Each player is associated with a color. For the sake of the game, it really doesn't matter whether black or white kicks off the game.
- **Number of move**: How many moves have been performed till now.
- **Total number of moves**: Number of all empty fields in the initial state. This information is needed for knowing when the simulation finishes (it's just quicker then checking whether there are no empty fields left).

Line 165. We get the move (each player holds the information regarding number of simulation–in terms of AI like *Charles*, *Random* or human player this data is muted, you can enter whatever you want, it will never be used).

Line 169. The obtained move is performed on the board and then on the board that is presented to the GUI. Remember that the move object is a tuple (the same notion as in Python). Next counter of the moves and currentIndex are adjusted. When the loop is finished (the only condition when the game finishes is no empty fields left) the score is calculated, lines 262-269, and the information of it pops out.

4 How The Tests Are Organized?

For the purpose of this project the tests were organized in multiple matches collection. In a nutshell, we want to compare 2 AIs in a match of 100 games. The test is organized as a 50 sets (set is 2 games played on the same board where in the second game the agents switch the sides, so both play as a player1 and player 2–this reduces the advantage of the first player, e.i. m#1: AI_1 vs. AI_2, m#2: AI_2 vs. AI_1). For each set we use different board, but keep in mind that all boards feature the same number of *black holes* (by the way, in the application they are called *dead fields*). To find out how the test is organized go to GigaTest1 class in the newtest package.

Lines 30-46. Those variables are used to track the statistics of the experiment, how many times agent #1 won the game when was playing the first move, and so on.

Lines 48-238. The games are played. It's pretty well covered by the comments so another explanation would be redundant.

Lines 244-288. Updating the file of the statistics. The file contains 6 tests, so if you only want to run one delete/comment the others.

5 How The Agents Are Organized

5.1 MCTS UCT

Go to *Monte Carlo* class, *ai.montecarlo* package. It's well commented and should be easy to figure out what's the purpose of each piece of code.

5.2 MCTS Beam Search

Go to *MonteCarloH5* class, *ai.montecarloheuristic5* package. It's well commented and should be easy to figure out what's the purpose of each piece of code. All the others beam search implementations differ in parameters. The core structure remains unchanged.

5.3 MCTS Boltzmann (Gibbs)

Go to *MonteCarloH5Boltzmann* class, *ai.mch5boltzmann* package. It's well commented and should be easy to figure out what's the purpose of each piece of code. All the others Boltzmann implementations differ in parameters. The core structure remains unchanged.

5.4 Charles

Package *ai.charles_2* contains the Charles engine.

6 Commentary of The Code

Classes mentioned in Section 5 are commented. Their siblings (e.g. different implementations of beam search) are also commented but the comments might be not always as comprehensive as in classes from Section 5.

7 Contact details

if you have any further questions feel free to contact the author or supervisor of this project.

Author:
Karol Górnicki
email: karol.gornicki (at) gmail.com
email: kg687 (at) york.ac.uk
Supervisor:

Daniel Kudenko
email: kudenko (at) cs.york.ac.uk