INF2D · Reasoning and Agents

Coursework 1 - Quoridor in Haskell

Ramon Fernández Mir, Claudia Chirita

Deadline: 3pm, Tuesday 9th March 2021

Practical details

This assignment is about problem solving by searching. You will implement an AI that plays Quoridor using Haskell.

The assignment is marked out of 100, and it is worth 15% of your overall grade for Inf2D.
It consists of four parts:

- Part I: The minimax algorithm (60 pts)
- Part II: Alpha-beta pruning (10 pts)
- Part III: AI helps solve Quoridor mystery (10 pts)
- Part IV: Extensions (20 pts)

The files you are going to use can be downloaded from the Inf2D Learn website. Go to Assessment → Assignment Submission → Coursework 1 - Search and Games. You should download the file `coursework1.tar.gz` which contains the code template you will use to write your solution.

Submission

Create a directory in which you keep the files you submit for this assignment. This directory should be called `Inf2d-ass1-s<matric>` where `<matric>` is your matriculation number (e.g 1234567). It should include all the files we provide, along any other files you create.

Copy the contents of the file `coursework1.tar.gz` in your directory. Then write your answers for Part I and II in the file `Players/Minimax.hs` that we provided, and the answer for Part III in `Players/Reed.hs` (the code) and in `Players/Reed.txt` (the answer written in natural language).

Submit your assignment by compressing your directory into a file `Inf2d-ass1-s<matric>.tar.gz`. You can do this using the following command in a DICE machine:

```
tar -cvzf Inf2d-ass1-s<matric>.tar.gz Inf2d-ass1-s<matric>}
```

You can check that this file stores the intended data with the following command, which lists all the files obtained by extracting the original directory (and its files) from this file:

```
tar -t Inf2d-ass1-s<matric>.tar.gz}
```

Submit this file via Learn by uploading the file using the interface on the Learn website for the course Inf2D. If you have trouble please refer to this blogpost: `https://blogs.ed.ac.uk/ilts/2019/09/27/assignment-hand-ins-for-learn-guidance-for-students/`.

The deadline for submission is **3pm, Tuesday 9th March 2021**.

You can submit more than once up until the submission deadline. All submissions are timestamped automatically. We will mark the latest submission that comes in before the deadline.

If you submit anything before the deadline, you may not resubmit afterward. (This policy allows us to begin marking submissions immediately after the deadline, without having to worry that some may need to be re-marked).

If you do not submit anything before the deadline, you may submit *exactly once* after the deadline, and a late penalty will be applied to this submission, unless you have received an approved extension. Please be aware that late submissions may receive lower priority for marking, and marks may not be returned within the same timeframe as for on-time submissions. For additional information about late penalties and extension requests, see: `http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests`.

Do not email any course staff directly about extension requests; you must follow the instructions on the web page.

Good Scholarly Practice: Please remember the University requirement as regards all assessed work for credit. Details about this can be found at :

- `http://www.ed.ac.uk/schools-departments/academic-services/students/undergraduate/discipline/academic-misconduct`, and
- `http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`.

Furthermore, you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put any such work on a public repository then you must set access permissions appropriately (generally permitting access only to yourself, or your group in the case of group practicals).

1 What is Quoridor? How do you play?

Quoridor is a 2 or 4-player board game. The board is a 9x9 grid and each player starts in one of the sides and is assigned a number of "walls" (there are 20 walls in total and they are split evenly between the players). The goal is to be the first player to reach the opposite side. In each turn, a player can:

- **Move a step** in any direction but not diagonally, provided the cell is free and not blocked by a wall.
- **Place a wall** in a 2x2 group of cells. Walls block the path of all players. They can only be placed if they don't overlap with any other walls and the player has walls remaining. Once placed, they can't be removed.

See the picture below for a more visual example. Also take a look at the Wikipedia page[1] for a detailed explanation and more examples.

