

Desarrollo del Algoritmo Minimax con Poda Alfa-Beta como Estrategia para el Juego Othello en Common Lisp

Luis Fernando Espino Barrios
Instituto Tecnológico de Costa Rica
luisespino@yahoo.com
Noviembre 2009

Resumen: Este artículo trata acerca del desarrollo del algoritmo minimax junto con la extensión de poda alfa-beta como estrategia para lograr un adversario robusto en el juego de Othello en Commons Lisp. La finalidad es la puesta en práctica de algoritmos de inteligencia artificial y claramente la construcción de un adversario que busca la mejor jugada basada en futuras decisiones del juego a través de arboles utilizando el algoritmo de minimax. Además se utilizaron dos heurísticas, una que cuenta el número de fichas del jugador y del adversario, y otra que le da pesos a las diferentes posiciones en el tablero, especialmente premiando a las esquinas y castigando a las casillas antes de las esquinas.

Palabras clave: Algoritmos de búsqueda, algoritmos en common lisp, algoritmo minimax, cortes alfa beta, poda alfa-beta, resolución de problemas, reversi.

1. Introducción

La verificación de los algoritmos de inteligencia artificial converge en la prueba utilizando juegos, debido a una inexplicable fascinación. Un algoritmo importante es el del adversario, que plantea un problema donde intervienen dos o más jugadores, sean personas o máquinas, y el objetivo es adelantarse a posibles movimientos para seleccionar el movimiento propio.

Hay muchas razones para la utilización de los juegos en inteligencia artificial, en [1] se presenta un enfoque en que mencionan dos razones por las que los juegos aparecen para ser un buen dominio, en el cual se explora la inteligencia artificial:

- Los juegos proveen tareas estructuradas en las cuales es muy fácil medir sucesos o fallos.
- Los juegos no requieren de mucho conocimiento, y se pensaba que la solución podría estar en búsquedas directas de un punto de partida a un punto final de éxito.

Sin embargo, la segunda razón no es tan cierta en juegos complicados, debido a la creación excesiva de nodos, el alto factor de ramificación y también por los muchos movimientos posibles. Y aquí es donde entran otros algoritmos como el minimax y sus extensiones que mejoran de una manera considerable la funcionalidad de algoritmos de búsqueda como los tratados en [2] y [3].

En el resto del artículo se hace una introducción al algoritmo minimax y sus extensiones, una breve descripción del juego Othello, la implementación del juego con el algoritmo estudiado y los resultados.

2. Algoritmo minimax

2.1. Descripción del algoritmo

La idea principal del algoritmo [1] es iniciar en una posición y utilizar un generador de movimientos plausibles para generar un conjunto de posiciones de posible sucesores, se aplica una función de evaluación estática y se selecciona el mejor movimiento promoviendo el valor.

Hay ciertas definiciones mencionadas en [4] que son importantes:

- La poda, nos permite ignorar partes de un árbol de búsqueda que no marcan diferencia para obtener un movimiento óptimo.
- Las funciones de evaluación, son heurísticas que permiten aproximar la utilidad verdadera de estado sin hacer una búsqueda completa.
- Una estrategia óptima es una secuencia de movimientos que conducen a un estado objetivo, en el cual será ganador.

Por lo que, el algoritmo minimax calcula la decisión minimax del estado actual, utilizando el cálculo simple recurrente de los valores minimax de cada estado sucesor, directamente implementando ecuaciones de definición.

2.2. Ejemplo

Se supone el árbol de juego de la Figura 1.

El primer paso por la disposición del árbol es minimizar las hojas, para el caso del nodo b minimizará 3 y 7, el resultado es 3 por ser el menor.

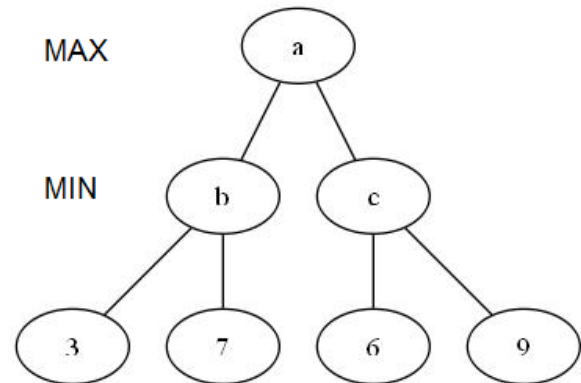


Figura 1: Árbol de juegos de ejemplo.

En el caso del nodo c minimizará 6 y 9, el resultado es 6, el resultado de ambas operaciones se promueve a los nodos padre, tal como se refleja en la Figura 2.

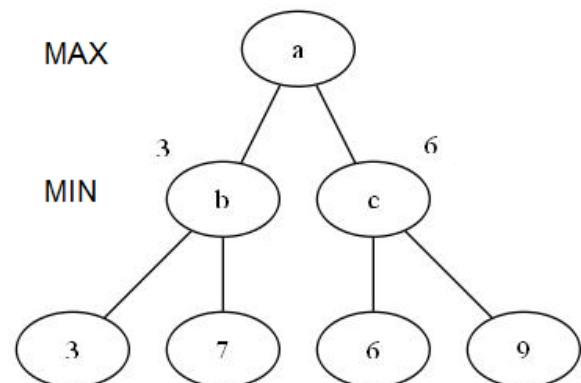


Figura 2: Minimización de valores del árbol ejemplo.

