Simple Game Solving Using Haskell

Daniel Leblanc

March 18, 2014

1 Introduction

A solved game is one for which, if we assume both players play perfectly, the outcome is known. Many games are much too complicated to completely explore the possible game states. Games such as Chess and Go have too many possible moves from each board position to ever be able to fully explore without vastly superior computing power to what is now available. To give you an idea of the potential complexity, Checkers, which is the most complicated game to be solved to date, has 500 billion billion possible board positions (5×10^{20}) [Jonathan Schaeffer(2007)] and took nearly two decades of constant processing to solve.

The vast majority of two player games can be mapped to decision problems that are PSPACE-complete. This puts them in a category of problems that are at least as difficult to solve as NP-complete problems. Despite this some simpler games have small enough game trees that we can fully explore them in reasonable amounts of time.

Here we experiment with implementing a game solver in Haskell for several simple two player games. Haskell is an excellent choice when exploring a complete game tree, as it allows us to avoid the traditionally tedious task of rewinding the board positions after each move.

All code for the project is available at my github account, `https://github.com/crophix/GameSolver`, as well as in the appendices.

2 Adversarial Search

Adversarial search is a branch of artificial intelligence that attempts to create optimal players for various games. It creates a tree of possible moves and then searches the game tree looking for the best possible sequence of moves. The algorithm we are using here is called *negamax* and looks for the worst move for the opponent at each step in the search. Figure 1 shows some simple pseudocode. In more complex games each board position needs to be scored even if it isn't a final game state. For our purposes we only score boards when the game is in a termination state, since we are not trying to produce an optimal player, just solve the game assuming optimal play. This allows us to not limit our running time to human patience, and instead completely explore the game tree.

There are several optimizations that can be done to improve the performance of negamax search such as $alpha - beta$ pruning and limiting the depth of the search that we have ignored for our initial implementation, but could easily be added later. Further improvements can also be made by taking into account board rotations and translations to reduce the size of the potential game tree. This however would take significant additional work and has been ignored for this paper.

```
negamax(g)
    if g is a final state (game over)
        return the value of g
    v <- -1
    for each legal move m in g
        g' <- m(g)
        v' <- -negamax(g')
        if v' > v
            v <- v'
    return v
```

Figure 1: Pseudocode for negamax search

In Haskell *negamax* has been implemented as a higher order function as seen in figure 2. It takes three functions and a game state as input and returns an integer value. If optimal play would result in the currently active player winning the integer is positive, negative if the active player has no way to prevent a loss, and zero if optimal play would result in a draw. The first function just determines if the current position is a final state. The second function produces a list of all possible game states that can be reached from the current position in one move. The final function produces the score if the game is over. Using *map* and *maximum* we can avoid the loop seen in figure 1.

```
negamax    :: (a -> Bool) -> (a -> [a]) -> (a -> Int) -> a -> Int
egamax over moves eval game
           | over game = eval game
           | otherwise = maximum (map
                           (negate . (negamax over moves eval))
                           (moves game))
```

Figure 2: Haskell implementation of negamax search

The negamax implementation was incorporated into a module that other games can then include. This allows anything that implements the required set of functions to use the same negamax search. The implementation I use could likely be simplified further by removing the check for final state and using a Maybe value in one of the other functions. Implementing a type class that restricts the type of a might also be good idea, but has been ignored at present.

3 Tic-Tac-Toe

3.1 Description

Tic-Tac-Toe, also known as Noughts and Crosses or Xs and Os, is a simple paper and pencil game for two players, who take turns marking the spaces on a 3×3 grid. The player who succeeds in placing three marks in row is the winner. The game is mostly common with younger children as most people quickly realize that optimal play will always result in a draw.

Despite the simple nature of the game, Tic-Tac-Toe has 255,168 possible game states [Bottomly(2001)] if we ignore any possible symmetries. Which is not an insignificant search space. Taking symmetries into account would reduce the number of possible state to only 26,830 [Schaeffer(2002)], which is much smaller, but identifying symmetry would likely cost us significant computation time.

