Informatics 2D. Coursework 1:
Search and Games

Stefanie Speichert, Vaishak Belle

January 30, 2019

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/A1.zip

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 12th March 2019.

Some basic commands of Haskell are listed below[1].

- `ghci` GHCi is the interactive environment, in which Haskell expressions can be evaluated and programs can be interpreted. You can fire up GHCi with the command `ghci` in a terminal. All following commands work in GHCi environment.
- `:help` Get information if you are stuck.
- `: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. You can also use the short version `:l Main`, where Main is the topmost module in our assignment.
- `main` Run function 'main' (defined in the Main.hs file).

If your Haskell is a bit 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

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`. All of the main functions you will write have their corresponding type-definitions already given. However, if you declare your own helper functions you will have to write these yourself.

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

2 Help

There are the following sources of help:

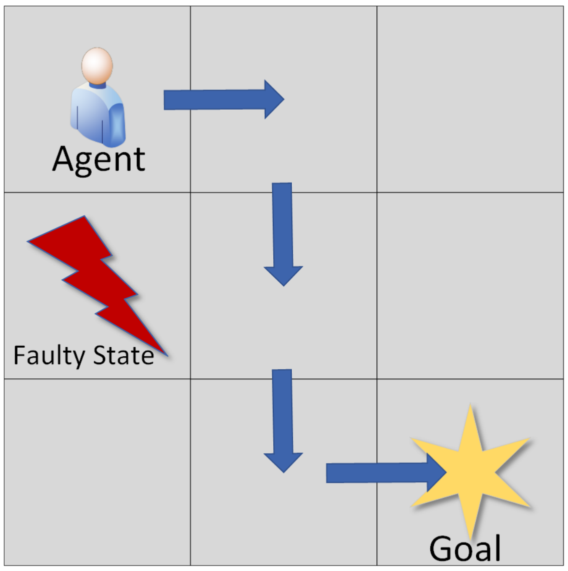
- Attend the lab sessions.
- Check Piazza for peer support and clarifications.
- Read "Artificial Intelligence: A Modern Approach" Third Edition, Russell & Norvig, Prentice Hall, 2010 (R&N) Chapters 3 and 5.
- Email the TA: Stefanie Speichert (s.speichert@ed.ac.uk) She'll also be happy to meet with you if you have a longer (but concrete!) question.
- Read any emails sent to the Inf2D mailing list regarding the assignment.

Do not be afraid to ask for help! While we are not able to give you solutions to the problems we know how to point you in the right direction or help you out when you are stuck!

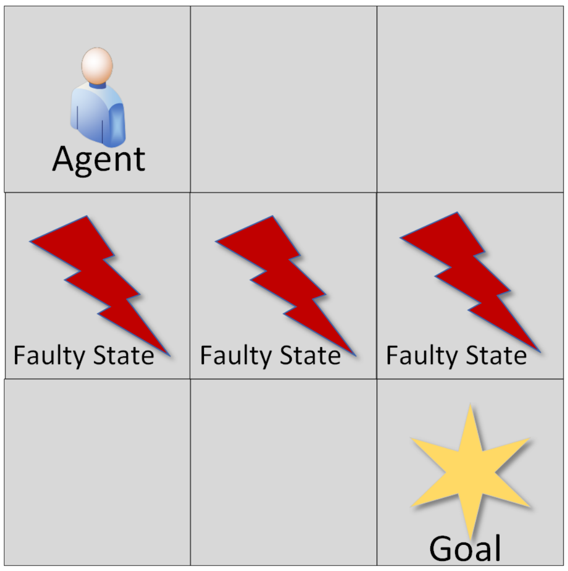
3 Uninformed Search (35%)

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

Problem Statement: You will build a goal orientated agent. The agent operates on a 6 X 6 grid and its aim is to *reach a specified goal state.* However, there is one problem. The grid is rusty and some positions can't be reached!



(a) A world where there is one faulty state. The agent learns to move around it and is able to reach the goal.



(b) A world where there are three faulty states. Since the agent **CAN'T** travel through a faulty state, it will never reach its goal.

Figure 1: Two different scenarios in a 3x3 Grid world our agent can find itself in.

Your goal is to implement the learned search structures with the addition that some states can't be reached. The position on the grid is represented as a pair of integers in `Int` type, which has been defined as type `Node`. The position on the top left corner, for example would be denoted as '(1,1)'. 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.
- A list `badNodesList` that denotes the faulty nodes which the agent is not allowed to use.
- 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, you will have to check if it is the destination node. 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. Note that you can have any number of faulty nodes and therefore might not be to reach the goal state at all!