En el siguiente nivel o siguiente iteración se maximizarán los valores del segundo nivel que son los valores 3 y 6, el mayor entre ambos valores es 6 por lo que este valor se promoverá al nodo a, es decir la raíz, siendo el valor seleccionado por el algoritmo, tal como se muestra en la Figura 3 que se muestra a continuación.

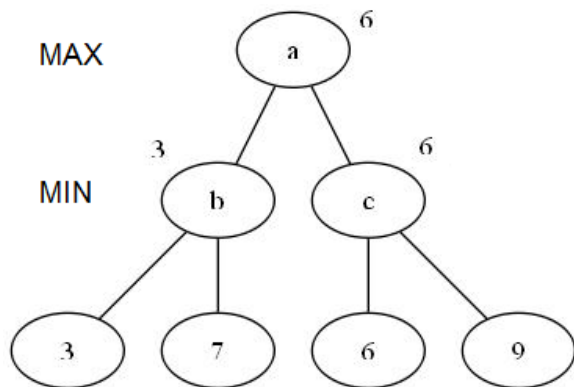


Figura 3: Maximización de valores del árbol ejemplo.

2.3. Poda alfa-beta

En juegos complejos el algoritmo minimax se vuelve prohibitivo con respecto al tiempo de respuesta y de espacio dependiendo del tamaño del nodo, volviéndose una estrategia ineficiente, debido a que sería similar a una búsqueda por niveles o por anchura, por lo que se debe hacer reducciones.

Estas reducciones se pueden hacer con la extensión del algoritmo con la estrategia de poda alfa-beta. En [1] se menciona que esta poda requiere el mantenimiento de dos variables, una representa el límite inferior en el valor que maximizando un nodo puede ser asignada, llamado alfa, y otro que representa el límite superior en el valor que minimizando un nodo puede ser asignada, llamado beta.

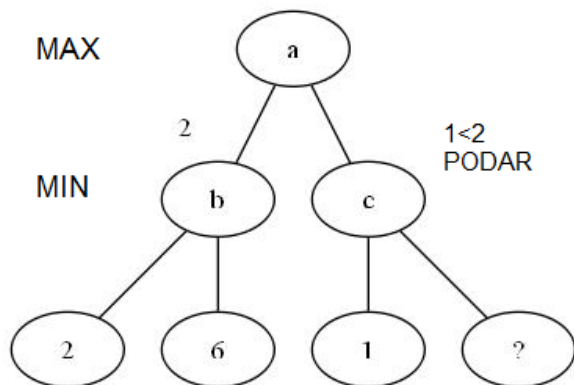


Figura 4: Poda alfa-beta del árbol ejemplo.

Por ejemplo se supone el árbol de la Figura 4, a la hora de minimizar el nodo b, se tiene los valores 2 y 6, por lo que el valor promovido es 2, luego en el nodo c se revisa el primer valor y si este es menor que el anterior valor promovido ya no es necesario seguir viendo los demás valores, puesto que esa rama ya tiene un valor menor que no servirá en el próximo nivel, debido a que se maximizará y por lo tanto no es necesario, por lo que esa rama deberá podarse.

3. El juego Othello

Es un juego de estrategia que se desarrolla en un tablero de 8x8. También llamado reversi. El objetivo es tener el mayor número de fichas sobre el tablero al final del juego.

Inicialmente el juego empieza de la siguiente manera:

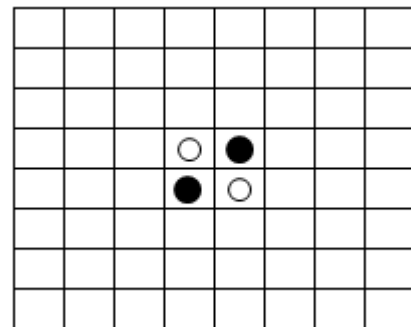


Figura 5: Disposición inicial del tablero.

Y las reglas del juego según la federación mundial de Othello son:¹

- Un movimiento consiste en colocar un ficha en una posición válida, en donde valida es que queden delimitadas en sus extremos por fichas del color del jugador con el fin de voltear las del adversario, ese movimiento se le llama flanquear.

