CSE 3401 Assignment 4
Winter 2013

Date out: March 26. Date due: April 6, at 11:55 pm

- The submitted assignment must be based on your individual work. Review the Academic Honesty Guidelines for more details.
- This assignment constitutes 6% of your total mark for the course and is marked out of (70 + 5).
- You should adhere to the coding guidelines posted on the website; comment your code and test it thoroughly.
- You may define any auxiliary relations if that helps in defining the required predicates.
- You should Submit 3 (or optionally 4) files for this assignment:
  - a4.pl, which is the source code of your solutions for questions 1 and 2. Start from the file a4.txt (available in the resources section of this assignment) which already contains the initial configuration and you need and fill in the missing code. Rename this file to a4.pl before editing.
  - a4.pdf, which contains your answer for question Q2.a (and optionally Q3.b)
  - a4test.txt, which consists of the test cases you have used to test questions 1, 2 and optionally 3, as well as the results of testing. Include a header with your name, student number and cse login.
  - Optional: abplay.pl, which consists of the implementation of alpha-beta pruning algorithm for this assignment.

Soft Copies: Gather all the required files in a directory named `a4answers` and submit it electronically by the deadline. To submit electronically, use the following Prism lab command:

```
submit 3401 a4 a4answers
```

Alternatively, you may use web submit (https://webapp.cse.yorku.ca/submit/) and choose the correct course and assignment number to upload your files.

Question 1: A two player game (55 Points)

In this assignment you are going to code a 2 player game using an implementation of an interactive depth-first minimax game tree search routine in the file play.pl. This file is available online in the resources section of this assignment. You will not change this file, but please read carefully the code there to see what predicates must be implemented and how they are used in the tree search.

A simple example of the game of tic-tac-toe that shows how to use this algorithm is also provided in the resources section (ttt.pl). All the files are uploaded with txt extension, and the names are all in lowercase letters. Rename the files play.txt, ttt.txt and a4.txt, rename them to play.pl, ttt.pl and a4.pl respectively.

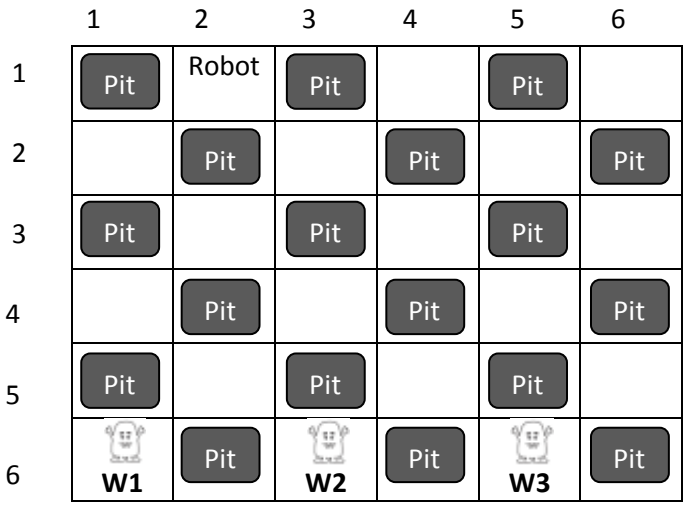

Figure 1: A starting configuration (a)

This game is played on a 6×6 grid of squares. There are 2 players in this game, a robot and a team of 3 Wumpuses (w1, w2 and w3). An initial configuration of the game is shown in Figure 1 with the Wumpuses at row 6, and the robot at (1,2). The robot always starts at row 1, from any of the unoccupied cells, and the Wumpuses always start at row 6, one in each cell. Other possible initial configurations are illustrated in Figure 2. The robot and the Wumpuses cannot move to squares that contain a pit.

Both the Wumpuses and the robot can move diagonally to unoccupied squares. In each round of the game:

- The robot can move both forward and backward, in any of the 4 diagonal directions, one or two squares at a time. For example, from (1,2), the robot can move to (2,1) , (2,3) or (3,4).
- The Wumpuses can move forward only (towards the first row), only one square at a time. As there are 3 Wumpuses, only one of them is allowed to move in each round. For example, in a certain round, w2 (6,3) can move to (5,2) or (5,4).

