TTT LM: Visualization, Analysis, and Statistics

Visualization and analysis of tic-tac-toe games played by random agents, and statistical breakdown for a large sample of games.

Jacob Peck

CSC 466

Professor Craig Graci

Spring 2011

Listing of `ttt.l`:

```
;; tic-tac-toe machine learning

;; select
(defmethod select ((l list))
  (nth (random (length l)) l)
)

;; snoc
(defmethod snoc ((s symbol) (l list))
  (append l (list s))
)

;; play
(defmethod play (&aux play avail move)
  (setf play ())
  (setf avail '(nw n ne w c e sw s se))
  (dolist (player '(x o x o x o x o x))
    (cond
      ((eq player 'x)
        (setf move (select avail))
        (setf avail (remove move avail))
        (setf play (snoc move play))
      )
      ((eq player 'o)
        (setf move (select avail))
        (setf avail (remove move avail))
        (setf play (snoc move play))
      )
    )
  )
  play
)

;; helper class - board
(defclass board ()
  (
    (nw :accessor board-nw :initform 'NIL)
    (n :accessor board-n :initform 'NIL)
    (ne :accessor board-ne :initform 'NIL)
    (w :accessor board-w :initform 'NIL)
    (c :accessor board-c :initform 'NIL)
    (e :accessor board-e :initform 'NIL)
    (sw :accessor board-sw :initform 'NIL)
    (s :accessor board-s :initform 'NIL)
    (se :accessor board-se :initform 'NIL)
  )
)

(defmethod populate-board ((l list) &aux board turnnum player)
  (setf board (make-instance 'board))
  (setf turnnum 1)
  (setf player 'x)
  (dolist (element l)
```

```
    ; go through the list, assigning move values to the board positions
    (cond
      ((eq element 'nw)
        (setf (board-nw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'n)
        (setf (board-n board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'ne)
        (setf (board-ne board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'w)
        (setf (board-w board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'c)
        (setf (board-c board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'e)
        (setf (board-e board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'sw)
        (setf (board-sw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 's)
        (setf (board-s board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'se)
        (setf (board-se board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
    )
    (if (eq player 'x)
      (setf player 'o)
      ;else
      (progn (setf player 'x) (setf turnnum (+ 1 turnnum)))
    )
  )
  board
)

(defmethod analyze-board ((l list) &aux board value turnnum player)
  (setf board (make-instance 'board))
  (setf value 'd)
  (setf turnnum 1)
  (setf player 'x)
  (dolist (element l)
```

```
    ;; populate board, checking after each move for a win/loss/draw in terms
of player X
    (cond
      ((eq element 'nw)
        (setf (board-nw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'n)
        (setf (board-n board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'ne)
        (setf (board-ne board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'w)
        (setf (board-w board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'c)
        (setf (board-c board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'e)
        (setf (board-e board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'sw)
        (setf (board-sw board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 's)
        (setf (board-s board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
      ((eq element 'se)
        (setf (board-se board) (concatenate 'string (write-to-string player)
(write-to-string turnnum)))
      )
    )
    (if (eq player 'x)
      (setf player 'o)
      ;else
      (progn (setf player 'x) (setf turnnum (+ 1 turnnum)))
    )
    ; only change if game is currently a draw
    (if (eq value 'd) (setf value (check-for-win board)))
  )
  value
)

(defmethod check-for-win ((b board) &aux value)
  (setf value (check-for-win-row-1 b))
  (if (eq value 'd) (setf value (check-for-win-row-2 b)))
  (if (eq value 'd) (setf value (check-for-win-row-3 b)))
```

```
  (if (eq value 'd) (setf value (check-for-win-col-1 b)))
  (if (eq value 'd) (setf value (check-for-win-col-2 b)))