¹ <http://www.worldothellofederation.com/>

- Las fichas negras mueven primero.
- Si en el turno no se puede flanquear se pierde el turno y el adversario vuelve a mover.
- El flanqueo puede ser a cualquier número de fichas en cualquier dirección.
- No se pueden pasar por alto las fichas propias a la hora de flanquear.
- En juegos personales, si un jugador no cambia todas las fichas a la hora de flanquear y si el adversario hace su movimiento ya no se pueden cambiar.
- El juego termina cuando ningún de los dos jugadores puede mover, contando las fichas actuales y el que tiene el mayor número es el ganador.
- Es posible que el juego termine antes de llenar las 64 casillas del tablero

Por ejemplo se supone la disposición inicial del juego, tal como se muestra en la Figura 6, según las reglas el turno es de quien tenga las fichas negras, las casillas posibles para colocar una ficha color negra son las marcadas con asterisco:

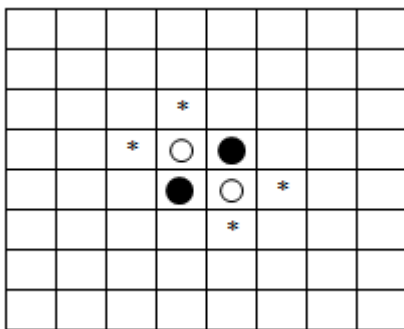


Figura 6: Posibilidades para colocar ficha negra en la disposición inicial del tablero.

Y el resultado de seleccionar la casilla más arriba sería como aparece en la Figura 7, que está a continuación.

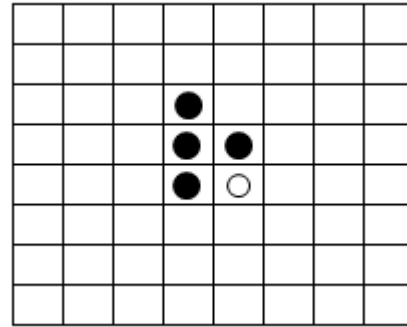


Figura 7: Nueva disposición del juego al seleccionar la casilla superior del tablero.

Othello es una marca registrada de Tsudaka Original, bajo licencia de Anjar, Co., copyright 1973, 1990 Pressman Toy.

4. Implementación

Con respecto a los recursos físicos para la implementación se utilizó una computadora tipo laptop, Dell Studio, procesador Intel Centrino con 4 GB de RAM. Se probó en Windows Vista y en Ubuntu 9.04, con las versiones de GNU Common Lisp (gcl) v.2.6.7. [5][6]

El tablero inicial del juego se especifica en un arreglo con valores de “0” para vacío y valores “b” para ficha negra y “w” para ficha blanca, de la siguiente manera:

```
(make-array '(8 8)
:initial-contents
'((0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0)
(0 0 0 w b 0 0 0)
(0 0 0 b w 0 0 0)
(0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0)))
```

La función principal para acabar el juego es la siguiente:

```
(defun game-over()
  (or
    (if (= (loop for x from 0 to 7
              sum (loop for y from 0 to 7
                    sum (if (eq
                            (aref state y x) 0)
                            1 0))) 0) t nil)
    (if (not (search-legal-moves
             state *turn*))
        (if (not (search-legal-moves
                 state (inverse *turn*)))
            t nil) nil))))
```

Está compuesta por dos condiciones que cualquiera hace verdadero el *game-over*; una es que el tablero se encuentre lleno, esto se hace recorriendo todo el tablero y verificando su contenido; y la otra es si ninguno de los dos jugadores tiene movimientos válidos, esto se hace preguntando si existen movimiento válidos para el turno actual y para el próximo turno.

Ejecutando e iniciando el juego en Common Lisp, se mostrará la siguiente disposición en el juego:

```

  0 1 2 3 4 5 6 7
  -----
0 | | | | | | | |
  -----
1 | | | | | | | |
  -----
2 | | | | | | | |
  -----
3 | | | |W|B| | | |
  -----
4 | | | |B|W| | | |
  -----
5 | | | | | | | |
  -----
6 | | | | | | | |
  -----
7 | | | | | | | |
  -----
BLACK Turn - Black coins: 2 - white
coins: 2
Your move:
```

Your move es un mensaje que está en espera de ingresar una lista con la posición de fila y columna, por ejemplo, si se desea colocar una ficha en la parte superior habría que ingresar la siguiente lista:

(2,3)

Las condiciones del juego se mapean a movimientos a todas direcciones, para diferenciar cada dirección se utilizaron los nombre de *north* para el norte, *south* para el sur, etc. Las 8 direcciones posibles tienen una manera similar de funcionar, con la diferencia que cambia ya sea el eje o cambian los límites del tablero. Por ejemplo, la función de búsqueda hacia el norte es:

```
(defun search-north (col row state turn)
  (when (> row 1)
    (loop for j from (- row 1)
          downto 0 do (cond
                      ((eq (aref state j col)
                           0) (return nil))
                      ((and
                        (eq
                          (aref state j col)
                          turn)
                        (= j (- row 1)))
                       (return nil))
                      ((and
                        (eq
                          (aref state j col)
                          turn)
                        (< j (- row 1)))
                       (return
                        (list col j)))
                      ((eq (aref state j col)
                           (inverse turn))
                       t))))))
```

Esta función recibe como parámetro la columna, la fila, el estado y el turno, como es un movimiento hacia arriba la fila mínima para ser legal tiene que ser mayor que uno, luego se realiza un barrido desde la fila-1 hasta 0 y se verifican cuatro condiciones:

- Si en medio del flanqueo hay un 0, es decir, si la casilla está vacía, no se puede realizar y devuelve nulo.
- Si en el flanqueo no hay fichas del adversario no se puede realizar el movimiento y devuelve nulo.
- Si por el contrario encuentra una ficha del mismo jugador quiere decir que el movimiento es válido y devuelve la posición para realizar el flanqueo.

