Informatics 2D Coursework 1: Search and Games

Xue Li, Thomas Wright, Stefano Albrecht, Cristina Alexandru

February 2, 2018

1 Introduction

The objective of this assignment is to help you understand the various search algorithms and their applications in path finding problems. You will implement the search algorithms using *Haskell*.

You should download the following file from the Inf2D web page:

http://www.inf.ed.ac.uk/teaching/courses/inf2d/coursework/Inf2dAssignment1.tar

Use the command **tar -xf Inf2dAssignment1.tar** to extract the files contained within. This will create a directory named *Inf2dAssignment1* containing *Inf2d1.hs* and four other supporting files for the assignment.

You will submit a version of the *Inf2d1.hs* file containing your implemented functions. Remember to add your **matriculation number** to the top of the file. Submission details can be found in Section 7.

The deadline for the assignment is:
3pm, Tuesday 13th March 2018

Some basic commands of Haskell are listed below[1]. An example of using these commands are given in Figure 1, in which commands are highlighted.

- `ghci` GHCi is the interactive environment, in which Haskell expressions can be interactively evaluated and programs can be interpreted. You can start GHCi with the command `ghci` in a terminal. All following commands work in GHCi environment.
- `:help` Get help information.
- `:cd <dir>` You can save *.hs files anywhere you like, but if you save it somewhere other than the current directory, then you will need to change to the right directory, the directory (or folder) in which you saved *.hs.
- `:show paths` Show the current directory.
- `:load Main` To load a Haskell source file into GHCi. For short, you can also use `:l Main`, where Main is the topmost module in our assignment.
- `main` Run function main.

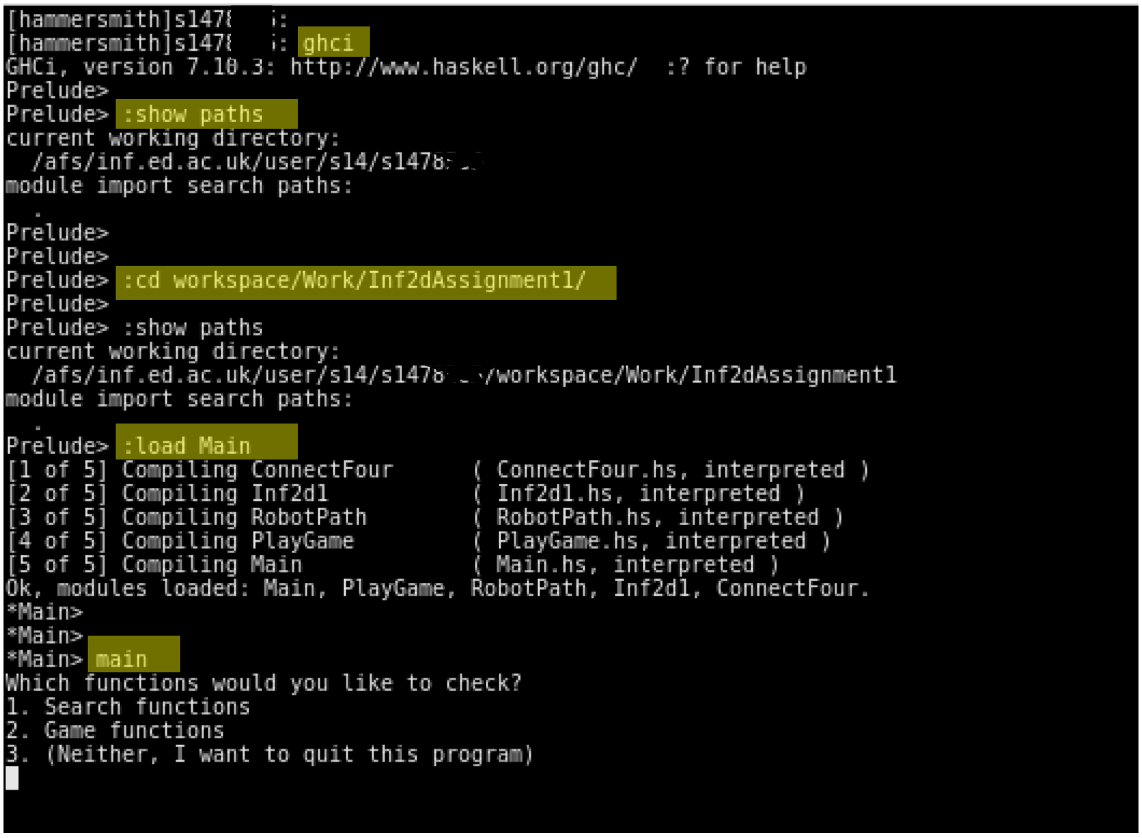
If your Haskell is quite rusty, you should revise the following topics:

- Recursion
- Currying
- Higher-order functions
- List processing functions such as map, filter, foldl, sortBy, etc
- The Maybe monad

The following webpage has lots of useful information if you need to revise Haskell. There are also links to descriptions of the Haskell libraries that you might find useful:

Haskell resources: http://www.haskell.org/haskellwiki/Haskell

[1]From `http://downloads.haskell.org/~ghc/latest/docs/html/users_guide/ghci.html`



```
[hammersmith]s147[redacted]:
[hammersmith]s147[redacted]: ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
Prelude> :show paths
current working directory:
  /afs/inf.ed.ac.uk/user/s14/s147[redacted]
module import search paths:
  .
Prelude>
Prelude>
Prelude> :cd workspace/Work/Inf2dAssignment1/
Prelude>
Prelude> :show paths
current working directory:
  /afs/inf.ed.ac.uk/user/s14/s147[redacted]/workspace/Work/Inf2dAssignment1
module import search paths:
  .
Prelude> :load Main
[1 of 5] Compiling ConnectFour      ( ConnectFour.hs, interpreted )
[2 of 5] Compiling Inf2d1           ( Inf2d1.hs, interpreted )
[3 of 5] Compiling RobotPath        ( RobotPath.hs, interpreted )
[4 of 5] Compiling PlayGame         ( PlayGame.hs, interpreted )
[5 of 5] Compiling Main             ( Main.hs, interpreted )
Ok, modules loaded: Main, PlayGame, RobotPath, Inf2d1, ConnectFour.
*Main>
*Main>
*Main> main
Which functions would you like to check?
1. Search functions
2. Game functions
3. (Neither, I want to quit this program)
```

Figure 1: Getting start with Haskell

In particular, to read this assignment sheet, you should recall the Haskell type-definition syntax. For example, a function `foo` which takes an argument of type `Int` and an argument of type `String`, and returns an argument of type `Int`, has the type declaration `foo :: Int → String → Int`. Most of the functions you will write have their corresponding type-definitions already given.

2 Help

There are the following sources of help:

- Attend lab sessions.
- Check Piazza for peer support and clarifications.
- Read any emails sent to the Inf2D mailing list regarding the assignment.
- Read ”Artificial Intelligence: A Modern Approach” Third Edition, Russell & Norvig, Prentice Hall, 2010 (R&N) Chapters 3 and 5.
- Email TA Xue Li (xue.lee@ed.ac.uk) or lab demonstrators Patrick Chen (patrick.chen@ed.ac.uk) and Zheng Zhao (s1533564@sms.ed.ac.uk).

3 Uninformed Search (35%)

In this section, you will implement some uninformed search procedures (see Chapter 3 of R&N).

The problem for which you are searching a solution is to find a path for a robot on a 6 by 6 grid. The position on grid is represented as a pair of integers in `Int` type, which has been defined as type `Node`. We use a list of nodes, defined as type `Branch`, to represent the search *agenda*: the set of nodes which are on the fringe of the search tree. In order to write search algorithms in Haskell, we need to represent the status of the search as we expand new nodes to search branches.

As a search problem we have:

- Initial state: `[start]`, where `start` is `(startRow, startColumn)`.
- Successor function: the function `next` which extends branches.
- Goal test: the function `checkArrival` checks whether the robot has reached its destination position or not.
- Path cost: the function `cost` return the cost of a path, which is its length.

Each time a node is taken from the search agenda, it is checked to see whether it is a solution to the problem. If it is a solution, the branch is returned as the desired path. Otherwise, the node is expanded using the successor function and the new nodes are added to the search agenda in the appropriate order.