3.2 Haskell Implementation

To begin I created two data types to track the game state, shown in figure 3. The first keeps track of information regarding a single square. The square may contain an X or an O, or it may be empty, in which case I keep track of an integer associated with it's location. Because I will often need to compare or display squares I am deriving Eq and Show. The exact game state is represented by the TicTacToe data type, which stores the game board and the Symbol of the player on move. I don't specify the size of the board here, so I can easy expand this for use on larger game boards.

```
data Symbol    = Empty Int | X | O deriving (Eq, Show)
data TicTacToe = Game { board :: [[Symbol]], onMove :: Symbol }
```

Figure 3: Tic-Tac-Toe representation in Haskell

A game of Tic-Tac-Toe ends when either a player has won or when there are no available moves. The functions shown in figure 4 check if the current state is a final state. The gameOver and tieGame functions are self explanatory, but the wonGame function is a little more interesting. It creates a list of winning symbols. In reality this list rarely contains more than one symbol, and when it does, all symbols in the list are the same. However, there are other variants of Tic-Tac-Toe that I might want to represent, so I made the function a little more powerful than required. The function checks to see if there is a row column or diagonal with all elements the same. The rows and columns are easy to check, just using the original board for the rows and the transpose for the columns. The diagonals are a little more interesting and are checked just using an explicit list comprehension. The downside of this implementation is it only works on a 3×3 board. A larger board would require extensive changes to the current implementation of the diag function.

```
gameOver  :: TicTacToe -> Bool
gameOver g = (not $ null (wonGame g)) || tieGame g

wonGame  :: TicTacToe -> [Symbol]
wonGame g = [x | [x,y,z] <- lines, x == y && y == z]
            where diag [[a,_,b],
                        [_,c,_],
                        [d,_,e]] = [[a,c,e],[b,c,d]]
                  brd   = board g
                  lines = brd ++ diag brd ++ transpose brd

tieGame :: TicTacToe -> Bool
tieGame  = null . moves
```

Figure 4: Checking for game completion.

To create the list of possible game states that can be reach from the current position on one move I created the functions shown in figure 5. The moves function just creates a list of all squares that aren't currently an X or an O. I then use that list to create a new list of game states where the active player has marked each of the available squares.

Scoring a game of Tic-Tac-Toe was quite simple and the function I built can be seen in figure 6. A tie game is worth zero points and a winning position is worth three points to whoever has won. The

```
applyMoves  :: TicTacToe -> [TicTacToe]
applyMoves g | gameOver g = []
             | otherwise  =[makeMove g s | s <- moves g]

moves  :: TicTacToe -> [Symbol]
moves g = [x | x <- concat (board g), x /= X, x /= O]

makeMove    :: TicTacToe -> Symbol -> TicTacToe
makeMove g s = Game (map (map place) (board g)) p
               where p = switchPlayer (onMove g)
                     place b | b == s    = onMove g
                             | otherwise = b
```

Figure 5: Creates a list of all possible game states that can be reached in one move.

one point result isn't actually used anywhere in my program, but I wanted an option for including limiting the search to a specified depth which would require a score for all possible results.

```
gameEval  :: TicTacToe -> Int
gameEval g | tieGame g               =  0
           | null winner             =  1
           | head winner == onMove g =  3
           | otherwise               = -3
             where winner = wonGame g
```

Figure 6: Tic-Tac-Toe game evaluation

As a final piece I turned the ascii art program we built for assignment three into a module and added code for displaying the board in a simple style. The code can be seen in figure 7. The main reason for including this was to verify everything is working correctly, as I can now create boards in several different states where I know what the outcome should be and verify that my results match. The implementation just takes advantage of the align function written previously to assemble the individual squares into the appropriate grid.

3.3 Results

On Tic-Tac-Toe my results were exactly what was expected. It explored the complete game tree in about twenty seconds when run from the interpreter on my home computer, and returned the expected result in all cases. The starting board position, displayed using the method shown in figure 7, and the result from my negamax function are shown in figure 8, figure 9, and figure 10.