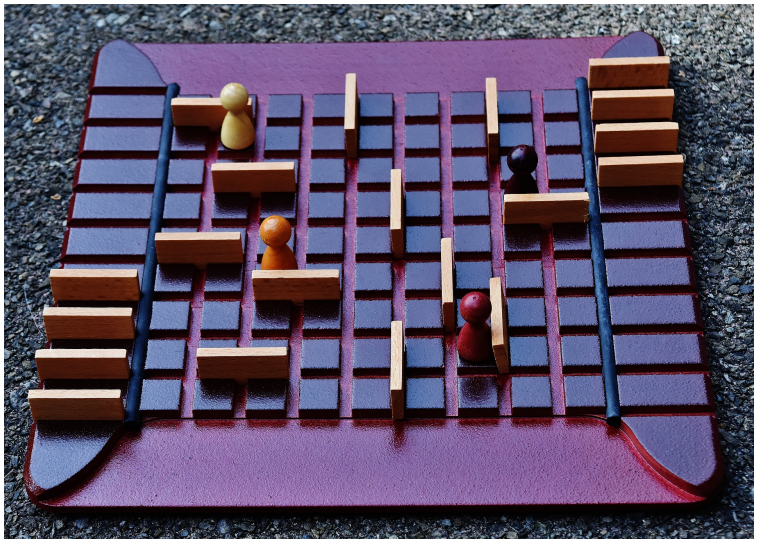

Figure 1: Example of a real Quoridor game.

Notation

We will use the official Quoridor notation[2] throughout. We look at the board from the point of view of the starting player, who starts at the middle column of the bottom row. From left to right, columns are labelled from 'a' to 'i' and, from the bottom to the top, rows are labelled from '1' to '9'. Hence the starting player starts in cell 'e1'.

Actions also use this notation:

- Moving a step is recorded as the destination cell. For instance, the starting player's first action could be 'd1', which would mean they've moved to the left.

[1] `https://en.wikipedia.org/wiki/Quoridor`
[2] `https://quoridorstrats.wordpress.com/notation/`

- Placing a wall is recorded as a cell plus 'h' (horizontal) or 'v' (vertical). The wall is placed in between the cell and its top-right neighbour, in the specified direction. Examples:
  - 'e3v' corresponds to the wall between columns 'e' and 'f' spanning columns '3' and '4'.
  - 'e3h' corresponds to the wall between rows '3' and '4' spaning rows 'e' and 'f'.

2 Quoridor in Haskell

In this section we will describe our implementation of Quoridor in Haskell which you will use for your assignment.

Differences with the real game

There are some simplifications with respect to the real game:

- Our board is 5x5. This makes it easier to test the game and reduces the memory comsumption.
- Since the board is smaller, we only allow 10 walls in total.
- We only allow two players.

We represent the board as a graph, which is a quite convenient data structure to work with. Cells correspond to vertices and possible paths are represented by edges. Placing a wall is equivalent to removing two edges.

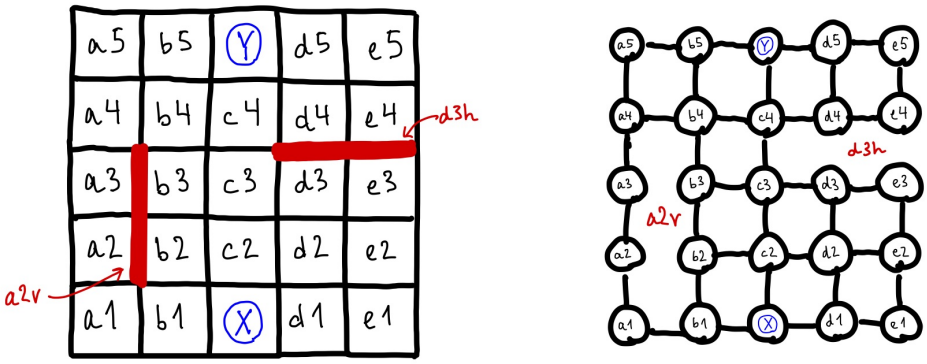

Figure 2: Usual representation (left) and our representation (right).

Installation instructions

In order to be able to play the game and run the tests, you need to make sure that you have:

- the Glasgow Haskell Compiler (GHC),
- the Cabal build system, and
- the Stack tool.

We suggest following the instructions on the Haskell website[3].

