SCHOOL OF COMPUTER SCIENCE
COURSEWORK ASSESSMENT PROFORMA

MODULE & LECTURER: CM3112, Steven Schockaert

DATE SET: 17 October 2016 (Monday week 4)

SUBMISSION DATE: 7 November 2016 (Monday week 7)

SUBMISSION ARRANGEMENTS: submit to Learning Central a zip file containing (i) the source code of your implementation and (ii) a short report.

TITLE: Artificial Intelligence Coursework

This coursework is worth 30% of the total marks available for this module. The penalty for late or non-submission is an award of zero marks. You are reminded of the need to comply with Cardiff University's Student Guide to Academic Integrity. Your work should be submitted using the official Coursework Submission Cover sheet.

INSTRUCTIONS

You are required to implement a computer player for the board game Quoridor based on random playouts, as described in more detail in the attachment. The implementation should be in Java using the framework that has been provided. Your entire source code should be included in the zip file you submit. Your source code should compile using

```
javac */*.java
```

Your computer player should run using

```
java quoridor/Quoridor
```

SUBMISSION INSTRUCTIONS

Description		Type	Name
Cover sheet	**Compulsory**	One PDF (.pdf) file	[student number]-cover.pdf
Solution	**Compulsory**	One ZIP (.zip) archive	[student number]-solution.zip

CRITERIA FOR ASSESSMENT

Credit will be awarded against the following criteria.

For the implementation:

- [Correctness] Does the code correctly implement the required algorithm?
- [Efficiency] Are the most appropriate data structures used?
  Is the computational complexity of the implementation optimal?
- [Clarity] Is the code clearly structured and easy to understand?
  Are comments provided where necessary to clarify non-standard code fragments?

For the report:

- [Correctness] Does the answer demonstrate a solid understanding of the problem?
- [Clarity] Is the answer well structured and clearly presented?
  Are suitable tables, diagrams or graphs used where needed?

FURTHER DETAILS

Feedback on your coursework will address the above criteria and will be returned in approximately 3 weeks.

This will be supplemented with oral feedback in the lectures.

Getting started

To get started with this coursework, first download `quoridor.zip` from Learning Central. In this archive, you will find a basic implementation of a slightly simplified version of the board game Quoridor. Quoridor is a turn-based strategy game which is played on a $k \times k$ grid. The original version uses a 9×9 grid and can be played with two or four players. For this coursework, we will only consider the two-player version of the game, and we will use a reduced board of size 5×5 to keep the size of the search space manageable.

A screenshot of the initial game state is shown in Figure 1(a). The aim for each player is to move their pawn to the opposite side of the board, i.e. the game is over when the red pawn is on the top row (in which case the red player has won) or when the green pawn is on the bottom row (in which case the green player has won). In each turn, a player can either move their pawn one position on the board (horizontally or vertically, but not diagonally) or place a wall segment on the board. Walls segments are two grid cells long and can be placed horizontally between two rows or vertically between two columns of the grid. Figure 1(b) shows a screenshot of the board after a wall segment has been added. The following restrictions apply[1]:

- Pawns cannot jump over walls, i.e. there cannot be a wall segment between the current position of the pawn and the position the pawn is being moved to.
- The number of wall segments each player can place on the board is restricted to 5[2].
- Wall segments cannot be placed on the board if they would intersect with a previously placed wall.
- It is illegal to place a wall segment that would cut off a player from all goal positions (e.g. a wall segment placed by the green player cannot cut off the red player from the top row).

In the default configuration of the framework, the red player is the human player and the green player is the computer player. The red player makes the first move. The controls for the human player are as follows:

- To move the player, simply use the arrow keys. This action is taken immediately and cannot be undone.
- To place a wall segment, first press 'w' and then choose the location of the wall segment using the arrow keys. At this point, the wall segment will be shown in grey (see Figures 1(c) and 1(d)). The wall segment can be rotated by pressing 'r'. To confirm the final location of the wall segment, press enter, after which its color will change (provided that it is in a legal position). Alternatively, pressing 'w' again will remove the wall segment and allow you to move the player instead.

Game framework

The main classes of the java framework are:

[1] In the original version of the game, two pawns cannot occupy the same grid cell at the same time. We will not consider this restriction.

[2] In the original version of the game, which uses the larger 9×9 board, players start with 10 wall segments each.

Quoridor This class represents an instance of the game, and will call the `chooseMove` method of the two players in turn, until the game is over. This class also contains the main method.

QuoridorPlayer An abstract class which contains some code that is common to both `HumanPlayer` and `ComputerPlayer`. The method `chooseMove` is left abstract.

HumanPlayer An extension of `QuoridorPlayer` which allows a human to play the game, using the controls outlined above.

AlphaBetaPlayerFixed An extension of `QuoridorPlayer` which implements a basic minimax player with $\alpha - \beta$ pruning, using a fixed ply.

AlphaBetaPlayerIterative An extension of `QuoridorPlayer` which implements a basic minimax player with $\alpha - \beta$ pruning and iterative deepening.