I was really quite pleased with the overall end result of the Tic-Tac-Toe program. The speed was reasonable, and might be improved by adding a main function so that I can compile the code before running. It also successfully solves some of the more obscure game states that humans often get incorrect. Some speed increases would be needed if I wanted to include this in an interactive Tic-Tac-Toe game, but not an excessive amount. This code was the first I finished and I was excited to move onto trying to solve a more complicated game.

```
picGame       :: TicTacToe -> Pic
picGame g      = picBoard (board g)

picBoard      :: [[Symbol]] -> Pic
picBoard board = align center (map (align middle) b)
                 where b = map (map picSquare) board

picSquare     :: Symbol -> Pic
picSquare x    = align center [tope, align middle [edge, item,edge], tope]
                 where edge = text ["|","|","|"]
                       tope = string "+-----+"
                       item = string ("  " ++ (r x) ++ "  ")
                       r (Empty n) = show n
                       r a         = show x
```

Figure 7: Ascii art display of the Tic-Tac-Toe board.

```
*Main> render (picGame initialState)
+-----++-----++-----+
|     ||     ||     |
|  1  ||  2  ||  3  |
|     ||     ||     |
+-----++-----++-----+
+-----++-----++-----+
|     ||     ||     |
|  4  ||  5  ||  6  |
|     ||     ||     |
+-----++-----++-----+
+-----++-----++-----+
|     ||     ||     |
|  7  ||  8  ||  9  |
|     ||     ||     |
+-----++-----++-----+
*Main> negamax initialState
0
```

Figure 8: Negamax starting with the empty board. Expected result: 0

```
*Main> render (picGame testState1)
+-----++-----++-----+
|     ||     ||     |
|  1  ||  O  ||  3  |
|     ||     ||     |
+-----++-----++-----+
+-----++-----++-----+
|     ||     ||     |
|  4  ||  X  ||  6  |
|     ||     ||     |
+-----++-----++-----+
+-----++-----++-----+
|     ||     ||     |
|  7  ||  8  ||  9  |
|     ||     ||     |
+-----++-----++-----+
*Main> negamax testState1
3
```

Figure 9: Negamax starting with X having the advantage. Expected result: 3

```
*Main> render (picGame testState2)
+-----++-----++-----+
|     ||     ||     |
|  O  ||  X  ||  3  |
|     ||     ||     |
+-----++-----++-----+
+-----++-----++-----+
|     ||     ||     |
|  4  ||  O  ||  6  |
|     ||     ||     |
+-----++-----++-----+
+-----++-----++-----+
|     ||     ||     |
|  7  ||  X  ||  9  |
|     ||     ||     |
+-----++-----++-----+
*Main> negamax testState2
-3
```

Figure 10: Negamax starting with O having the advantage. Expected result: -3

4 Fox and Hounds

4.1 Description

The game Fox and Hounds has also been known as Wolf and Sheep, Hounds and Hares, or Devil and Tailors. It is played on an 8×8 checkers or chess board and is somewhat unusual in that the two teams are mismatched. As in checkers, only the dark squares are used and pieces only move diagonally. The Hounds player begins with four pieces starting at their board edge and may move diagonally only forwards. The Fox player starts with a single piece and may move diagonally forwards and backwards. Unlike checkers no jumps or captures may be made. A typical starting position is shown in figure 11. The goal of the Hounds is to corner the Fox in a position where it can no longer move. The goal of the Fox is to reach the starting position of one of the Hounds.

Fox and Hounds was solved in 1982 [Elwin Berlekamp and Guy(1982)] and was found to be a Hounds victory when played perfectly. The game was not solved by exploring the game tree, instead Berlekamp, Conway and Guy devised a winning strategy that could respond to any possible move by the Fox. Unlike Tic-Tac-Toe this game has an extremely large game tree that needs to be searched. This proved to be much more difficult than the solving Tic-Tac-Toe.