3.1 6 by 6 Grid (4%)

You must first implement the following functions which represent traces in the grid world:

- `next :: Branch → [Branch]`. The first argument is an input search branch. The `next` function should expand the input branch with its possible continuations, and then return the list of expanded search branches.
  - The robot can only move up, down, left or right and cannot move off the grid, which are from position (1,1) to (6, 6). However, the `Next` function does not need to handle the case that the head of input branch is invalid.
  - The search branch generated by the `Next` function should not contain repeated node, which means that every node in a branch could only occur once.
- `checkArrival :: Node → Node → Bool`. The first argument of this function is the destination position. The second argument is the current position of the robot. The `checkArrival` function is for determining whether the robot has arrived at the destination or not.

3.2 Breadth-first Search (7%)

Breadth-first search expands old nodes on the search agenda before new nodes.

- `breadthFirstSearch :: Node → (Branch → [Branch]) → [Branch] → [Node]→ Maybe Branch`
- The first argument is the destination position in type `Node`, based on which `checkArrival` function determines whether a node is the destination position or not. The branch achieving the destination node is a solution to the search problem.
- `(Branch → [Branch])` is the type of the `next` function which expands a search branch with new nodes.
- `[Branch]` argument is the search agenda.
- `[Node]` is the list of explored nodes. Before expanding a search branch, you should check whether the current node of the search branch is an old node or not. You should not search for continuations of a repeated node.
- The function returns a value of type `Maybe Branch` which is either `Nothing`, if a solution is not found, or `Just Branch` solution, if one is found.

3.3 Depth-first Search (7%)

Depth-first search expands new nodes on the search agenda over old nodes.

- `depthFirstSearch :: Node → (Branch → [Branch]) → [Branch] → [Node] → Maybe Branch`
- The type of `depthFirstSearch` is the same as `breadthFirstSearch`, but their behaviour is different.

3.4 Depth-limited Search (7%)

Depth-limited search solves a possible failure of depth-first search in infinite search spaces by supplying depth-first search with a pre-determined depth limit. Branches at this depth are treated as if they have no successors.

- `depthLimitedSearch :: Node → (Branch → [Branch]) →[Branch] → Int → [Node] → Maybe Branch`
- The `Int` argument is the maximum depth at which a solution is searched for.

3.5 Iterative-deepening Search (10%)

Iterative deepening search tries depth-first down to a given depth, and if it cannot find a solution it increases the depth. This process is repeated until a solution is found.

- `iterDeepSearch :: Node → (Branch → [Branch]) → Node → Int → Maybe Branch`
- The third argument is the start position of the search problem, whoes type is `Node`.
- The `Int` argument is the initial depth that a solution is searched for, and subsequent depths should be generated by increasing by one every time.

4 Informed Search (30%)

In this section you will implement some heuristic search functions. See Chapter 3 of R&N.

4.1 Manhattan Distance Heuristic (5%)

The Manhattan distance, also known as *city block distance* (see Chapter 3 of R&N) gives the distance between two positions on the grid, when we can only travel up, down, left or right. It can be used as a heuristic to rank branches based upon how far they are from the goal node.

- `manhattan :: Node → Node → Int`

4.2 Best-First Search (10%)

Best-first search ranks nodes on the search agenda according to a heuristic function and selects the one with the smallest heuristic value.