- Y por último hay que verificar si hay una o más fichas del adversario en medio del flanqueo, si las hay simplemente devuelve verdadero para continuar la prueba.

Para conseguir que se ejecute la lógica del juego es necesario que hayan movimientos y que sean válidos, de lo contrario el turno se cambia al adversario, además, siempre hay que tener en cuenta la verificación de *game-over*.

```
(if (search-legal-moves state *turn*)
    (let ((moves (dfs state *turn* *level*)))
        (let ((pos
                (search-child
                 (minimax-ab moves) moves)))
            (m (second pos) (first pos))
              (setf *turn* (inverse *turn*)))
          (setf *turn* (inverse *turn*)))
```

Ya sea para el jugador siendo la computadora o siendo un usuario, la lógica solicita la verificación de movimientos legales, si es así, guarda esos movimientos para realizar luego el algoritmo minimax con la poda alfa-beta, ya sea que se tenga movimiento o no, la lógica debe cambiar el turno del jugador por el adversario.

En el caso de un jugador normal, el proceso se ejecuta cuando el turno corresponde al del jugador:

```
(let ((m (get-user-move)))
    (cond
     ((eq m 'exit) (return-from init
                               'EXIT-BY-USER))
     ((eq m 'illegal) (print
                       'ILLEGAL-MOVE))
     ((eq m 'move) (setf *turn*
                          (inverse *turn*))))
```

Prácticamente se ejecuta una función de movimiento, la cual, devuelve un valor de retorno, hay tres condiciones, ya sea si el usuario escribió la instrucción *'exit* para salir, o que la función devuelva que fue un movimiento ilegal con *'illegal-move* o que realmente se realizó el movimiento.

En este caso se cambia de turno para el adversario y así sucesivamente hasta que haya un *game-over*.

Para realizar el algoritmo minimax, se utilizó el algoritmo de búsqueda *depth-first* modificado y con límite de nivel, en el cual, se utiliza una estructura de lista para almacenar las posibles hojas o sucesores de determinado nivel:

```
(push (list (first state) *child*) leaves)
```

La estructura se llama *leaves* y cuando el nivel del árbol es el último se guarda a través de hacer un *push* en el estructura. Lo que se guarda es el estado que tiene la evaluación estática.

El algoritmo minimax con poda alfa-beta está basado en una versión previa escrita en Lisp por Mark Kantrowitz en el año 1990, se realizaron las adaptaciones necesarias y se realizó una extensión para integrar el algoritmo de búsqueda *depth-first* con el de minimax, debido a que se crean ciertas hojas con valor nulo y que en vez de manejar valores se manejan listas.

Esto porque se necesita recordar el camino de la solución. La extensión para verificar nulos fue:

```
(if (null (car tree)) *infinity*
```

Allí mismo se puede observar el (*car tree*) donde maneja el primer elemento de la lista que es enviada.

El algoritmo funciona como un *breadth-first* inverso de la siguiente manera:

```
(do* ((branches tree (cdr branches))
      (branch (car branches) (car
                               branches)))
```

Se revisa el primer elemento de la ramificación y se deja el resto, para evaluar si ya no hay más hojas:

```
(or (null branches)
    (>= alpha beta))
```

Asimismo, en esta instrucción se verifica si el valor de alfa es mayor o igual que el valor de beta, si es así, se realiza la poda.

Además se maneja un valor *mult*, que dependerá si se está realizando una operación *max* o *min*, intercambiando los valores para su evaluación y promoción.

```
(setq alpha
  (max alpha
    (- (minimax-ab branch (- beta)
                          (- alpha) (- mult))))
```

5. Heurísticas utilizadas y resultados

Se realizaron dos tipos de heurísticas:

- La heurística de conteo de fichas
- La heurística basada en esquinas

5.1. Heurística de conteo de fichas

Esta heurística calcula la diferencia de fichas del jugador con las fichas del adversario, esto nos proporciona nuestra función de evaluación:

- Si la evaluación es > 0 , el jugador va ganando.
- Si la evaluación es < 0 , el jugador va perdiendo
- Si la evaluación es $= 0$, el juego va empatado.

Esta heurística es bastante buena, la única desventaja que tiene es que si se juega con un jugador experto hay cierta probabilidad de perder.

Esto debido a la mala selección de las esquinas y de las casillas antes de las esquinas, porque esta heurística le da el mismo valor a todas las casillas, sin importar que las esquinas se consideran mejores.

5.2. Heurística basada en esquinas

Esta heurística le da pesos o valores a ciertas casillas consideradas más o menos importantes en el juego, aquí entra la estrategia del mismo, se probaron varias disposiciones y se llegó a la conclusión que la propuesta por Peter Norvig² es una de la mejores, la cual, se describe a continuación:

```
'(( 120 -20 20 5 5 20 -20 120 )
  (-20 -40 -5 -5 -5 -5 -40 -20 )
  ( 20 -5 15 3 3 15 -5 20 )
  ( 5 -5 3 3 3 3 -5 5 )
  ( 5 -5 3 3 3 3 -5 5 )
  ( 20 -5 15 3 3 15 -5 20 )
  (-20 -40 -5 -5 -5 -5 -40 -20 )
  ( 120 -20 20 5 5 20 -20 120 )))
```

Básicamente es darle más peso a las esquinas en los bordes (120), a las esquinas interiores (15), a los movimientos centrales poco valor (3), a los movimientos centrales del borde (5), y a las casillas antes de las esquinas de los bordes se castiga con un valor de (-20) cuando sea horizontal o vertical, y cuando esta casilla es diagonal se castiga más con (-40).