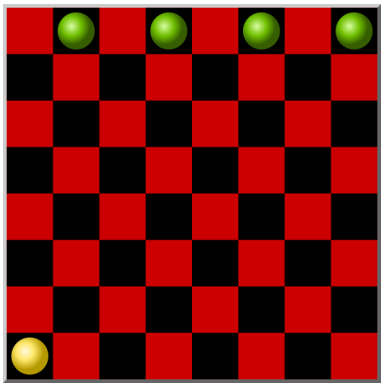

Figure 11: Typical starting positions for Fox and Hounds

4.2 Haskell Implementation

To track the game state of Fox and Hounds I creates two data types just like for Tic-Tac-Toe. The code is shown in figure 12. Each square on the board has a location, which is represented by a tuple of integers and is either empty or contains a piece. A complete game state is a list of squares, a player on move, and the positions of all of the pieces. Unlike in Tic-Tac-Toe I chose to represent the positions of all the pieces separately in addition to their location on the board. This allowed me to check for winning game states and available moves much more easily. Keeping track of the positions on the board allowed me to display the game more easily as well. I also store the allowed moves for each of the pieces since the two sides move differently.

My initial attempts at recognizing a final game state only took into account the two possible winning conditions. Recognizing when the fox has no available moves is easy enough, as is recognizing when the fox is in a winning position. The third possibility is that the hounds have no available moves, but the fox is not yet in a winning position. In traditional play, the hounds would lose a turn and the fox would get a chance to move again, but because I was already having issues with the size of the game tree I chose to represent this as a final state. My implementation is shown in figure 13 and takes heavy advantage null to recognize the various positions.

```
type Position   = (Int, Int)
type Moves      = [(Int, Int)]
data Symbol     = Empty | H | F deriving (Eq, Show)
data FoxHounds  = Game { board  :: [(Position,Symbol)],
                         onMove :: Symbol,
                         hounds :: [Position],
                         fox    :: Position}

foxMoves   :: Moves --Diagonal in any direction
foxMoves    = [(-1,-1),(1,-1),(-1,1),(1,1)]
houndMoves :: Moves --Diagonal forward only
houndMoves  = [(1,-1),(1,1)]
```

Figure 12: Fox and Hounds representation in Haskell

```
gameOver  :: FoxHounds -> Bool
gameOver g = null (moves (fox g) (hounds g) foxMoves) ||
             (and $ map null [moves s [fox g] houndMoves | s <- hounds g]) ||
             fst (fox g) == 1
```

Figure 13: Checking for game completion.

Unlike Tic-Tac-Toe, where creating the list of moves was quite simple, Fox and Hounds requires significant work to accurately represent all game states that can be reached in one move. The functions shown in figure 14 were created for this purpose. The moves function creates a list of all positions that can be reached from a given location, using one of the defined movement patterns shown in figure 12. All moves are checked to make sure they are still on the board and not currently occupied by another piece. The applyMove function generates a new game state with a piece moved from the starting position to the ending position. The final function, allMoves, just used a list comprehension and the previous two functions to create a list of all states that can be reach from the present position.

The function for scoring a game can be seen in figure 15 and is quite simple. If the fox has no available moves, then the hounds have won, and if the fox has reach the first row, the fox has won. All other cases are treated as a draw.