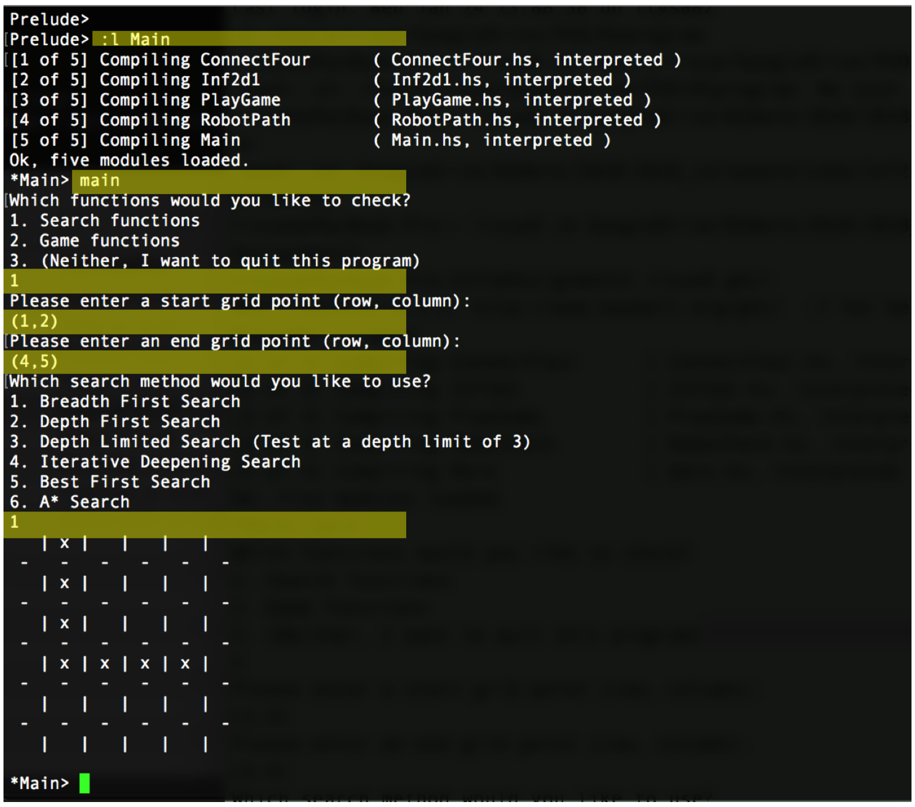
- `bestFirstSearch :: Node → (Branch → [Branch]) → (Node → Int) → [Branch] → [Node] → Maybe Branch`
- The first argument is the destination position in type `Node`, based on which `checkArrival` function determines whether a node is the destination position or not (same as uninformed search).
- `(Branch → [Branch])` is the type of the `next` function (same as uninformed search).
- `(Node → Int)` is the type of the heuristic function, which defines at least an ordering on the nodes in the search agenda.
- `[Branch]` is the search agenda (same as uninformed search).
- `[Node]` is the list of explored nodes (same as uninformed search).
- `Maybe Branch` is the value the function returns (same as uninformed search).

4.3 *A** Search (15%)

*A** search includes the cost it takes to reach the node in the ranking of nodes on the search agenda. All the other arguments are the same as `bestFirstSearch`, except the fourth argument, which is a cost function.

- `aStarSearch :: Node → (Branch → [Branch]) → (Node → Int) → (Branch → Int) → [Branch] → [Node] → Maybe Branch`
- `(Node → Int)` is the heuristic function as in `bestFirstSearch`.
- `(Branch → Int)` is the cost function which needs to be implemented by you. Here the cost of a path is its length. Both heuristic function and cost function return an `Int` type to which makes it possible to get a combined ranking of nodes.

An example of employing a search algorithm is given in Figure 2, with commands highlighted.



```
Prelude>
Prelude> :l Main
[1 of 5] Compiling ConnectFour      ( ConnectFour.hs, interpreted )
[2 of 5] Compiling Inf2d1           ( Inf2d1.hs, interpreted )
[3 of 5] Compiling PlayGame         ( PlayGame.hs, interpreted )
[4 of 5] Compiling RobotPath        ( RobotPath.hs, interpreted )
[5 of 5] Compiling Main             ( Main.hs, interpreted )
Ok, five modules loaded.
*Main> main
Which functions would you like to check?
1. Search functions
2. Game functions
3. (Neither, I want to quit this program)
1
Please enter a start grid point (row, column):
(1,2)
Please enter an end grid point (row, column):
(4,5)
Which search method would you like to use?
1. Breadth First Search
2. Depth First Search
3. Depth Limited Search (Test at a depth limit of 3)
4. Iterative Deepening Search
5. Best First Search
6. A* Search
1
   | x |   |   |   |
 -   -   -   -   -   -
   | x |   |   |   |
 -   -   -   -   -   -
   | x |   |   |   |
 -   -   -   -   -   -
   | x | x | x | x |
 -   -   -   -   -   -
   |   |   |   |   |
 -   -   -   -   -   -
   |   |   |   |   |

*Main> 
```

Figure 2: Employ Breadth First Search in GHCi.