  (if (eq value 'd) (setf value (check-for-win-col-3 b)))
  (if (eq value 'd) (setf value (check-for-win-diag-1 b)))
  (if (eq value 'd) (setf value (check-for-win-diag-2 b)))
  value
)

(defmethod check-for-win-row-1 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-nw b) :test #'equalp)
      (find #\x (board-n b) :test #'equalp)
      (find #\x (board-ne b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-nw b) :test #'equalp)
      (find #\o (board-n b) :test #'equalp)
      (find #\o (board-ne b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-row-2 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-w b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-e b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-w b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-e b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-row-3 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-sw b) :test #'equalp)
```

```
      (find #\x (board-s b) :test #'equalp)
      (find #\x (board-se b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-sw b) :test #'equalp)
      (find #\o (board-s b) :test #'equalp)
      (find #\o (board-se b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-col-1 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-nw b) :test #'equalp)
      (find #\x (board-w b) :test #'equalp)
      (find #\x (board-sw b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-nw b) :test #'equalp)
      (find #\o (board-w b) :test #'equalp)
      (find #\o (board-sw b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-col-2 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-n b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-s b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-n b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-s b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
```

```
  value
)

(defmethod check-for-win-col-3 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-ne b) :test #'equalp)
      (find #\x (board-e b) :test #'equalp)
      (find #\x (board-se b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-ne b) :test #'equalp)
      (find #\o (board-e b) :test #'equalp)
      (find #\o (board-se b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-diag-1 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-nw b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-se b) :test #'equalp)
    )
  )
  (setf owin
    (and
      (find #\o (board-nw b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-se b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod check-for-win-diag-2 ((b board) &aux value xwin owin)
  (setf value 'd)
  (setf xwin
    (and
      (find #\x (board-ne b) :test #'equalp)
      (find #\x (board-c b) :test #'equalp)
      (find #\x (board-sw b) :test #'equalp)
    )
  )
  (setf owin
```

```
    (and
      (find #\o (board-ne b) :test #'equalp)
      (find #\o (board-c b) :test #'equalp)
      (find #\o (board-sw b) :test #'equalp)
    )
  )
  (if xwin (setf value 'w))
  (if owin (setf value 'l))
  value
)

(defmethod visualize ((b board))
  (format t "~%~A ~A ~A~%~A ~A ~A~%~A ~A ~A~%"
    (board-nw b) (board-n b) (board-ne b)
    (board-w b)  (board-c b) (board-e b)
    (board-sw b) (board-s b) (board-se b)
  )
  NIL
)

;; visualize
(defmethod visualize ((l list) &aux board)
  (setf board (populate-board l))
  (visualize board)
  NIL
)

;; analyze
(defmethod analyze ((l list))
  (analyze-board l)
)

;; demo
(defmethod demo (&aux p)
  (setf p (play))
  (format t "~A~%" p)
  (visualize p)
  (format t "~A~%" (analyze p))
  NIL
)

;; stats
(defmethod stats ((f function) (n number) (demo t) &aux w l d p result
results)
  (setf w 0 l 0 d 0)
  (dotimes (i n)
    (setf p (apply f ()))
    (if demo (format t "~A~%" p))
    (if demo (visualize p))
    (setf result (analyze p))
    (if demo (format t "~A~%" result))
    (cond
      ((eq result 'w) (setf w (+ 1 w)))
      ((eq result 'l) (setf l (+ 1 l)))
      ((eq result 'd) (setf d (+ 1 d)))
```

```
    )
  )
  (setf results (mapcar #'probability (list w l d) (list n n n)))
  (mapcar #'list '(w l d) results)
)

;; probability
(defmethod probability ((special integer) (total integer))
  (/ (float special) (float total))
)
```