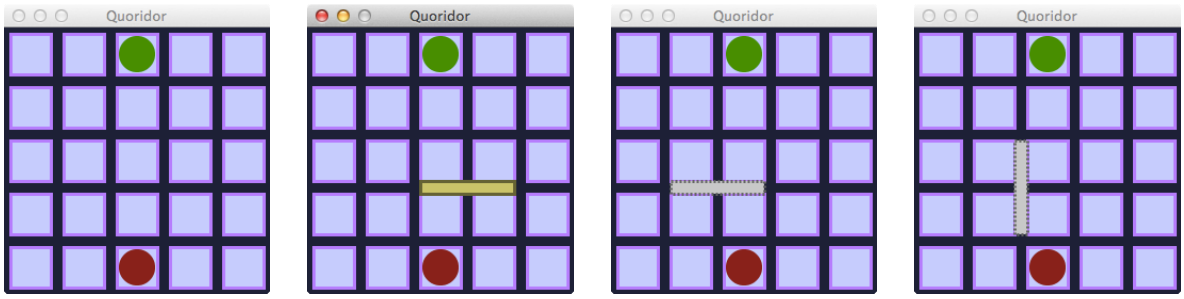
GameState2P This class represents a state of the game. Among others, it contains the code to compute which moves are legal as well as the code to heuristically evaluate the utility of a state. Analysing the wall structure requires the use of breadth-first search to check whether any of the players is cut off from their goal. As this is computationally demanding, a memoization technique is used to avoid repeating the same computation over and over again.

GameDisplay This class takes care of the visualisation of the game.

In addition to the provided framework, you are allowed to reuse any code from the lab classes. However, you are not allowed to use any other existing code or libraries. If you use any external sources for solving your coursework (e.g. pseudocode from a website), you should add a comment with a reference to these sources and a brief explanation of how they have been used. You are allowed to discuss high-level aspects of your solution with other students, but you should disclose the names of these students in a comment.

Assignment

The objective of this coursework is to implement a computer player based on a simplified form of Monte Carlo tree search. The different variants that you need to implement are all based on



(a) Initial state (b) Fixed wall position (c) Choosing a wall position (d) Rotating the wall

Figure 1: Screenshots of the Quoridor framework used for the coursework.

running simulations of the game, which are called playouts in this context. The basic principle, in each case, is to run a large number of simulations, and to decide based on the outcome of these simulations what is the most promising move to play.

Task 1: Random playouts (25 marks)

Create a class `RandomSimulationPlayer`, implementing a computer player that selects the next move based on random simulations of the game. In each random simulation, it is assumed that both players repeatedly choose the next move to play by (i) determining the set of legal moves, and (ii) choosing at random one of these moves (assigning equal probability to each legal move), until the simulated game finishes. The computer player repeatedly runs such simulations for 5 seconds, storing for each of the possible initial moves (i) how often it was chosen as the first move and (ii) in how many cases this led to a win. Finally, the computer player selects the move that led to a win in the highest percentage of cases.

Task 2: Heuristic playouts (25 marks)

Create a class `HeuristicSimulationPlayer`, which differs from `RandomSimulationPlayer` in how the simulations are run. Rather than making a move purely at random, it is now assumed that each player plays the 'heuristically best' move with 90% probability and a purely random (legal) move otherwise. To determine what is the heuristically best move, you should use the move that would be played by `AlphaBetaPlayerFixed` when using a very shallow depth (e.g. 1 or 2).

Task 3: UCB1 selection (20 marks)

Create a class `UCB1SimulationPlayer`, which differs from `HeuristicSimulationPlayer` in how the initial move is chosen. One important limitation of the players in Tasks 1 and 2 is that they waste a lot of computation time on moves that are clearly not promising. In practice, it is usually beneficial to focus on the initial moves that seem most promising, where we can use earlier simulations to estimate which are the most promising moves. For this player, the initial move is chosen as follows. As long as there is still some initial move that has not been chosen in any simulation, run a simulation with such a previously unexplored move as the initial move. Then, after each initial move has been chosen at least once, choose the initial move as the one which maximises the following UCB1 score:

$$\mathit{UCB1} = \frac{w}{n} + \sqrt{\frac{2 \ln t}{n}}$$

where n is the number of previous simulations that have started with that move, w is the number of these previous simulations that have resulted in a win, t is the total number of previous simulations, and $\ln$ is the natural logarithm. This score can be theoretically justified as an optimal balance between exploration (i.e. making sure you consider all the possibilities) and exploitation (i.e. focusing on the most promising moves). After the initial move has been chosen based on the UCB1 score, the player should use the same simulation as `HeuristicSimulationPlayer`.

Task 4: Recursive UCB1 selection (10 marks)

The idea that most of the computation time should be spent on the most promising moves does not only pertain to the initial move. We can similarly argue that in most of the simulations, we should

assume that the opponent plays the move which seems most promising (given the previous simulations), and so on. For this task, you should create a class `RecursiveUCB1SimulationPlayer`, where up to some fixed depth d (e.g. $d = 2$), each move is chosen as the one maximising the UCB1 score. In particular, suppose we have a node in the game tree s with children $s_1, ..., s_k$. Let t be the total number of simulations that have involved node s, let n_i be the total number of times the move to s_i was chosen as the next move (i.e. $t = n_1 + ... + n_k$), and let w_i be the number of times this led to a win. If the simulation ends up in node s, and s is not a final state of the game, then as the next move, we choose the one that ends up in node s_i such that the following score is maximised:

$$UCB1 = \frac{w_i}{n_i} + \sqrt{\frac{2 \ln t}{n_i}}$$

After choosing the first d moves in this way, we run the heuristic simulation from Task 2.

Task 5: Documentation (20 marks)

You should include a one page PDF containing the following

- A brief discussion about the effectiveness and limitations of the considered techniques. Overall, would you recommend a player based on rollouts or a minimax based player for this kind of game? How could the effectiveness of the computer players be further improved?
- A table with some experimental results to support your main conclusions (e.g. showing how often one type of player wins against the other types). There is no need for an exhaustive experimental analysis of all the different variants. Think carefully about what is important to include in the table to support your main conclusions.

Note that you will be marked down if the report is longer than one page.