5 Games (35%)

In this section you will implement minimax search and minimax search with alpha-beta pruning (see Chapter 5 of R&N) for the two-player game **Connect Four**. The board size is 4 columns and 4 rows..

Rules introduction:

1. Two players take turns adding one piece at a time.
2. For each column, only the lowest unoccupied cell is the available space for a player to add a piece. An example is given by Figure 3.
3. The winner is the player who is the first to form a horizontal, vertical, or diagonal line of four of one's own pieces. If the game board fills before either player achieves four in a row, then the game is a draw.

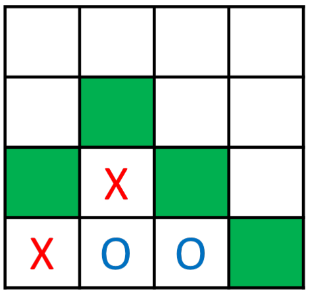


Figure 3: An example for available cells to add a piece. Given existing moves by MAX(X) and MIN(O), the available cells have been highlighted in green.

5.1 Game Functions

We will represent the 4 by 4 grid for the game board as a list type with 16 elements. Your game player will interact with the game by providing the (row, column) pair for his/her moves. Several useful functions for the Connect Four game are defined in ConnectFour.hs.

- `Game, Cell` and `Role` types: The `Game` type is a list of `Int` types. It represents the 16 cells on the game board. The `Cell` type represents positions on the board as the `(row,column)`, an `Int` pair. The `Role` type describes which player it is, the human player or computer player.
- `getLines` provides all possible lines on the board (rows, columns and diagonals) for winning the game.
- `terminal` function takes a `Game` state and returns `True` if it is a terminal state or `False` if otherwise.
- `moves` takes a `Game` and a `Role` as arguments and returns a list of possible game states that the player can move into.
- `checkWin` takes a `Game` and a `Role` and determines if the game is a winning position for the player.
- `switch` allows you to alternate roles of players, e.g. MAX and MIN.

5.2 Evaluation Function (5%)

For this section, you will implement an evaluation function for terminal states. The function determines the score of a terminal state, assigning it a value of +1, -1 or 0 as defined below:

- `eval :: Game → Int`
- `eval` takes a `Game` as an argument, and should return an `Int` type so that game states can be ranked.
  - A winning position for MAX should be +1.
  - A winning position for MIN should be -1.
  - A draw has a 0 value.
  - The `terminal` and `checkWin` functions defined in ConnectFour.hs will be useful.

5.3 Minimax Algorithm (15%)

The **minimax** value of a state represents the value of the best possible outcome from that state. This assumes that the MAX player plays moves which maximise the value and MIN plays moves which minimise the value (see Figure 4). The minimax algorithm from R&N is shown in Figure 5, however note that your function should return the best **evaluation value** reached from the input game state, not the action that achieves this value.

- `minimax :: Role → Game → Int`
- `minimax` takes a type `Role` and a `Game` as arguments.
- `minimax` returns an `Int` since the terminal evaluation of a game is either +1, a win for player MAX; 0, a draw; or -1, a win for player MIN.
- Your `minimax` function should use the `eval::Game → Int` function to evaluate terminal states.