None of the players are allowed to remain in their current location and must make a move when it is their turn to play. The first player that is unable to move in a certain state (i.e. does not have any legal moves), loses the game. In this game, robot is played by the human player (who plays first) and the Wumpuses are controlled by the system. The system can only move one of its Wumpuses at each turn.

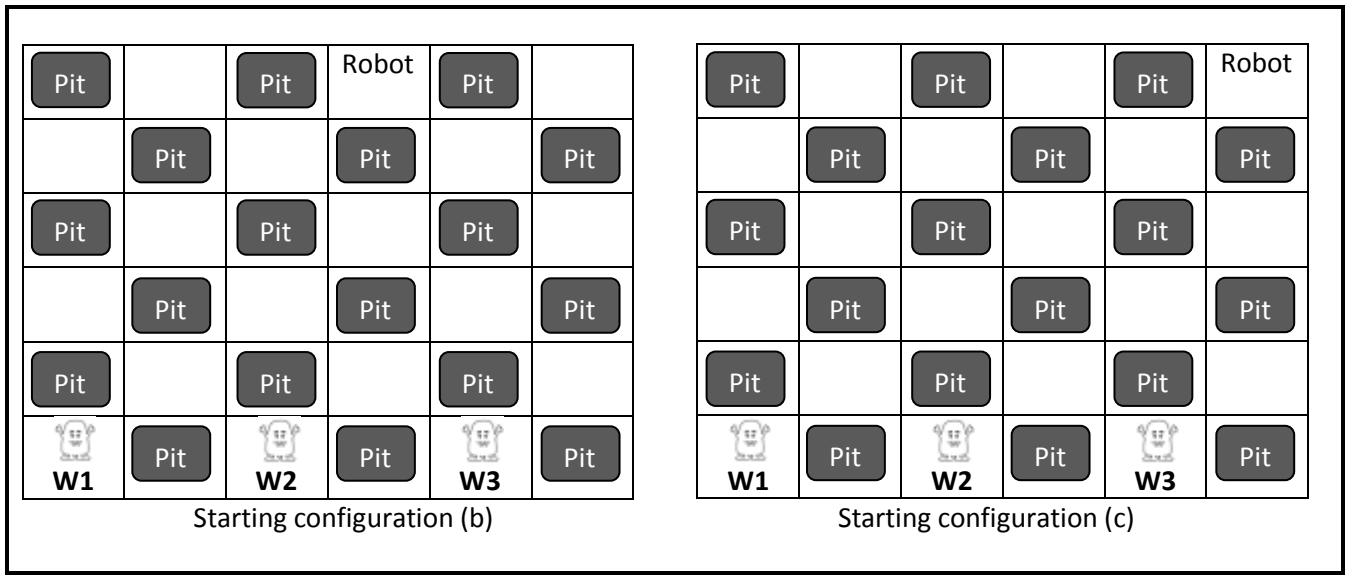

Figure 2: Possible starting configurations

The predicate `config/2` in file a4.pl provides the initial representation of the grid in Figure 1. You have to define various predicates to interface with the game tree search routine. Your implementation will include the following:

Q1.a State Representation and initialization
Implement the predicate `initialize(InitialState,InitialPlyr)` which holds if and only if `InitialState` is the initial state and `InitialPlyr` is the player who moves first.

Q1.b Winner
Implement the predicate `winner(State,Plyr)` which returns the winning player, `Plyr`, if `State` is a terminal position.

Q1.c Tie
Implement the predicate `tie(State)` which is true if terminal `State` is a "tie" (no winner). Note that there are no ties in this game.

Q1.d Terminal State
Implement the predicate `terminal(State)` which is true if `State` is a terminal.

Q1.e Moves
Implement the predicate `moves(Plyr,State,MvList)` so that it returns a list `MvList` of all legal moves `Plyr` can make in the given state `State`.