[3] `https://www.haskell.org/platform/`

Playing the game

Once everything is installed, you can play the game from the command line. You need to call the `main` function defined in `Main.hs`. Then you will be asked to select the player types (explained later) and the game will begin. You should see the following:

```
ramon@Ramons-MacBook-Air haskell-quoridor % cd src
ramon@Ramons-MacBook-Air src % ghci Main
...
Ok, 13 modules loaded.
*Main> main
What kind of player is player X? (Human/Dumb/Minimax/Reed)
Human
What kind of player is player Y? (Human/Dumb/Minimax/Reed)
Dumb

Player X's turn.

  a5-- b5-- Y -- d5-- e5
  |    |    |    |    |
  |    |    |    |    |
  a4-- b4-- c4-- d4-- e4
  |    |    |    |    |
  |    |    |    |    |
  a3-- b3-- c3-- d3-- e3
  |    |    |    |    |
  |    |    |    |    |
  a2-- b2-- c2-- d2-- e2
  |    |    |    |    |
  |    |    |    |    |
  a1-- b1-- X -- d1-- e1
```

For players of type `Human`, the program waits on an input. Following the example above, we could type `c2` to make `X` move to the cell above or `b4h` to place a wall under `Y`.

You can exit the game using `Ctrl + c` and exit the interactive GHC console with `Ctr + d`.

Running the tests

The project includes two test suites:

- **Basic tests**. These test the basic functionality of the files that you don't need to modify (until perhaps Part IV). You can run them as follows:

```
ramon@Ramons-MacBook-Air src % stack test :basic-tests
haskell-quoridor> test (suite: basic-tests)
...

Finished in 0.0794 seconds
```

```
97 examples, 0 failures

haskell-quoridor> Test suite basic-tests passed
```

- **Minimax tests**. You are meant to use them as you start implementing Part I (a-d). If they pass, it indicates that your solution for those parts is quite likely to be correct. You should run the command `stack test :minimax-tests`:

```
ramon@Ramons-MacBook-Air src % stack test :minimax-tests
haskell-quoridor> test (suite: minimax-tests)
...

Finished in 0.1300 seconds
23 examples, 0 failures

haskell-quoridor> Test suite minimax-tests passed
```

Project structure

In this section we explain how the project is structured and what you can expect to find in each file. Inside the `src` directory we find:

- `Types.hs`. In this file you will find all the main data types and type synonyms used throughout the project. The definitions of `Cell`, `Action`, `Board`, `Player` and `Game` are found here.
- `Constants.hs`. Here we define the size of the board, number of players, number of walls and other related values.
- `Cell.hs`. We define operations on cells such as computing the cells around a given cell.
- `Action.hs`. Given a cell, we define functions that return steps and walls in all directions.
- `Board.hs`. Here we find functions to check if a step and a wall are valid in a given board. We also define what it means to place a wall.
- `Player.hs`. We define functions to check the position of the player and whether it can move. There are functions to move the player and keep track of used walls too.
- `Game.hs`. Provides an interface to check whether an action is valid in a game, (`validStepAction` and `validWallAction`) and to perform an action: `performAction`.
- `Main.hs`. Here you will find the main game loop.
- `Print.hs`. Here you find the function to translate a game to a string in order to display it in the command line.
- `Players/Human.hs`. We define a function that translates a command to an action.
- `Players/Dumb.hs`. We define a player that moves at random.
- `Players/Minimax.hs`. Here you should write your solutions for Part I and Part II.

- `Players/Reed.hs` (and `Reed.txt`). Here you should write your solution for Part III.

In the `test` directory you will find the two test suites described earlier. You don't need to understand how they work although we recommend taking a look at the minimax tests as they provide useful information about what your code should do.

3 What you need to do

Part I: The minimax algorithm (60 pts)

The basic *minimax* algorithm[4] is used to choose the next action in a 2-player game. At a given game state, one considers the *game tree* of all the possible future game states. We need a *utility function* that assigns a value to each game; a higher value implies a higher chance of winning. The idea is that we traverse the game tree maximising the score at the levels of the tree that correspond to our player and minimising the score at the levels that correspond to our opponent.