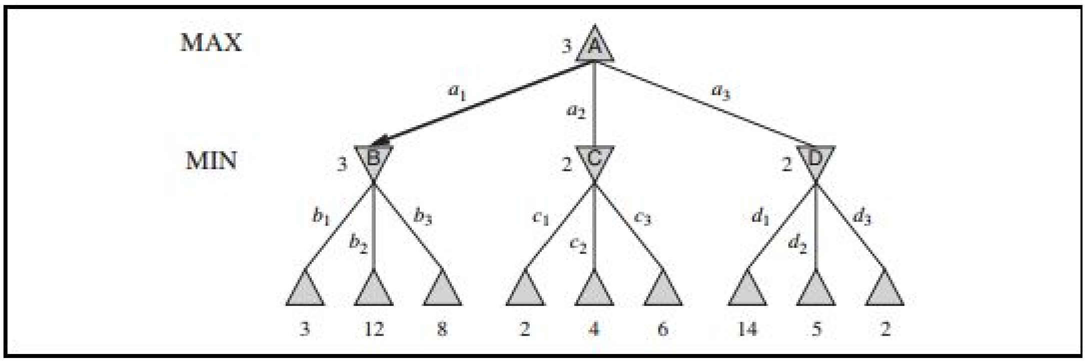
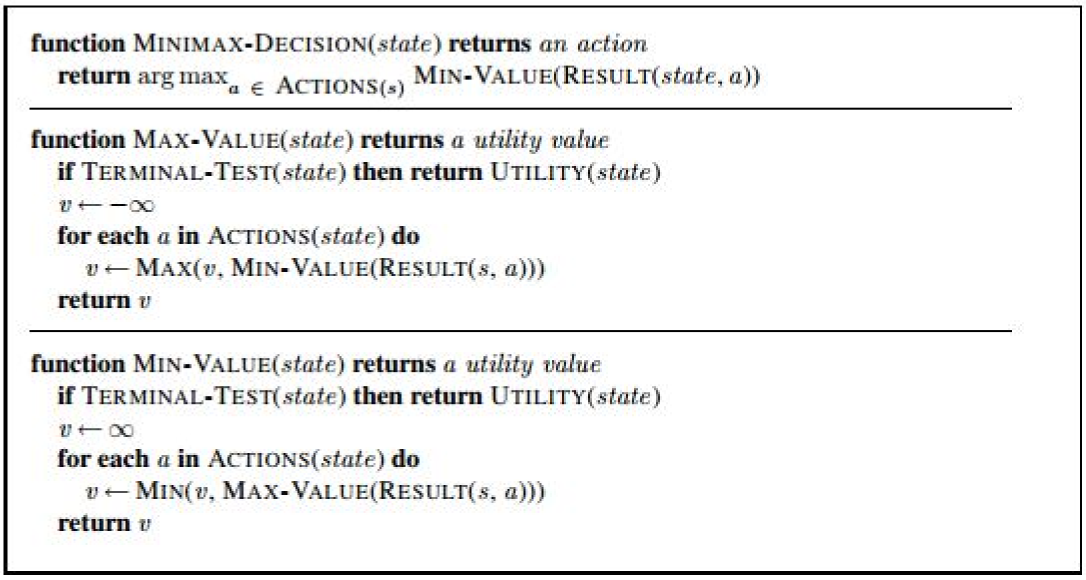


Figure 4: The MAX player can play moves to get to either B, C, or D. MIN can then play moves which result in different possible valuations. For each of B, C, or D MAX must find what the least value MIN could select is. For example from state B MIN could play either b_1, b_2 or b_3 to reach states with valuation 3,12 or 8 respectively. So, if the state was B MIN would play b_1 to reach the state with valuation 3. Since the best response for MIN in C and D is 2, the best state for MAX is B, hence MAX should play move a_1. (From R&N)