3.1 6 by 6 Grid (6%)

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.
  - The robot cannot move to a state if it is faulty! Your function should include a check against the `BadNodesList` and only add the position/node if it is not in the list.
- `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 (6%)

Breadth-first search expands old nodes on the search agenda before new nodes. You will implement the function and return a search path if the goal was reached or an error otherwise.

- `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. Haskell will automatically throw an error if a function returns `Nothing`. As long as you account for the goal not being found you can simply return `Nothing` and not worry about error declarations.

3.3 Depth-first Search (6%)

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

- `depthFirstSearch :: Node → (Branch → [Branch]) → [Branch] → [Node] → Maybe Branch`
- The types used in the declaration of `depthFirstSearch` are the same as `breadthFirstSearch`. You will simply have to modify the body to perform depth first search.

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 or a maximum depth is reached (To avoid never terminating functions in case the goal state cannot be reached. Your task is twofold:

1. First you have to choose a good value for `MaxDepth`. Your value should ensure that a solution can be found if one exists (that is, it should not cut off too early). At the same time it should ensure that no extra computation time is used. For example, if the maximum depth of a grid is 50 then choosing 100 is not a good value. You will also add a comment describing why you choose the value you choose.

   Hint: You can calculate the optimal number using the grid parameters.

2. Once an optimal value is set you can go about implementing the function. The function is typed as follows:
   - `iterDeepSearch :: Node → (Branch → [Branch]) → Node → Int → Maybe Branch`
   - The third argument is the start position of the search problem, whose 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. The search should terminate once a solution is found or the maximum depth is reached.

4 Informed Search (30%)

In this section you will implement some heuristic search functions. See Chapter 3 of R&N. For simplicity, we will assume now that **the goal is always reachable!**

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.

5 Games (35%)

In this section you will implement min-max search and min-max search with alpha-beta pruning for two games: Tic Tac Toe and Wild Tic Tac Toe. You will also experience why alpha-beta pruning is so powerful. We start with normal Tic Tac Toe:

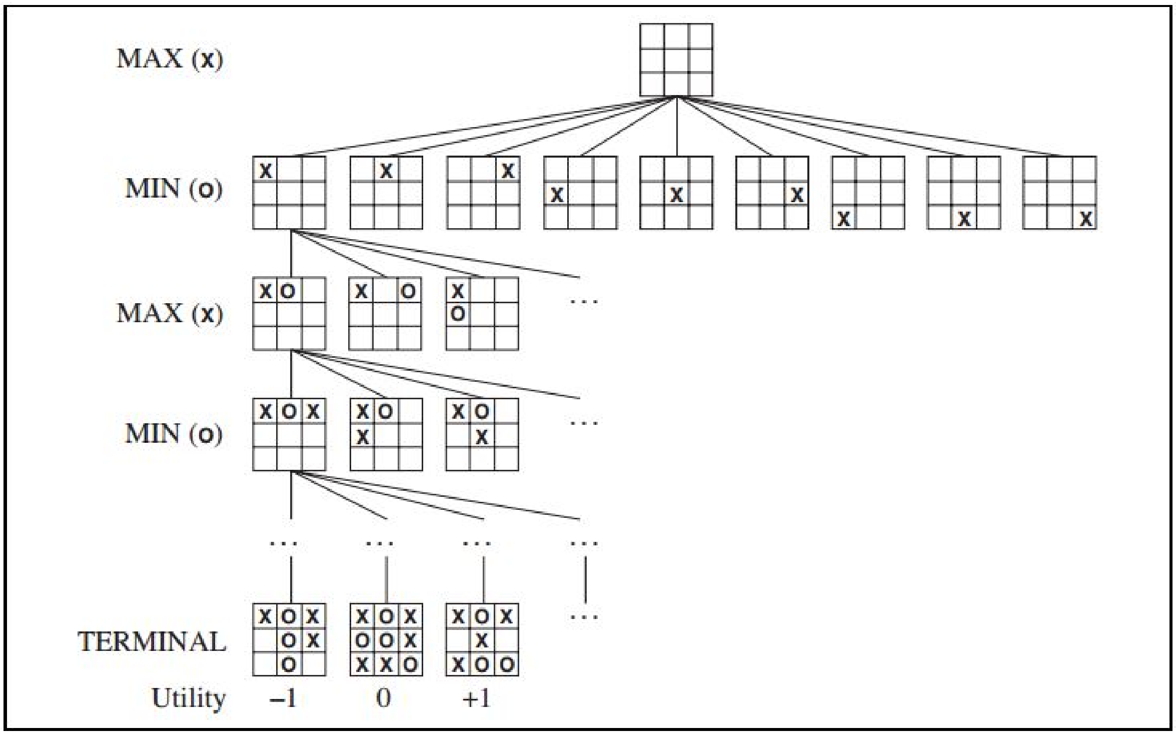


Figure 2: A (partial) game tree of Tic-Tac-Toe (From R&N)

5.1 Tic Tac Toe Game functions

We will represent the 3 by 3 grid for the game board as a list type with 9 elements. Your game player will interact with the game by providing the (row, column) pair for their moves. Several useful functions for the Tic-Tac-Toe game are defined in the TTTGame.hs function.

- **Game, Cell** and **Line** types: The **Game type is a list** of **Int** types. It represents the 9 cells on the game board. The **Cell** type represents the *(row,column) Int* pairs that the human player enters. The **Line** type holds all possible lines on the board (rows, columns and diagonals).
- **getCellIndex** takes a (row,column) tuple and maps it to the corresponding index in the list representation of the game board.
- **terminal** function takes a Game and returns true if it is a terminal state or False if otherwise.
- **moves** takes a Game and Player as arguments and returns a list of possible game states that the player can move into.

- **checkWin** takes a Game state and a Player and determines if the game state is a winning position for the player.
- **switch** allows you to alternate between max and min players.

5.2 Tic Tac Toe: 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** take a **Game** as a 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 TTTGame.hs will be useful.

5.3 Tic Tac Toe: Minimax algorithm (12%)

The **min-max** value of a state represents the best 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 3). The minimax algorithm from R&N is shown in Figure 4, however note that your function should return the value of the best state, not the action that achieves this value.

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

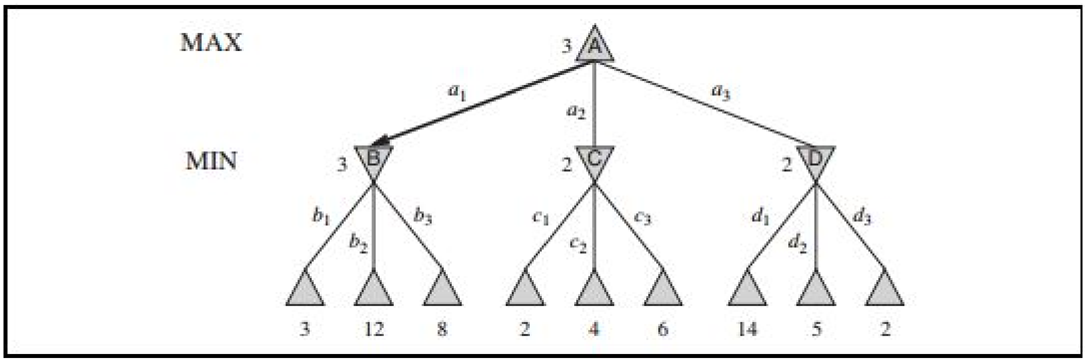
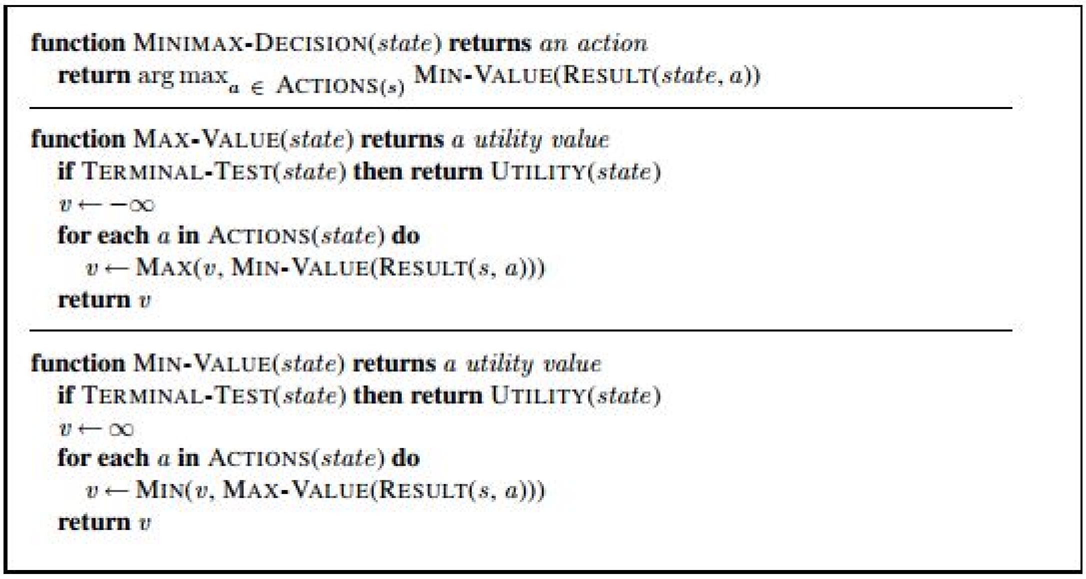