Luego esta heurística se calcula para las fichas del jugador y esa es la heurística que da un jugador artificial robusto.

Otra extensión a esta misma heurística es la resta de heurísticas entre adversarios, la cual, al principio se pensó que no tendría repercusión debido a que es un tabla de pesos y no de contraparte, pero al iniciar el juego se observó un mejor comportamiento.

² <http://norvig.com/paip/>

Por lo que, la mejor heurística es la basada en las esquinas, tomando la diferencia entre la evaluación del jugador y la del adversario para proporcionar la evaluación estática.

Por el tipo de algoritmo de búsqueda, por el algoritmo de minimax con poda alfa-beta y por el tipo de juego, existe un crecimiento exponencial de sucesores en ciertas etapas del juego y los niveles de exploración se vuelven prohibitivos respecto al tiempo de respuesta, por lo que se definió un nivel máximo de 3.

En la Figura 8 se muestran los datos de tiempo en total que se tardó la computadora en hacer sus movimientos:

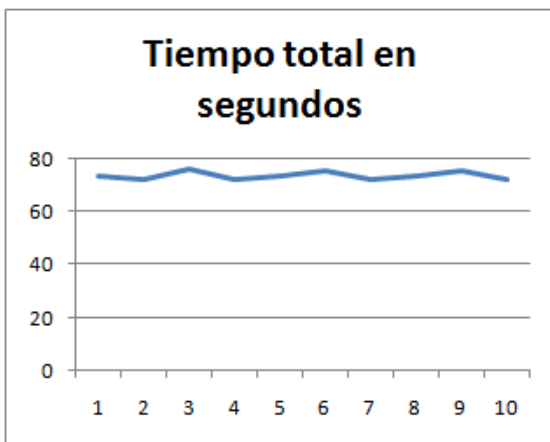


Figura 8: Tiempo total en segundos que tarda el adversario artificial durante un juego de 30 movimientos

Estos tiempos totales son de 10 ejecuciones distintas que cubren 30 movimientos, en promedio fueron 73.3 segundos en total, por lo que el promedio de tiempo que se tardó la computadora en hacer un movimiento en un tercer nivel del árbol minimax fue de:

2.44 segundos

El tiempo máximo en que se tardó para un movimiento fue de:

7 segundos

6. Conclusiones

Se implementó el algoritmo minimax con poda alfa-beta como estrategia para crear un adversario robusto en el juego de Othello en Common Lisp.

Las heurísticas utilizadas fueron dos: la diferencia de fichas de los jugadores para determinar quien va ganando, y la segunda es una heurística basada en pesos según las posiciones que se consideran mejores que otras.

Con base a las pruebas realizadas la mejor heurística fue la heurística de las esquinas que proporciona una visión adelantada más detallada del juego para volver robusto al jugador artificial.

7. Referencias

- [1] E. Rich and K. Knight, *Artificial Intelligence*, 2nd ed. United States of America: McGraw-Hill, Inc., 1991.
- [2] L. Espino, "Implementación de Algoritmos Básicos de Búsqueda para la Solución de Problemas en Common Lisp," Instituto Tecnológico de Costa Rica, 2009.
- [3] L. Espino, "Comparación de Heurísticas para la Solución del Problema 8-puzzle mediante el Algoritmo A* en Common Lisp," Instituto Tecnológico de Costa Rica, 2009.

- [4] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. United States of America: Pearson Education, Inc., 2003.
- [5] G. Steele, *Common Lisp: The Language*, 2nd ed. United States of America: Digital Equipment Corporation, 1990.
- [6] D. Friedman and M. Felleisen, *The Little Schemer*, 4th ed., M. I. o. Technology, Ed. United States of America, 1996.
- [7] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," in *IEEE Transactions of Systems Science and Cybernetics*, vol. SSC-4 No.2, United States of America, 1968, pp. 100-107.