Once again I used the Ascii art program written previously to display the game board for testing purposes. The code can be seen in figure 16. This was also significantly more challenging than displaying the Tic-Tac-Toe board. Each square is responsible for displaying it's contents as well as it's top and left edges. Squares that are available to be moved through are shown empty and squares which are off limits are filled with '#'. Since the board in this case is stored as a single list instead of a list of lists, I needed a way to split the list every eight squares. I used splitEvery to do this, which I took from the *Data.List.Split* library. The code is included here for convenience.

```
allMoves  :: FoxHounds -> [FoxHounds]
allMoves g | onMove g == H = [applyMove g s e | s <- hounds g,
                                e <- moves s (fox g : hounds g) houndMoves]
           | otherwise     = [applyMove g (fox g) e |
                                e <- moves (fox g) (hounds g) foxMoves]

moves           :: Position -> [Position] -> Moves -> [(Int, Int)]
moves (a,b) o ms = [(a+x, b+y) | (x,y) <- ms,
                                 a+x > 0 && a+x < 9,
                                 b+y > 0 && b+y < 9,
                                 not ((a+x,b+y) `elem` o)]

applyMove  :: FoxHounds -> Position -> Position -> FoxHounds
applyMove g start end
            = Game (map (clear . place) (board g)) (switchPlayer $ onMove g) h f
              where clear a | fst a == start = (start,Empty)
                            | otherwise      = a
                    place a | fst a == end   = (end, onMove g)
                            | otherwise      = a
                    h | onMove g == H = end : [x | x <- hounds g, x /= start]
                      | otherwise     = hounds g
                    f | onMove g == F = end
                      | otherwise     = fox g
```

Figure 14: Creates a list of all possible game states that can be reached in one move.

```
scoreGame  :: FoxHounds -> Int
scoreGame g | null (moves (fox g) (hounds g) foxMoves) =  p
            | fst (fox g) == 1                         = -p
            | otherwise                                =  0
              where p | onMove g == H =  1
                      | otherwise     = -1
```

Figure 15: Fox and hounds game evaluation

```
picGame   :: FoxHounds -> Pic
picGame g  = picBoard (board g)

picSquare          :: (Position,Symbol) -> Pic
picSquare ((a,b),x) = align center [tope, align middle [edge, item]]
                      where edge = text ["|"]
                            tope = string "+---"
                            item = string (r x)
                            r Empty | even (a+b) = "   "
                                    | odd  (a+b) = "###"
                            r t                  = " " ++ show x ++ " "

picBoard   :: [(Position,Symbol)] -> Pic
picBoard ps = align center ([picRow x | x <- splitEvery 8 ps] ++
                            [string "+---+---+---+---+---+---+---+---+"])

picRow ps = align middle ((map picSquare ps) ++ [text ["+","|"]])

splitEvery     :: Int -> [a] -> [[a]]
splitEvery i ls = map (take i) (build (splitter ls))
                  where splitter [] _ n = n
                        splitter l c n  = l `c` splitter (drop i l) c n
                        build g         = g (:) []
```

Figure 16: Ascii art for the Fox and Hounds game.

4.3 Results

Here's where my success starts to wane a bit. My representation correctly solves simpler game states but is quite slow even on those instances. Solving the complete game tree would take longer than I have patience for. Even with 24 hours of continuous processing I still hadn't reached a final state. While everything certainly seems to work for simple cases, I can't explicitly guarantee that it works for solving the complete game. Figures 17 and 18 show the results from running on some simpler game states.

```
*Main> render (picGame tState1)
+---+---+---+---+---+---+---+---+
|   |###|   |###|   |###|   |###|
+---+---+---+---+---+---+---+---+
|###|   |###|   |###|   |###|   |
+---+---+---+---+---+---+---+---+
|   |###|   |###|   |###|   |###|
+---+---+---+---+---+---+---+---+
|###|   |###|   |###|   |###|   |
+---+---+---+---+---+---+---+---+
|   |###|   |###| H |###| H |###|
+---+---+---+---+---+---+---+---+
|###|   |###|   |###|   |###|   |
+---+---+---+---+---+---+---+---+
|   |###|   |###| H |###|   |###|
+---+---+---+---+---+---+---+---+
|###|   |###| H |###|   |###| F |
+---+---+---+---+---+---+---+---+
*Main> negamax tState1
-1
```

Figure 17: Solved game with Hounds having the advantage.

5 Future Work

There are several task I would have liked to have gotten to that I just didn't get a chance to finish. There are several improvements that can be made to the negamax search algorithm that would significantly improve the running time. I also started implementing a solver for a third game, but was unable to complete it in time for the report.