Consider a game where there are only two possible actions (a and b). The minimax algorithm would perform the following calculations and decide that b is the best action:

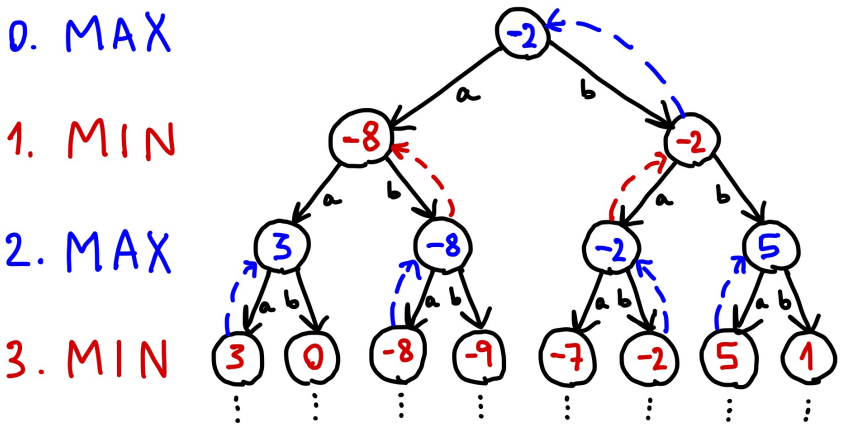

Figure 3: Simple minimax example.

The first issue that we need to address is that we don't want to consider infinite game trees, which are certainly possible in Quoridor. We want to look ahead a fixed number of steps into the future, so we will need a function to *prune the depth* of our tree.

Moreover, in our game, there are way more than 2 possible actions at each turn. This means that we will have a large number of branches at each level. That is a problem as the size of the tree will grow exponentially with respect to the branches per node. Therefore, it would be nice to have a function to *prune the breadth* of our tree. Note that, before, we will need to *order the tree* so that the branches that we cut off are the least meaningful.

You will get to implement a version of the minimax algorithm with all of these features. The skeleton file can be found at `Players/Minimax.hs`.

[4]See Chapter 5 in *"Artificial Intelligence: A Modern Approach"* by Russell and Novig.

Part I.a: Generating a game tree (10 pts)

First, we will generate a game tree from a game. Each node in the game tree holds a game and a list of branches where each branch is a tuple consisting of an action and another node. The action associated to each branch is meant to be the one used to get to the next game state.

`generateGameTree :: Game → GameTree`

You probably want to use the functions `validActions` and `performAction`.

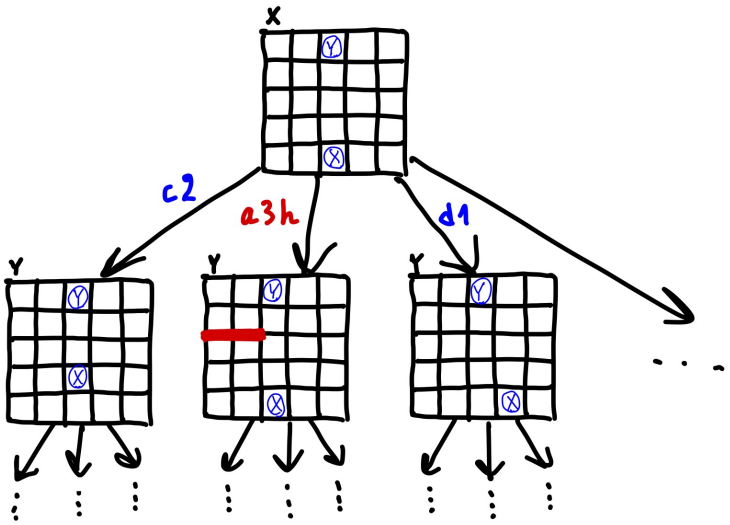

Figure 4: Game tree example.

Part I.b: Ordering a tree (5 pts)

Next, we will define two functions that order a tree in an alternating way. If at one level, the nodes with the highest value go first, in the next level, the nodes with the lowest value should go first, and so on.

`highFirst, lowFirst :: (Ord v) ⇒ StateTree v a → StateTree v a`

The definition should be mutually recursive.