Apéndice

```
(defparameter *turn* 'w)
(defparameter machine 'b)
(defparameter legal-moves '())
(defvar *infinity* 1000000)
(defvar *report-cutoffs* nil)
(defun p(board)
  (if board (show-board board) nil))

(defun show-board(board)
  (format t "~%~%~% 0 1 2 3 4 5 6 7")
  (format t "% -----")
  (loop for i from 0 to 7
    do (format t "%~%~%D |" i)
      (loop for j from 0 to 7
        do (format t "~A|" (if (eq (aref board i j) 0) " " (aref board
i j))))
      (format t "% -----")
      (format t "%~% ")
      (format t "%~%S Turn - Black coins: ~D - White coins: ~D" (if (eq
*turn* 'w) 'White 'Black) (show-coins 'B state) (show-coins 'W
state))
      (format t "%~%~%S" legal-moves))

(defun show-coins(coin board)
  (loop for i from 0 to 7
    sum (loop for j from 0 to 7
      sum (if (eq (aref board i j) coin) 1 0))))

(defun duplicate(board)
  (let ((new-state (make-array '(8 8))))
    (loop for row from 0 to 7
      do (loop for col from 0 to 7
        do (setf (aref new-state row col) (aref board row col) )))
    new-state))

(defun inverse(coin)
  (cond ((eq coin 'w) 'b) ((eq coin 'b) 'w)))
```

```
(defun search-legal-moves(state turn)
  (setf legal-moves '())
  (loop for x from 0 to 7
    do (loop for y from 0 to 7
      do (when (eq (aref state y x) 0) (try-legal-moves x y turn))))
  (setf legal-moves (successor-order legal-moves))
  (if legal-moves legal-moves nil))

(defun successor-order (queue)
  (sort queue (lambda (x y) (> (third x) (third y))))))

(defun try-legal-moves(col row turn)
  (if (or (search-north col row state turn)
    (search-south col row state turn)
    (search-east col row state turn)
    (search-west col row state turn)
    (search-neast col row state turn)
    (search-nwest col row state turn)
    (search-seast col row state turn)
    (search-swast col row state turn)
    nil) (push (list col row (eval-state col row state turn)) legal-
moves)))

(defun search-north (col row state turn)
  (when (> row 1)
    (loop for j from (- row 1) downto 0
      do (cond ((eq (aref state j col) 0) (return nil)) ; empty, fail
        ((and (eq (aref state j col) turn) (= j (- row 1))) (return
nil)); same turn near, fail
        ((and (eq (aref state j col) turn) (< j (- row 1))) (return
(list col j) )); same turn, succesful
        ((eq (aref state j col) (inverse turn) t))))); inverse coin,
continue

(defun search-south(col row state turn)
  (when (< row 6)
    (loop for j from (+ row 1) to 7 do (cond
      ((eq (aref state j col) 0) (return nil)) ; empty near, fail
      ((and (eq (aref state j col) turn) (= j (+ row 1))) (return
nil)); empty, succesful
      ((and (eq (aref state j col) turn) (> j (+ row 1))) (return
(list col j))); empty, succesful
      ((eq (aref state j col) (inverse turn) t))))); inverse coin,
continue

(defun search-east(col row state turn)
  (when (< col 6)
    (loop for i from (+ col 1) to 7 do (cond
      ((eq (aref state row i) 0) (return nil)) ; empty near,
fail
      ((and (eq (aref state row i) turn) (= i (+ col 1))) (return
nil)); empty, succesful
      ((and (eq (aref state row i) turn) (> i (+ col 1))) (return
(list i row))); empty, succesful
      ((eq (aref state row i) (inverse turn) t))))); inverse
coin, continue

(defun search-west(col row state turn)
  (when (> col 1)
    (loop for i from (- col 1) downto 0 do (cond
      ((eq (aref state row i) 0) (return nil)) ; empty near,
fail
      ((and (eq (aref state row i) turn) (= i (- col 1))) (return
nil)); empty, succesful
      ((and (eq (aref state row i) turn) (< i (- col 1))) (return
(list i row))); empty, succesful
```

```
      ((eq (aref state row i) (inverse turn) t)))) ; inverse coin, continue
```

```
(defun search-neast (col row state turn)
  (when (and (> row 1) (< col 6))
    (loop for j from (- row 1) downto 0
          for i from (+ col 1) to 7 do (cond
            ((eq (aref state j i) 0) (return nil)); empty near, fail
            ((and (eq (aref state j i) turn) (= i (+ col 1)) (= j (- row 1)))
              (return nil)); empty, succesful
            ((and (eq (aref state j i) turn) (> i (+ col 1)) (< j (- row 1)))
              (return (list i j))); empty, succesful
            ((eq (aref state j i) (inverse turn) t)))) ; inverse coin, continue
```

```
(defun search-nwest (col row state turn)
  (when (and (> row 1) (> col 1))
    (loop for j from (- row 1) downto 0
          for i from (- col 1) downto 0 do (cond
            ((eq (aref state j i) 0) (return nil)); empty near, fail
            ((and (eq (aref state j i) turn) (= i (- col 1)) (= j (- row 1)))
              (return nil)); empty, succesful
            ((and (eq (aref state j i) turn) (< i (- col 1)) (< j (- row 1)))