The game Quantum Tic-Tac-Toe is a variant on the tradition game where each player takes turns marking two squares which are superpositions of eachother. The game was developed by Alan Goff as a teaching aid for students trying to understand quantum superpositions [Allan Goff(2012)] and adds an interesting additional dimension to the game. The game was solved in January of 2011 by Ishizeki and Matsuura [Takumi Ishizeki(2011)]. I've previously written a solver for the game in Python, but I couldn't figure out an efficient way to find the quantum entanglements in Haskell. I should be able to get my solution working with another couple days worth of work, but at this point the representation isn't complete.

```
*Main> render (picGame tState2)
+---+---+---+---+---+---+---+---+
| H |###|   |###|   |###|   |###|
+---+---+---+---+---+---+---+---+
|###|   |###|   |###|   |###|   |
+---+---+---+---+---+---+---+---+
|   |###|   |###|   |###|   |###|
+---+---+---+---+---+---+---+---+
|###|   |###|   |###|   |###|   |
+---+---+---+---+---+---+---+---+
|   |###|   |###|   |###|   |###|
+---+---+---+---+---+---+---+---+
|###|   |###|   |###|   |###| F |
+---+---+---+---+---+---+---+---+
|   |###|   |###|   |###| H |###|
+---+---+---+---+---+---+---+---+
|###|   |###| H |###| H |###|   |
+---+---+---+---+---+---+---+---+
*Main> negamax tState2
1
```

Figure 18: Solved game with Fox having the advantage.

6 Conclusion

After using Haskell in this project for several weeks, it seems to adapt extremely well to the task of game solving. While not as fast as when implemented in C, it still ran in a respectable time for Tic-Tac-Toe. Dealing with the final game states was much easier to write in Haskell than it is in C. As was generating all possible game states from the current position. Not actually changing the game state and instead producing a new one makes rewinding to the previous position unneccessary, which made things much simpler. Having higher order functions also allowed for building only one version of negamax that worked for all games. An imperative language would require that I build a seperate negamax for each game I was working with.

The most difficult part of adjusting game solving to Haskell was getting used to creating a new game every time the position changes. I've been changing variables for years and not being able to was difficult initially. I really enjoyed having access to higher order functions in general however. Things like map were great for dealing with the lists of game states, and list comprehensions made building the set of possible moves much easier.

References

[Allan Goff(2012)] Joel Siegel Allan Goff, Dale Lehmann. Quantum tic-tac-toe, spooky coins and magic-envelopes, as metaphors for relativistic quantum physics. January 2012. URL www.paradignpuzzles.com/QT3%20&%20SCME%20Papers15.pdf.

[Bottomly(2001)] Henry Bottomly. How many tic-tac-toe games are possible?, February 2001. URL www.se16.info/hgb/tictactoe.htm.

[Elwin Berlekamp and Guy(1982)] John Conway Elwin Berlekamp and Richard Guy. *Winning Ways for your Mathematical Plays*. Academic Press, 1982.

[Jonathan Schaeffer(2007)] et al. Jonathan Schaeffer. Checkers is solved. *Science*, 317, July 2007.

[Schaeffer(2002)] Steve Schaeffer. Mathematical recreations, January 2002. URL www.mathrec.org/old/2002jan/solutions.html.

[Takumi Ishizeki(2011)] Akihiro Matsuura Takumi Ishizeki. Solving quantum tic-tac-toe. January 2011. URL rgconferences.com/proceed/acct11/pdf/258.pdf.