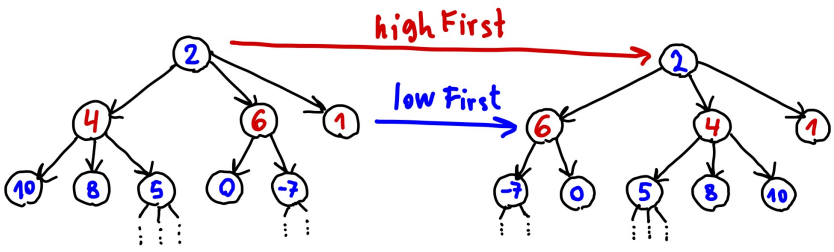

Figure 5: Ordered tree using `highFirst` and `lowFirst`.

Part I.c: Depth pruning (5 pts)

As discussed at the beginning of the section, we want to limit how many steps ahead the algorithm looks. This boils down to reducing the depth of a tree. Given a depth and a tree, you should return the same tree but without the branches at a level lower than the first input value.

```
pruneDepth :: Int → StateTree v a → StateTree v a
```

You should consider different cases on the input value and write a recursive definition.

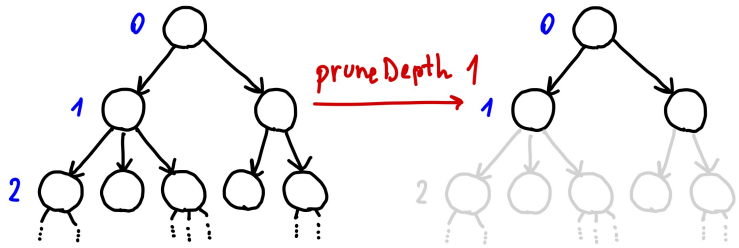

Figure 6: Example of applying `pruneDepth 1`.

Part I.d: Breadth pruning (5 pts)

Similarly, we want to be able to limit the breadth (or width) of a tree.

```
pruneBreadth :: Int → StateTree v a → StateTree v a
```

Consider using the function `take`, defined in Haskell's prelude.

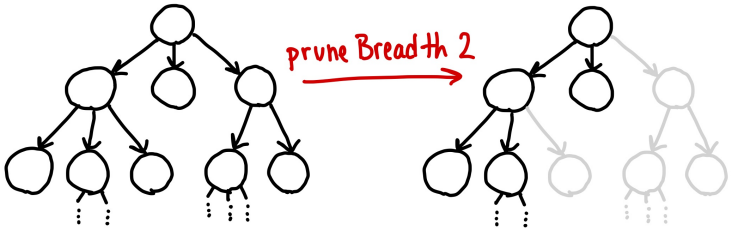

Figure 7: Example of applying `pruneBreadth 2`.

Part I.e: Defining a utility function (15 pts)

Next, you should define a utility function that assigns a score to a game. Note that given a game, we know whose turn it is (for instance using `currentPlayer` on the players list).

```
utility :: Game → Int
```

There are many ways in which one could write this utility function. A naive approach would be to give 1 to winning positions and 0 otherwise. However, for this approach to work we would need, in general, to look many steps into the future, making the algorithm very slow. Instead, you should

somehow calculate the distance between the player's current cell and its winning positions. If you're able to do so, it is straightforward to define the utility function as the closer we are to our winning positions, the more likely we are to win the game. For instance, you can use `reachableCells`.

Once we have a utility function we define the following function that transforms a game tree into an evaluation tree:

```
evalTree :: GameTree → EvalTree
evalTree = mapStateTree utility
```

Part I.f: Running minimax on an evaluation tree (20 pts)

The last ingredient that we need is, unsurprisingly, an implementation of minimax. It should traverse an evaluation tree (tree with scores on the nodes and actions on the branches) and choose an action.

```
minimaxFromTree :: EvalTree → Action
```

You will probably need a helper function to keep track of sequences of actions as well as your calculations. There are many ways of doing it. For instance, you could use the `Result` data type and have a helper function with the following signature:

```
minimaxFromTree' :: [Action] → EvalTree → Result
```

Part II: Alpha-beta pruning (10 pts)

Alpha-beta pruning[5] is a strategy that reduces the number of nodes checked by the minimax algorithm. The idea is that while traversing the tree, we keep track of two quantities: α and β. They are used to decide when to stop checking the branches. We ask you to rewrite minimax but incorporating this optimisation.

```
minimaxABFromTree :: EvalTree → Action
```

Again, you will need a helper function. One possibility would be:

```
minimaxABFromTree' :: [Action] → Result → Result → EvalTree → Result
```

Note that to test it, you have to change the `minimax` function.