Listing of `ttt-demo-and-stats.text`:

```
$ clisp
<...snip...>
[1]> (load "ttt.l")
;; Loading file ttt.l ...
;; Loaded file ttt.l
T
[2]> (demo)
(C E SW NW N SE S NE W)

O2 X3 O4
X5 X1 O1
X2 X4 O3
W
NIL
[3]> (demo)
(NE SW C N S SE NW E W)

X4 O2 X1
X5 X2 O4
O1 X3 O3
D
NIL
[4]> (demo)
(SE S N NW W E NE C SW)

O2 X2 X4
X3 O4 O3
X5 O1 X1
D
NIL
[5]> (demo)
(NE E C S NW N SW W SE)

X3 O3 X1
O4 X2 O1
X4 O2 X5
W
NIL
[6]> (demo)
(SE S NE E N C NW W SW)

X4 X3 X2
O4 O3 O2
X5 O1 X1
W
NIL
[7]> (demo)
(SE SW E NE C S N W NW)

X5 X4 O2
O4 X3 X2
O1 O3 X1
W
NIL
```

```
[8]> (demo)
(NE C N SW S E W SE NW)

X5 X2 X1
X4 O1 O3
O2 X3 O4
W
NIL
[9]> (demo)
(SE S SW E NW NE N C W)

X3 X4 O3
X5 O4 O2
X2 O1 X1
W
NIL
[10]> (demo)
(SW NW C W S E N SE NE)

O1 X4 X5
O2 X2 O3
X1 X3 O4
W
NIL
[11]> (demo)
(NE E N S W SE SW NW C)

O4 X2 X1
X3 X5 O1
X4 O2 O3
W
NIL
[12]> (demo)
(N SW W S E SE NW C NE)

X4 X1 X5
X2 O4 X3
O1 O2 O3
L
NIL
[13]> (demo)
(NW SW NE W E SE C S N)

X1 X5 X2
O2 X4 X3
O1 O4 O3
L
NIL
[14]> (demo)
(SW E NE NW W S N SE C)

O2 X4 X2
X3 X5 O1
X1 O3 O4
W
```

```
NIL
[15]> (demo)
(S NW E NE SW W SE N C)

O1 O4 O2
O3 X5 X2
X3 X1 X4
W
NIL
[16]> (demo)
(SE S W E C NW SW N NE)

O3 O4 X5
X2 X3 O2
X4 O1 X1
W
NIL
[17]> (stats #'play 10 t)
(S NE NW SW SE N C E W)

X2 O3 O1
X5 X4 O4
O2 X1 X3
W
(C S W NW SE N SW E NE)

O2 O3 X5
X2 X1 O4
X4 O1 X3
W
(SE E S C W NE NW N SW)

X4 O4 O3
X3 O2 O1
X5 X2 X1
W
(SE NW SW C W N S E NE)

O1 O3 X5
X3 O2 O4
X2 X4 X1
W
(SW NE N E SE C NW S W)

X4 X2 O1
X5 O3 O2
X1 O4 X3
W
(NW N SE W S C E NE SW)

X1 O1 O4
O2 O3 X4
X5 X3 X2
W
(S C W NW SW E NE N SE)
```

```
O2 O4 X4
X2 O1 O3
X3 X1 X5
W
(S W NW SW N E SE NE C)

X2 X3 O4
O1 X5 O3
O2 X1 X4
W
(N NE C NW S SE W E SW)

O2 X1 O1
X4 X2 O4
X5 X3 O3
W
(SW W NE S SE C E N NW)

X5 O4 X2
O1 O3 X4
X1 O2 X3
W
((W 1.0) (L 0.0) (D 0.0))
[18]> (stats #'play 10 t)
(N NE SW NW S SE W C E)

O2 X1 O1
X4 O4 X5
X2 X3 O3
L
(C S SW W NE NW E SE N)

O3 X5 X3
O2 X1 X4
X2 O1 O4
W
(E W NW SW C N S SE NE)

X2 O3 X5
O1 X3 X1
O2 X4 O4
D
(SE SW NE E N NW C W S)

O3 X3 X2
O4 X4 O2
O1 X5 X1
L
(NE C SE S NW SW N W E)

X3 X4 X1
O4 O1 X5
O3 O2 X2
W
```

```
(SW S NE NW N E W SE C)

O2 X3 X2
X4 X5 O3
X1 O1 O4
W
(S C E NE W NW N SE SW)

O3 X4 O2
X3 O1 X2
X5 X1 O4
L
(NE NW W C SW SE N E S)

O1 X4 X1
X2 O2 O4
X3 X5 O3
L
(E S NW C SE NE SW W N)

X2 X5 O3
O4 O2 X1
X4 O1 X3
D
(NE NW C W E S SE N SW)

O1 O4 X1
O2 X2 X3
X5 O3 X4
W
((W 0.4) (L 0.4) (D 0.2))
[19]> (stats #'play 10 t)
(E SE SW NE NW W S C N)

X3 X5 O2
O3 O4 X1
X2 X4 O1
D
(SE SW W E N NW NE S C)

O3 X3 X4
X2 X5 O2
O1 O4 X1
D
(S NE N C NW E SW SE W)

X3 X2 O1
X5 O2 O3
X4 X1 O4
L
(W SE S C NE NW SW E N)

O3 X5 X3
X1 O2 O4
X4 X2 O1
```

```
L
(NW NE S W E N SE C SW)

X1 O3 O1
O2 O4 X3
X5 X2 X4
W
(NE W SW N C E NW S SE)

X4 O2 X1
O1 X3 O3
X2 O4 X5
W
(SE W N C NE S SW E NW)

X5 X2 X3
O1 O2 O4
X4 O3 X1
L
(C NE W SE N E NW SW S)

X4 X3 O1
X2 X1 O3
O4 X5 O2
L
(N SW E S C SE NE NW W)

O4 X1 X4
X5 X3 X2
O1 O2 O3
L
(NE NW SE N C E SW S W)

O1 O2 X1
X5 X3 O3
X4 O4 X2
W
((W 0.3) (L 0.5) (D 0.2))
[20]> (stats #'play 7000 nil)
((W 0.584) (L 0.287) (D 0.129))
[21]> (stats #'play 10000 nil)
((W 0.5854) (L 0.2865) (D 0.1281))
[22]> (stats #'play 100000 nil)
((W 0.58266) (L 0.28951) (D 0.12783))
[23]> (bye)
Bye.
```

After running a battery of games, and running empirical analysis on increasingly large numbers of games, it appears that with two random agents playing a game of tic-tac-toe where X always goes first, player X will win the game approximately 58% of the time. X will lose (O will win) approximately 29% of the time, and the game will be drawn around 13% of the time. This leads me to believe that tic-tac-toe is an inherently broken game, but that makes sense as X both goes first *and* has one more turn than O. This double advantage conferred upon X leaves O hoping for a draw at best, should X be a skilled player.