Appendix A Tic-Tac-Toe Complete Source Code

```
{-

Copyright (c) 2014 Daniel Leblanc

-}

import Negamax
import Data.List(transpose)
import AsciiPic

data Symbol    = Empty Int | X | O deriving (Eq, Show)
data TicTacToe = Game { board :: [[Symbol]], onMove :: Symbol }
                 deriving Show

solveGame :: TicTacToe -> Int
solveGame g = negamax gameOver applyMoves gameEval g

switchPlayer :: Symbol -> Symbol
switchPlayer X = O
switchPlayer O = X

applyMoves :: TicTacToe -> [TicTacToe]
applyMoves g | gameOver g = []
             | otherwise  =[makeMove g s | s <- moves g]

moves :: TicTacToe -> [Symbol]
moves g = [x | x <- concat (board g), x /= X, x /= O]

makeMove    :: TicTacToe -> Symbol -> TicTacToe
makeMove g s = Game (map (map place) (board g)) p
               where p = switchPlayer (onMove g)
                     place b | b == s    = onMove g
                             | otherwise = b

gameOver :: TicTacToe -> Bool
gameOver g = (not $ null (wonGame g)) || tieGame g

wonGame :: TicTacToe -> [Symbol]
wonGame g = [x | [x,y,z] <- lines, x == y && y == z]
            where diag [[a,_,b],
                        [_,c,_],
                        [d,_,e]] = [[a,c,e],[b,c,d]]
                  brd   = board g
                  lines = brd ++ diag brd ++ transpose brd

gameEval :: TicTacToe -> Int
gameEval g | tieGame g              = 0
           | null winner            = 1
           | head winner == onMove g = 3
```

```
            | otherwise                = -3
              where winner = wonGame g

tieGame :: TicTacToe -> Bool
tieGame  = null . moves

picGame        :: TicTacToe -> Pic
picGame g       = picBoard (board g)

picBoard       :: [[Symbol]] -> Pic
picBoard board = align center (map (align middle) b)
                 where b = map (map picSquare) board

picSquare      :: Symbol -> Pic
picSquare x     = align center [tope, align middle [edge, item,edge], tope]
                  where edge = text ["|","|","|"]
                        tope = string "+---+"
                        item = string ("  " ++ (r x) ++ "  ")
                        r (Empty n) = show n
                        r a         = show x

initialState :: TicTacToe
initialState  = Game [[Empty 1,Empty 2,Empty 3],
                      [Empty 4,Empty 5,Empty 6],
                      [Empty 7,Empty 8,Empty 9]] X
testState1 :: TicTacToe
testState1  = Game [[Empty 1,O       ,Empty 3],
                    [Empty 4,X       ,Empty 6],
                    [Empty 7,Empty 8,Empty 9]] X

testState2 :: TicTacToe
testState2  = Game [[      O, X,Empty 3],
                    [Empty 4, O,Empty 6],
                    [Empty 7, X,Empty 9]] X

testState3 :: TicTacToe
testState3  = Game [[      O, X,Empty 3],
                    [Empty 4, O,Empty 6],
                    [      X, X,Empty 9]] O

testState4 :: TicTacToe
testState4  = Game [[O,X,O],
                    [O,X,X],
                    [X,O,X]] O
```