Part III: AI helps solve Quoridor mystery (10 pts)

One might not expect a game like Quoridor to incite spirits and start internet wars, but rather to gather around a small but tight community of players. However, the Quoridor players community is in flames. The reason: openings.

[5] See section 5.2.3 in *"Artificial Intelligence: A Modern Approach"* by Russell and Novig.

You might know what a chess opening is and how important they are for the game. Quoridor openings are similar to the chess ones. But while there are many books on chess openings, the Quoridor community has at disposal just a couple of documented openings.[6]. One of these openings is the subject of a long lasting dispute in the community: the Reed opening[7].

Your task for this part of the assignment is to help the community to decide once and for all if the Reed opening is a good opening or not. Your answers should be written in the files `Players/Reed.hs` and `Players/Reed.txt`.

In this section, we need to assume that the board has the official size so go to `Constants.hs` and set `boardSize = 9`.

The **Reed opening** is rather simple: `c3h f3h`. This creates a narrow space in the middle on your side of the board, which should, in principle, make it harder for your opponent to reach your side. But does it?...

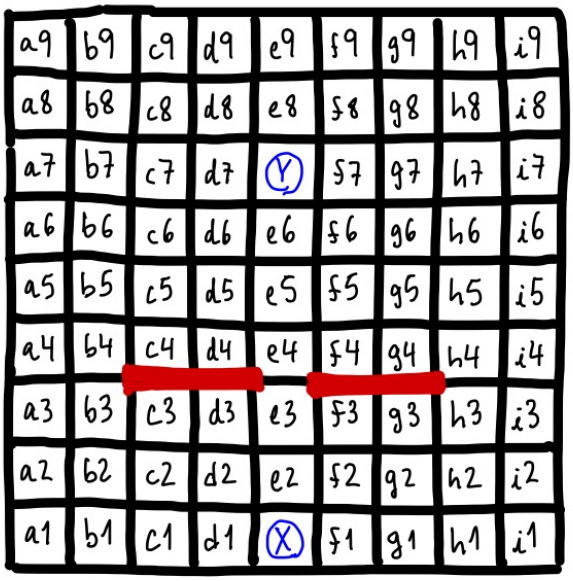

Figure 8: Player X applies the Reed opening.

In this part, we ask you to:

1. Describe in plain text how you would test if the opening is good or not using the code that you have already written for Parts I and II (or perhaps using one of your extensions from Part IV).
2. Create a player that always uses the Reed opening.
3. Fight the player according to the plan you described.
4. Write about your findings. Is it a good opening? Save your answer in the file `Players/Reed.txt`.

[6] `https://en.wikipedia.org/wiki/Quoridor#Strategies`
[7] `https://quoridorstrats.wordpress.com/2014/10/15/openings-the-reed/`

Part IV: Extensions (20 pts)

Finally, this last part is completely open-ended. You are free to extend the game in any way you want (as long as the tests still pass!). Here we list a few suggestions that you may want to consider:

- Change the game mechanics to allow more actions (e.g. diagonal moves, jumping over other players, etc.).
- Allow 4-player games. A lot of it is already in place to allow 4-player games. More interestingly, how would you make minimax work with 4 players?
- Improve the user interface. You can modify the `Print.hs` file to change how the board looks. Maybe you could use a Haskell graphics library.
- Improve game experience (e.g. better error handling, letting the user select the difficulty level or the name of the player, etc.).
- Extend and optimise minimax. And you could also use any other algorithm that you like to write another AI. Then they could play against each other.
- Set up a platform to play against your other coursemates (challenging but would be cool).
- ...