Figure 3: The max player can play moves to get to either B, C, or D. Min can then play moves which result in different possible valuations. For each of B, C, or D max must find what the least value min could select is. For example from state B min could play either b_1, b_2 or b_3 to reach states with 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)



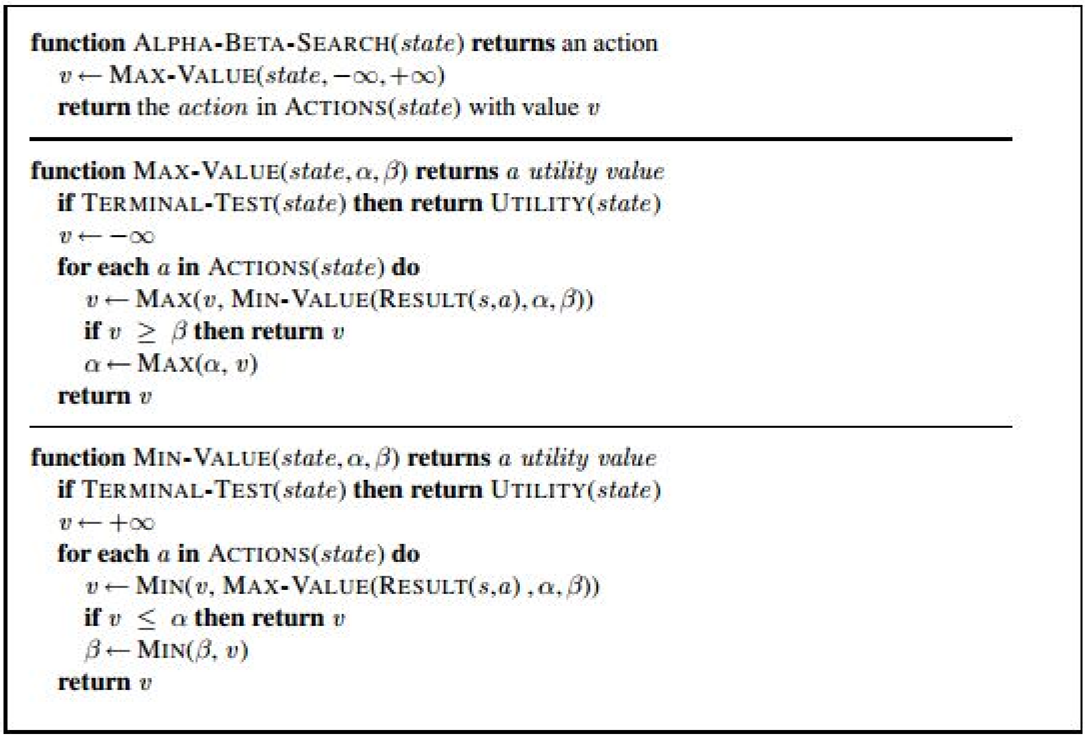
```
function MINIMAX-DECISION($state$) returns $an\ action$
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT($state$, $a$))

function MAX-VALUE($state$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s$, $a$)))
  return $v$

function MIN-VALUE($state$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow \infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$, $a$)))
  return $v$
```

Figure 4: An algorithm for calculating minimax decisions (From R&N)

