Haskell Project 2: Gomoku

CS2006, 2017-18

Due date: Tuesday 17th April, 21:00
33.3% of Overall Mark for the Module

Overview

The objective of this project is to implement a board game in Haskell. By completing this project, you should further enhance your Haskell programming skills and get more detailed insights into Haskell's applications and available libraries.

The Game

The game chosen for this practical is **Gomoku**, also called **Five In A Row**. It is traditionally played on a Go board (a 19×19 grid, such as we have in the coffee areas!) with Go stones. It is a two player game, where each player takes it in turns to place a piece on the board. Black moves first, no piece is ever moved or removed from the board, and the winner is the first player to form a line of exactly 5 in a row either straight or diagonally (i.e. 6 in a row or more does not win!)

There are several variations of the game, and additional "house" rules can be introduced:

- It can be played on a smaller board, with smaller rows required to win.
- There can be rules to limit where stones are played, e.g.
  - The rule of **three and three**: neither player is allowed to have two open rows (i.e. not blocked at either end by their opponent) of three in a row.
  - The rule of **four and four**: neither player is allowed to have two rows (open or otherwise) of four in a row.
  - To increase fairness and reduce the advantage of moving first, the above rules may only be applied to black.

Preparation

Some code is given as a starting point. You can download the code from the `Practicals/H2/Code` directory on Studres. There is an initial `cabal` setup, including a file `Gomoku.cabal`, the `cabal` generated `Setup.hs` and a source directory `src` containing:

- `Main.hs`, which contains the main program initialising the graphics and setting up event handlers
- `Draw.hs`, which contains a function for drawing the board
- `Input.hs`, which contains a function for dealing with input events (mouse clicks and key presses)
- `Board.hs`, which contains an initial representation of the board and other game state
- `AI.hs`, which includes an outline of a minimax search

You are free to edit these files or introduce new source files as you see fit.

In order to compile this, you will need to be working in Linux. You will first need to install the `gloss` library, which is available from hackage, using the command `cabal install gloss`. Then, to build the project:

- `cabal configure` in the `Code/` directory to configure
- `cabal build` in the `Code/` directory to build an executable, which you will find in `dist/build/gomoku/`
- As before, you can edit and test individual functions at the ghci REPL, which you can start with `cabal repl`. (However, unfortunately `gloss` does not work in ghci so you will need to compile in order to test your final program.)

Basic Requirements

The minimum requirement of this project is to implement the game of Gomoku, such that a human player can play against a computer AI opponent. The Gomoku variant you should use initially should have the following characteristics:

- A 6×6 board with a target of 3 in a row (this is primarily to simplify the AI requirements — you may want to use the full board and 5 in a row instead)
- No restrictions on where either player can play (except that a move cannot be played where there is already a piece)

To achieve this, you will need to:

1. Implement the game mechanics in `Board.hs`. The board is represented as a list of tuples of positions and colour; you will need to write functions to add moves to the board (checking if a move is legal) and to check whether there is n in a row in any direction
2. Implement the `drawWorld` function in `Draw.hs` to display the current board state graphically
3. Implement appropriate *event handlers* for input events such as clicking on the board; these will need to identify which board position a mouse location maps to, in particular
4. Implement a move generator (in `AI.hs`) and an evaluation function (in `Board.hs`) to provide a computer opponent

A basic implementation of these four items, with correct implementation of the game rules, an accurate graphical display, and a reasonable method for choosing the computer's next move, would achieve a grade of 13. In order to achieve a grade higher than 13, you should implement some of the additional requirements below.

Additional Requirements

NB: It is strongly recommended that you ensure you have completed the Basic Requirements and have something to submit before you attempt these Additional Requirements!

In order to achieve a grade higher than 13, you should implement some of the following requirements (at least the **Easy** requirements); for a grade higher than 17, you should implement all of the **Easy** requirements and at least three of the **Medium** requirements (but note that this doesn't *guarantee* a higher grade! You still need to implement them well, and describe them clearly in a report.)