              (return (list i j))); empty, succesful
            ((eq (aref state j i) (inverse turn) t)))) ; inverse coin, continue
```

```
(defun search-seast (col row state turn)
  (when (and (< row 6) (< col 6))
    (loop for j from (+ row 1) to 7
          for i from (+ col 1) to 7 do (cond
            ((eq (aref state j i) 0) (return nil)); empty near, fail
            ((and (eq (aref state j i) turn) (= i (+ col 1)) (= j (+ row 1)))
              (return nil)); empty, succesful
            ((and (eq (aref state j i) turn) (> i (+ col 1)) (> j (+ row 1)))
              (return (list i j))); empty, succesful
            ((eq (aref state j i) (inverse turn) t)))) ; inverse coin, continue
```

```
(defun search-swest (col row state turn)
  (when (and (< row 6) (> col 1))
    (loop for j from (+ row 1) to 7
          for i from (- col 1) downto 0 do (cond
            ((eq (aref state j i) 0) (return nil)); empty near, fail
            ((and (eq (aref state j i) turn) (= i (- col 1)) (= j (+ row 1)))
              (return nil)); empty, succesful
            ((and (eq (aref state j i) turn) (< i (- col 1)) (> j (+ row 1)))
              (return (list i j))); empty, succesful
            ((eq (aref state j i) (inverse turn) t)))) ; inverse coin, continue
```

```
(defun move-north(initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for j from (second initial-pos) downto (second final-pos)
          do (setf (aref board-tmp j (first initial-pos)) turn) finally
              (return board-tmp))))
```

```
(defun move-south(initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for j from (second initial-pos) to (second final-pos)
          do (setf (aref board-tmp j (first initial-pos)) turn) finally
              (return board-tmp))))
```

```
(defun move-east(initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for i from (first initial-pos) to (first final-pos)
```

```
      do (setf (aref board-tmp (second initial-pos) i) turn) finally
          (return board-tmp))))
```

```
(defun move-west(initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for i from (first initial-pos) downto (first final-pos)
          do (setf (aref board-tmp (second initial-pos) i) turn) finally
              (return board-tmp))))
```

```
(defun move-neast (initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for i from (first initial-pos) to (first final-pos)
          for j from (second initial-pos) downto (second final-pos)
          do (setf (aref board-tmp j i) turn) finally (return board-tmp))))
```

```
(defun move-nwest (initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for i from (first initial-pos) downto (first final-pos)
          for j from (second initial-pos) downto (second final-pos)
          do (setf (aref board-tmp j i) turn) finally (return board-tmp))))
```

```
(defun move-seast (initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for i from (first initial-pos) to (first final-pos)
          for j from (second initial-pos) to (second final-pos)
          do (setf (aref board-tmp j i) turn) finally (return board-tmp))))
```

```
(defun move-swest (initial-pos final-pos board turn)
  (let ((board-tmp (duplicate board)))
    (loop for i from (first initial-pos) downto (first final-pos)
          for j from (second initial-pos) to (second final-pos)
          do (setf (aref board-tmp j i) turn) finally (return board-tmp))))
```

```
(defun search-choice (choice moves-list)
  (cond ((null moves-list) nil)
        ((eq choice (car (car moves-list))) (list (second (car moves-list))
            (third (car moves-list))))
        (t (search-choice choice (cdr moves-list)))))
```

```
(defun game-over()
  (or (if (= (loop for x from 0 to 7
                  sum (loop for y from 0 to 7
                          sum (if (eq (aref state y x) 0) 1 0))) 0) t nil)
      (if (not (search-legal-moves state *turn*))
          (if (not (search-legal-moves state (inverse *turn*)))
              t nil) nil))))
```

```
(defun get-user-move()
  (format t "~%Your move: ")
  (let ((move (read))) (cond
    ((eq move 'exit) 'exit)
    ((listp move)
     (if (and (numberp (first move)) (numberp (second move)))
         (if (m (first move) (second move)) 'move 'illegal)))
    ((get-user-move))))))
```

```
(defun init(coin level)
  (cond ((eq level 1) (and (setf *heuristic* corner-heuristic) (setf *level* 1)))
        ((eq level 2) (and (setf *heuristic* static-heuristic) (setf *level* 1))))
```

```

    ((eq level 3) (and (setf *heuristic* static-heuristic) (setf *level*
3)))
    ((eq level 4) (and (setf *heuristic* corner-heuristic) (setf
*level* 3))))