Q1.f Next state
Implement the predicate `nextState(Plyr,Move,State,NewState,NextPlyr)`. Given that `Plyr` makes `Move` in `State`, it determines `NewState` (i.e. the next state) and `NextPlayer` (i.e. the next player who will move).

Q1.g Valid Move
Implement the predicate `validmove(Plyr,State,Proposed)` which is true if `Proposed` move by `Plyr` is valid at `State`.

Q1.h Evaluation Function
Implement the predicate `h(State,Val)` which given `State`, returns heuristic `Val` of that state: larger values are good for Max, smaller values are good for Min. If State is terminal, then `h` should return its true value. If `State` is not terminal `h` should be an estimate of the value of state (see Question 2 for more details).

If you decide NOT to do Question 2, to get credit for Question 1, you need to define a very simple heuristic instead: your `h(State,Val)` can return `Val` = 0 for any non-terminal state `State`. If `State` is a terminal state, `h` must return a positive value (say 100) for a win state, and a negative value (say -100) for losing state, and 0 for a tie state. Clearly, this `h` provides no guidance in the depth-bounded search.

Q1.i Lower Bound
Implement the predicate `lowerBound(B)`, which returns a value `B` that is less than the actual or heuristic value of all states.

Q1.j Upper Bound
Implement the predicate `upperBound(B)`, which returns a value `B` that is greater than the actual or heuristic value of all states.

It is recommended that you develop two helper predicates: `set` (that sets the value of the cell at a certain position of the grid) and `get` (that returns the value of the cell at a certain position of the grid). You can use the built-in predicate `nth1/3` for locating elements in the grid.

You may define any number of valid configurations and use them as test cases too. In your answers, do not use features of Prolog that have side effects, such as `assert` and `retract`.

How to play:

To invoke the interactive shell you need to type the query:
```
?- play.
```

Assuming all the required predicates have already been defined, the interactive game playing shell will prompt the human player to input moves. The player can enter a move. In play.pl, the predicate

`read(Proposed)` is used to read the user's move—this will bind the variable `Proposed` to anything the user enters; your implementation of the predicate `validmove/3` is used to check that they have entered a valid move in the right syntax.

When it is the computer's turn, the engine will invoke a mini-max search for the best move. This search is done to a bounded depth, and you can set the depth bound (refer to the documentation in play.pl and the predicate `mmeval/6` on how to do this). You should set a bound that yields reasonable performance (less than 16 Seconds). In your comments in a4.pl, mention clearly what depth bound you have used.

Question 2: State Evaluation Function (15 Points)

Q2.a Design an evaluation function that returns a value for each state. In designing such an evaluation function, you may consider a number of factors (features), and their importance (represented as a weight):

Eval(s) = feature_1(s) × weight_1 + … + feature_n(s) × weight_n

Describe clearly the features you have considered and any formula you might have used. For example, one feature may relate to the fact that in general, it is better for the robot to move towards row 6, since Wumpuses can only move forward towards row 1.

Q2.b Implement the heuristic function you designed in Q2.a, so that the predicate `h(State,Val)`, given `State`, returns heuristic `Val` of that state. To test the evaluation function, you need to show that the computer/Wumpus team player performs better with the evaluation function. This means showing that there are games where it wins while the default player would have lost. It could also mean it selects moves faster or can search deeper in the same amount of time.

Optional: Question 3: Alpha-Beta Pruning (5 Bonus Points)

Q3.a The play predicate is based on depth-bounded, depth-first, minimax evaluation; but it does no pruning. This part asks you to replace the predicate `mmeval(Plyr,State,Value,Move,Depth, StatesSearched)` with a new predicate `abmmeval`, that evaluates states using minimax with alpha-beta pruning. The arguments to this predicate can be of your own choosing.

Place your alpha-beta implementation in a file called abplay.pl. This file should contain all of the functionality of play.pl except that `abmmeval` replaces `mmeval`. To do this, first copy play.pl into a new file called abplay.pl, and then make necessary changes there.

Q3.b Provide a comparison between running the game with and without the Alpha-Beta pruning.