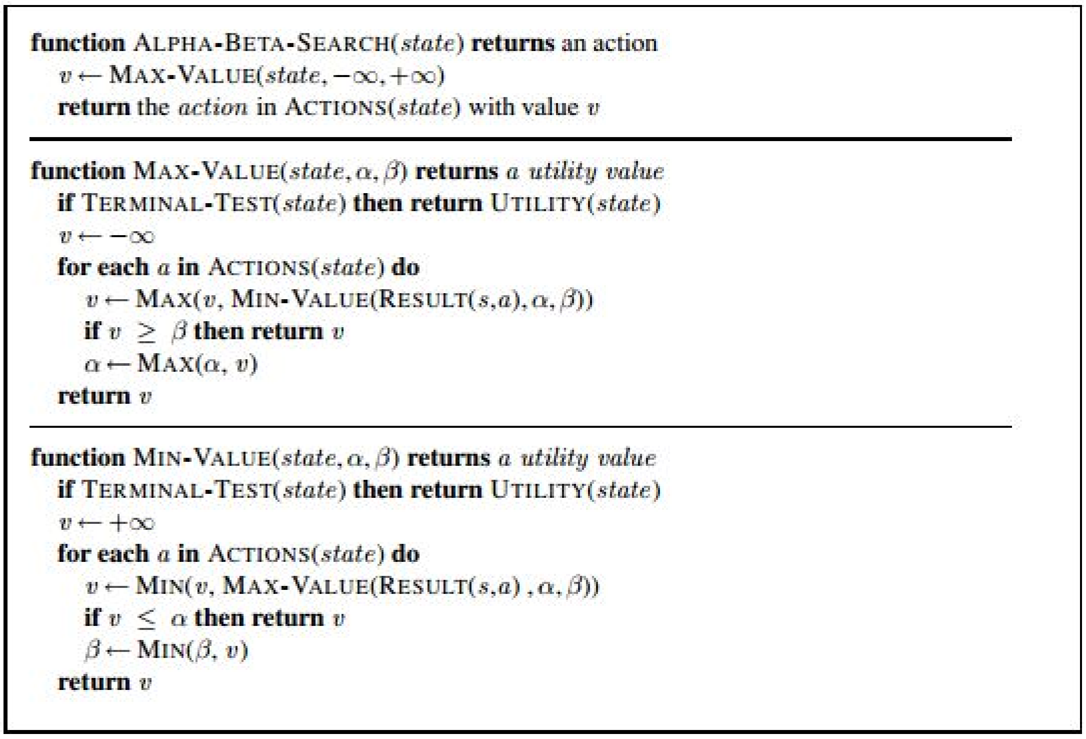
```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
---
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v
---
function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Figure 5: An algorithm for calculating minimax decisions (From R&N)

5.4 Alpha-Beta Pruning (15%)

Alpha-beta finds the minimax value of a state, but searches fewer states than the regular `minimax` function. It does this by ignoring branches of the search space which cannot affect the minimax value of the state. A range of possible values for the minimax value of the state is calculated. If a node in a branch has a value outside this range then the rest of the nodes in that branch can be ignored, as the player can avoid this branch of the search space (see Figure 6). You will have to choose an initial range for alpha-beta pruning. We can't represent infinite numbers in Haskell, so you should use the range (-2, +2) for your `alphabeta` function. Since the maximum evaluation of a game position is +1 and the minimum is −1, this range covers all possible minimax values for the state.

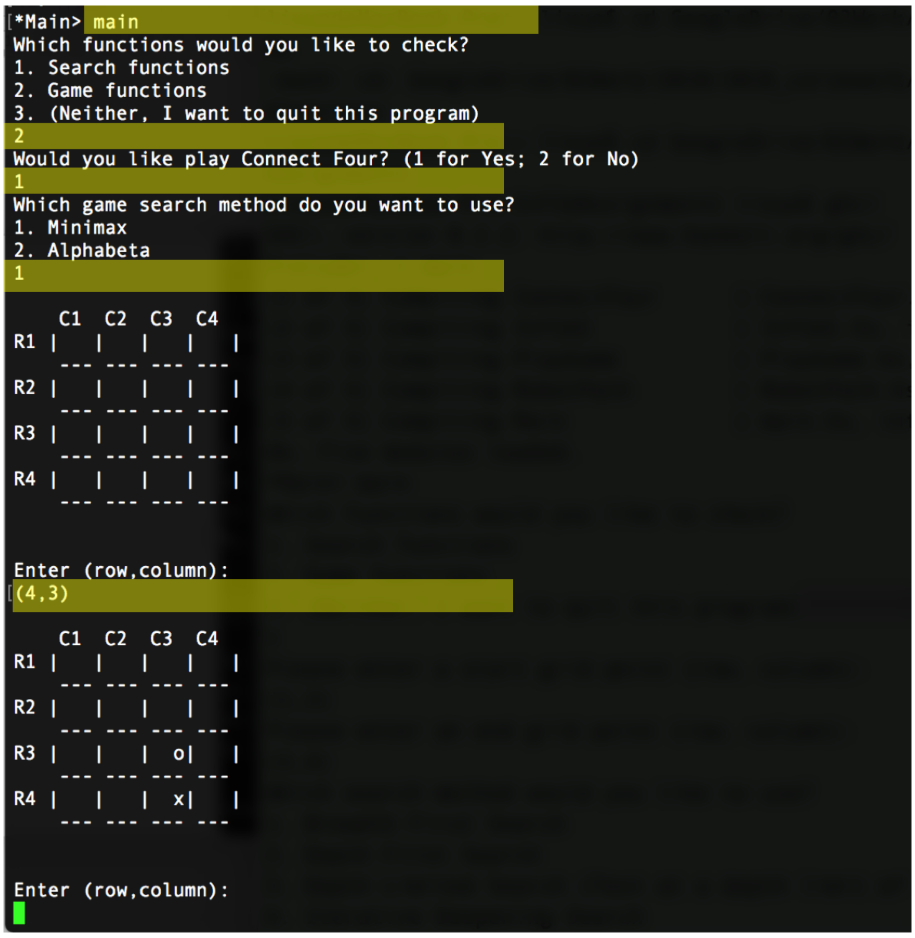
- `alphabeta :: Role→ Game → Int`
- Your `alphabeta` function should use the `eval :: Game → Int` function to evaluate terminal states.