- **Easy** Add command line options to set up game options (e.g. to set which player is human and which is the computer, to set the size of board, etc). (Hint: look up `getArgs`)
- **Easy** Implement Undo, to roll back the game state to after the previous move
- **Easy** Implement some rule variants such as the rules of **three and three** or **four and four**
- **Medium** Allow options to be set in-game as well as at the command line (Hint: the `World` state could also describe options, and represent whether the game is currently running, then `handleInput` can be extended to change the options when the game is not running.)

- **Medium** Use bitmap images to display the board and pieces rather than simple gloss `Pictures`
- **Medium** Add the possibility of displaying hints
- **Medium** Implement a save game feature, and reload
- **Medium to Hard** Implement time limits for moves and "Pause".
- **Medium to Hard** Investigate and implement other variants such as **Renju** and **Pente** (Pente, in particular, is different in that it allows pieces to be *captured*)
- **Hard** Look up the SGF file format, and allow recording and replaying of games
- **Hard** Implement multiple AIs with different skill levels and different strategies, and allow the player to choose
- **Hard** Allow network play

Deliverables

Hand in via MMS, by the deadline of 9pm on Tuesday of Week 11 (17th April 2017), a single `.zip` or `.tar.gz` file containing three top level subdirectories as follows:

- A `Code`, containing your group's code, and should be the same for everyone in the group. There may be further subdirectories if you wish, containing the source code in well-commented `.hs` files.

  Everything that is needed to run your application should be in that directory or part of the Haskell standard library — there should be no dependencies on external libraries. I should be able to run your program from the command line of a bash shell.

- A `.hg` directory containing your mercurial repository.
  - Alternatively, as long as *everyone* in the group agrees, and you say so in your report, you may submit a git repository

- A `Report` directory, containing a group report of around 1500 words, and an individual report of 100–500 words. The *group* report should include:
  - A summary of the functionality of your program indicating the level of completeness with respect to the Basic Requirements, and any Additional Requirements.
  - Any known problems with your code, where you do not meet the Basic Requirements or any Additional Requirements you attempted.
  - Any specific problems you encountered which you were able to solve, and how you solved them.
  - An accurate summary of provenance, i.e. stating which files or code fragments were
    1. written by you
    2. modified by you from the source files provided for this assignment
    3. sourced from elsewhere and who wrote them

  The *individual* report should describe your own contribution to the group work, and summarise how you worked together as a team.

  Both reports should be submitted as PDF.

Marking Guidelines

This practical will be marked according to the guidelines at `https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html`
To give an idea of how the guidelines will be applied to this project:

- The **most simple solution** which implements the basic game mechanics supporting two human players using the computer to take turns, accompanied by a report, should get you a grade 10.
- Completing the **basic requirements** with a well-commented/documented, fully-working solution, with accurate graphics and using a reasonable algorithm to choose the computer's move, accompanied by a clear and informative report, should get you a grade 13. The report should address all the Basic Requirements, and should make clear which of these you have completed.
- To achieve a **grade higher than 17** you will need to implement the **Easy** and at least three of the **Medium** additional requirements listed above, with well-commented and documented code, accompanied by a clear and informative report.

Useful Resources

- `gloss` documentation
  - `http://hackage.haskell.org/package/gloss`
- Gomoku rules (and suggested variants)
  - `http://www.yucata.de/en/Rules/Gomoku`
  - `https://en.wikipedia.org/wiki/Gomoku`
- Online games
  - `http://www.mindoku.com/`

Policies and Guidelines

Marking

See the standard mark descriptors in the School Student Handbook: `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html#Mark_Descriptors`

Lateness penalty

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof): `http://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

Good academic practice

The University policy on Good Academic Practice applies: `https://www.st-andrews.ac.uk/students/rules/academicpractice/`

Finally

Don't forget to enjoy yourselves and use the opportunity to experiment and learn! If you have any questions or problems please let me, a demonstrator, or your tutor know — don't suffer in silence!