5.4 Tic Tac Toe: Alpha-Beta pruning (12%)

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 5). 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 :: Game → Player → Int**
- Your **alphabeta** function should use the **eval :: Game → Int** function to evaluate terminal states.



```
function ALPHA-BETA-SEARCH($state$) returns an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value $v$

function MAX-VALUE($state, \alpha, \beta$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s$,$a$), $\alpha$, $\beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE($state, \alpha, \beta$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$,$a$) , $\alpha$, $\beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Figure 5: The alpha-beta search algorithm (From R&N)

5.5 Wild Tic Tac Toe: Introduction and Helper Functions

In Wild Tic Tac Toe each player can move both symbols. That is once it's a player's turn, they will first choose the symbol they want to play this turn (x or o) and where they want to place it. You will see this when you select to play Wild Tic Tac Toe in our Haskell code. The computer will then have to decide which symbol to play as well as where to place it in response to your move.

The helper functions are very similar to the ones in Tic Tac Toe and can also be found in the same file:

- ***Unchanged :*** **Game, Cell** and **Line** types: The **Game** type is a list of **Int** types. It represents the 9 cells on the game board. The **Cell** type represents the *(row,column) Int* pairs that the human player enters. The **Line** type holds all possible lines on the board (rows, columns and diagonals).
- ***Unchanged :*****getCellIndex** takes a (row,column) tuple and maps it to the corresponding index in the list representation of the game board.
- ***Unchanged :*****terminal** function takes a Game and returns true if it is a terminal state or False if otherwise.
- **movesWild** takes a Game and Player as arguments and returns a list of possible game states that the player can move into. You will see that the function now considers moves with both symbols.
- **checkWin** takes a Game state and a move type (*x, encoded as 1* or *o, encoded as 0*) and determines if the game state is a winning position for the specific move type.
- ***Unchanged :*****switch** allows you to alternate between max and min players.

5.6 Wild Tic Tac Toe: Evaluation Function (3%)

You will see that it doesn't take much to change our previous code to accustom Wild Tic Tac Toe. First you will implement the new evaluation function *evalwild* which will return 1 if either symobol, that is either x or o, is in a winning position and 0 if we landed on a terminal draw.

- **evalWild :: Game → Int**
- **evalWild** take a **Game** as a argument, and should return an **Int** type so that the game states can be ranked.
- A winning position of symbol x should be rewarded with 1.
- A winning position of symbol o should be rewarded with 1.
- A **draw** has a 0 value.
- The **terminal** and **checkWin** functions defined in TTTGame.hs will be useful.

5.7 Wild Tic Tac Toe: Mini-Max with Alpha Beta Pruning (3%)

You will now modify your minimax with alpha-beta pruning function from before to play Wild Tic Tac Toe. The structure of the algorithm stays the same. However you will have to exchange some calls to the helper functions and the evaluation function with *evalWild.* You will also have to modify your code to give a reward of -1 if the player who made the last move before reaching a terminal state that resulted in a win was the computer (the min-player) and +1 if if the player who made the last move before reaching a terminal state that resulted in a win was the human player (the max-player). Remember: evalWild only returns 1!

- **alphabetaWild :: Game → Player → Int**
- **Modifications:**
  - Your **alphabetaWild** function should use the **evalWild :: Game → Int** function to evaluate terminal states.
  - Change the calls to the helper functions to play Wild Tic Tac Toe instead of Tic Tac Toe.
  - Make sure that the reward for winning as min-player is still -1 and the reward for the max-player is still 1.

You might have noticed that we did not ask you to implement Wild Tic Tac Toe only with the minimax algorithm. The reason for this is that the search space in Wild Tic Tac Toe is significantly bigger than Tic Tac Toe. You can see this by comparing the times it takes for alphabeta and alphabetaWild to find a good next move: You had to wait a bit longer when testing alphabetaWild rather than alphabeta. Now, imagine, we didn't prune at all! The pure minimax will take a significantly longer time to find a move. We encourage you to try it (this is not graded!) and you will see for yourself that you have to wait quite a bit before the algorithm returns a single next step.

6 Notes

Please note the following:

- We mark coding style as well as correctness, make sure you comment your code accordingly!
- To ensure maximum credit, make sure that your code 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 1686631).
- 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 **3 pm, Tuesday 12th March 2019**.

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.

If you got an extension and have submitted before the deadline, it is **YOUR** responsibility to get in touch with Stefanie Speichert (the TA) to tell her that this happened otherwise we cannot guarantee that the version that you submit within your extension period will be the one to be graded.

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).