```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
---
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
---
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 6: The alpha-beta search algorithm (From R&N)

An example of playing one step in Connect Four is given in Figure 7, with commands highlighted.



```
[*Main> main
Which functions would you like to check?
1. Search functions
2. Game functions
3. (Neither, I want to quit this program)
2
Would you like play Connect Four? (1 for Yes; 2 for No)
1
Which game search method do you want to use?
1. Minimax
2. Alphabeta
1

    C1  C2  C3  C4
R1 |   |   |   |   |
    --- --- --- ---
R2 |   |   |   |   |
    --- --- --- ---
R3 |   |   |   |   |
    --- --- --- ---
R4 |   |   |   |   |
    --- --- --- ---


Enter (row,column):
[(4,3)

    C1  C2  C3  C4
R1 |   |   |   |   |
    --- --- --- ---
R2 |   |   |   |   |
    --- --- --- ---
R3 |   |   |  o|   |
    --- --- --- ---
R4 |   |   |  x|   |
    --- --- --- ---


Enter (row,column):
```

Figure 7: Play Connect Four in a terminal.

6 Notes

Please note the following:

- To ensure maximum credit, make sure you have commented your code and that it can be properly loaded on GHCi before submitting.
- All search algorithms in this assignment are expected to terminate within 2 minutes or less. If any of your algorithms take more than 2 minutes to terminate, you should make a comment in your code with the name of the algorithm, and how long it takes on average. You may lose marks if your algorithms do not terminate in under 3 minutes of runtime for any of the search problems.
- You are free to implement any number of custom functions to help you in your implementation. However, you must comment these functions explaining their functionality and why you needed them.
- Do not change the names or type definitions of the functions you are asked to implement. Moreover, you may not alter any part of the code given as part of the CSP framework. You may only edit and submit your version of the **Inf2d1.hs** file.
- You are strongly encouraged to create your own tests to test your individual functions (these do not need to be included in your submitted file).
- Ensure your algorithms follow the pseudocode provided in the books and lectures. Implementations with increased complexity may not be awarded maximum credit.

7 Submission

You should submit your version of the file *Inf2d1.hs*, that contains your implemented functions, in two steps:

- Step1: Rename *Inf2d1.hs* into `Inf2d_ass1_s⟨matric⟩.hs`, where ⟨matric⟩ is your matriculation number (e.g 1778565).
- Step2: Submit `Inf2d_ass1_s⟨matric⟩.hs` using the following command:

submit inf2d cw1 Inf2d_ass1_s⟨matric⟩.hs

Your file must be submitted by **3pm, Thursday 13th March 2018**.

You can submit more than once up until the submission deadline. All submissions are timestamped automatically. Identically named files will overwrite earlier submitted versions, so 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.

Warning: Unfortunately the `submit` command will technically allow you to submit late even if you submitted before the deadline (i.e. it does not enforce the above policy). Dont do this! We will mark the version that we retrieve just after the deadline, and (even worse) you may still be penalized for submitting late because the timestamp will update.

For additional information about late penalties and extension requests, see the School web page below. Do **not** email any course staff directly about extension requests; you must follow the instructions on the web page.

`http://web.inf.ed.ac.uk/infweb/student-services/ito/admin/coursework-projects/late-coursework-extension-requests`

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

`https://www.ed.ac.uk/academic-services/students/conduct/academic-misconduct`

and at:

`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).