Appendix B Fox and Hounds Complete Source Code

```
{-

Copyright (c) 2014 Daniel Leblanc

-}

import Negamax
import Data.List
import AsciiPic

type Position   = (Int, Int)
type Moves      = [(Int, Int)]
data Symbol     = Empty | H | F deriving (Eq, Show)
data FoxHounds  = Game { board  :: [(Position,Symbol)],
                         onMove :: Symbol,
                         hounds :: [Position],
                         fox    :: Position}

solveGame :: FoxHounds -> Int
solveGame g = negamax gameOver allMoves scoreGame g

foxMoves  :: Moves --Diagonal in any direction
foxMoves   = [(-1,-1),(1,-1),(-1,1),(1,1)]
houndMoves :: Moves --Diagonal forward only
houndMoves = [(1,-1),(1,1)]

moves           :: Position -> [Position] -> Moves -> [(Int, Int)]
moves (a,b) o ms = [(a+x, b+y) | (x,y) <- ms,
                                 a+x > 0 && a+x < 9,
                                 b+y > 0 && b+y < 9,
                                 not ((a+x,b+y) `elem` o)]

scoreGame :: FoxHounds -> Int
scoreGame g | null (moves (fox g) (hounds g) foxMoves) =  p
            | fst (fox g) == 1                         = -p
            | otherwise                                =  0
              where p | onMove g == H =  1
                      | otherwise     = -1

gameOver :: FoxHounds -> Bool
gameOver g = null (moves (fox g) (hounds g) foxMoves) ||
             (and $ map null [moves s [fox g] houndMoves | s <- hounds g]) ||
             fst (fox g) == 1

switchPlayer :: Symbol -> Symbol
switchPlayer p | p == H    = F
               | otherwise = H

applyMove :: FoxHounds -> Position -> Position -> FoxHounds
```

```
applyMove g start end
            = Game (map (clear . place) (board g)) (switchPlayer $ onMove g) h f
              where clear a | fst a == start = (start,Empty)
                            | otherwise      = a
                    place a | fst a == end   = (end, onMove g)
                            | otherwise      = a
                    h | onMove g == H = end : [x | x <- hounds g, x /= start]
                      | otherwise     = hounds g
                    f | onMove g == F = end
                      | otherwise     = fox g

allMoves  :: FoxHounds -> [FoxHounds]
allMoves g | onMove g == H = [applyMove g s e | s <- hounds g,
                                  e <- moves s (fox g : hounds g) houndMoves]
           | otherwise     = [applyMove g (fox g) e |
                                  e <- moves (fox g) (fox g : hounds g) foxMoves]

picGame   :: FoxHounds -> Pic
picGame g  = picBoard (board g)

picSquare           :: (Position,Symbol) -> Pic
picSquare ((a,b),x) = align center [tope, align middle [edge, item]]
                      where edge = text ["|"]
                            tope = string "+---"
                            item = string (r x)
                            r Empty | even (a+b) = "   "
                                    | odd  (a+b) = "###"
                            r t                  = " " ++ show x ++ " "

picBoard   :: [(Position,Symbol)] -> Pic
picBoard ps = align center ([picRow x | x <- splitEvery 8 ps] ++
                            [string "+---+---+---+---+---+---+---+---+"])

picRow ps = align middle ((map picSquare ps) ++ [text ["+","|"]])

splitEvery     :: Int -> [a] -> [[a]]
splitEvery i ls = map (take i) (build (splitter ls))
                  where splitter [] _ n = n
                        splitter l c n  = l `c` splitter (drop i l) c n
                        build g         = g (:) []

emptyBoard :: [(Position, Symbol)]
emptyBoard  = [((a,b),Empty) | a <- [1..8], b <- [1..8]]

startState :: FoxHounds
startState  = Game emptyBoard F [(1,1),(1,3),(1,5),(1,7)] (8,6)

fwin, hwin :: FoxHounds
fwin  = Game emptyBoard F [(2,6),(2,4),(3,5),(4,6)] (1,1)
hwin  = Game emptyBoard H [(8,6),(7,7),(6,8),(1,1)] (8,8)
```

```
fillPos            :: [(Position,Symbol)] -> [Position] -> Symbol -> [(Position,Symbol)]
fillPos b [] s      = b
fillPos b (p:ps) s = fillPos (map (place p s) b) ps s
                     where place p s (l,e) | l == p    = (l,s)
                                           | otherwise = (l,e)

tState1 :: FoxHounds
tState1 = Game (fillPos (fillPos emptyBoard h H) [f] F) F h f
          where h = [(5,7),(5,5),(7,5),(8,4)]
                f = (8,8)

tState2 :: FoxHounds
tState2  = Game (fillPos (fillPos emptyBoard h H) [f] F) F h f
          where h = [(1,1),(8,6),(7,7),(8,4)]
                f = (6,8)
```

Appendix C Negamax Complete Source Code

```
{-

Copyright (c) 2014 Daniel Leblanc

-}

module Negamax where


negamax     :: (a -> Bool) -> (a -> [a]) -> (a -> Int) -> a -> Int
negamax over moves eval game
            | over game = eval game
            | otherwise = maximum (map
                                   (negate . (negamax over moves eval))
                                   (moves game))
```