```

```

(defparameter state (make-array '(8 8) :initial-contents
'((0 0 0 0 0 0 0 0) (0 0 0 0 0 0 0 0) (0 0 0 0 0 0 0 0)
(0 0 0 b w 0 0 0 0)(0 0 0 w b 0 0 0 0) (0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0) (0 0 0 0 0 0 0 0))))
(setf *turn* coin) (search-legal-moves state *turn*)
(p state)
(loop until (game-over) do
  (and (if (eq machine *turn*)
    (if (search-legal-moves state *turn*)
      (let ((moves (dfs state *turn* *level*)))
        (let ((pos (search-child (minimax-ab moves) moves)))
          (m (second pos) (first pos))
          (setf *turn* (inverse *turn*)))
        (setf *turn* (inverse *turn*)))
      (if (search-legal-moves state *turn*)
        (let ((m (get-user-move)))
          (cond
            ((eq m 'exit) (return-from init 'EXIT-BY-USER))
            ((eq m 'illegal) (print 'ILEGAL-MOVE))
            ((eq m 'move) (setf *turn* (inverse *turn*)))
            (setf *turn* (inverse *turn*)))
          (if (search-legal-moves state *turn*)
            (p state) (p state))))))'GAME-OVER

```

```

(defun temp-move(row col state turn)
  (let ((n (search-north col row state turn))
        (s (search-south col row state turn))
        (e (search-east col row state turn))
        (w (search-west col row state turn))
        (ne (search-neast col row state turn))
        (nw (search-nwest col row state turn))
        (se (search-seast col row state turn))
        (sw (search-sw west col row state turn))
        (temp-state state))

```

```

    (if n (setf temp-state (move-north (list col row) n temp-state
turn)))
    (if s (setf temp-state (move-south (list col row) s temp-state
turn)))
    (if e (setf temp-state (move-east (list col row) e temp-state
turn)))
    (if w (setf temp-state (move-west (list col row) w temp-state
turn)))
    (if ne (setf temp-state (move-neast (list col row) ne temp-state
turn)))
    (if nw (setf temp-state (move-nwest (list col row) nw temp-state
turn)))
    (if se (setf temp-state (move-seast (list col row) se temp-state
turn)))
    (if sw (setf temp-state (move-sw west (list col row) sw temp-state
turn)))

```

```

    (if (or n s e w ne nw se sw) temp-state nil)))

```

```

(defun m(row col)
  (let ((n (search-north col row state *turn*))
        (s (search-south col row state *turn*))
        (e (search-east col row state *turn*))
        (w (search-west col row state *turn*))
        (ne (search-neast col row state *turn*))

```

```

    (nw (search-nwest col row state *turn*))
    (se (search-seast col row state *turn*))
    (sw (search-sw west col row state *turn*)))

```

```

    (if n (setf state (move-north (list col row) n state *turn*)))
    (if s (setf state (move-south (list col row) s state *turn*)))
    (if e (setf state (move-east (list col row) e state *turn*)))
    (if w (setf state (move-west (list col row) w state *turn*)))
    (if ne (setf state (move-neast (list col row) ne state *turn*)))
    (if nw (setf state (move-nwest (list col row) nw state *turn*)))
    (if se (setf state (move-seast (list col row) se state *turn*)))
    (if sw (setf state (move-sw west (list col row) sw state *turn*)))
    (if (or n s e w ne nw se sw) t nil)))

```

```

(defun suc(state turn)
  (setf mab-legal-moves '())
  (loop for x from 0 to 7
    do (loop for y from 0 to 7
      do (when (eq (aref state y x) 0) (suc-moves state x y turn))))
  (setf mab-legal-moves (ordenar mab-legal-moves turn))
  (if mab-legal-moves mab-legal-moves nil))

```

```

(defun ordenar (queue turn)
  (sort queue (lambda (x y) (> (car x) (car y))))))

```

```

(defun suc-moves(state col row turn)
  (if (or
    (search-north col row state turn)
    (search-south col row state turn)
    (search-east col row state turn)
    (search-west col row state turn)
    (search-neast col row state turn)
    (search-nwest col row state turn)
    (search-seast col row state turn)
    (search-sw west col row state turn)
    nil) (push (list (static-eval-state col row state turn) col row
(temp-move row col state turn)) mab-legal-moves)))

```

```

(defun search-child(x queue)
  ;(print (car queue))
  (cond
    ((null queue) nil)
    ((numberp (car queue)) (if (= (car queue) x) (second queue)
nil))
    ((search-child x (car queue))
     ((search-child x (cdr queue))))))

```

```

(defun dfs (state turn depth)
  (setf leaves '())
  (setf *depth* depth)
  (setf *child* nil)
  (dfs1 (list 0 0 0 state) (inverse turn) depth)

  ;(setf leaves (remove-nil leaves))
  ;(print leaves)
  (if leaves (when (and (null (car leaves)) (null (second leaves)))
(setf leaves nil)))
  (if leaves (when (and (null (car (car leaves))) (null (second (car
leaves)))) (setf leaves nil)))
  (setf leaves (reverse leaves))
  (if (null (car leaves)) (setf leaves (rest leaves)))
  (if (null (car leaves)) (setf leaves (rest leaves)))
  (if leaves
    leaves ;(rest (reverse leaves))
    (list (list (third (car legal-moves))
                (list (first (car legal-moves))

```

```

(second (car legal-moves))))))

(defun dfs1 (state turn depth)
  ;(setf eval (eval-state (first state) (second state) (third state)))
  ;(print eval)
  ;(print state)
  (setf turn (inverse turn))
  (if (= (- *depth* depth) 1) (setf *child* (list (second state) (third
state))))
  (if (zerop depth)
    ;(and
      ;(print (first state))
      ;(setf eval (eval-state (first state) (second state) (third state)
(inverse turn)))
    (cond
      ((= *depth* 1) (push (list (first state) *child*) leaves))
      ((= *depth* 2) (push (list (first state) *child*) (car leaves)))
      ((= *depth* 3) (push (list (first state) *child*) (car (car
leaves)))));)
    (cond
      ((= (- *depth* depth) 0) (setf leaves (list leaves)))
      ((= (- *depth* depth) 1) (setf leaves (cons (list) leaves)))
      ((= (- *depth* depth) 2) (push (list) (car leaves))))))
    (cond
      ((zerop depth) nil)
      ((let ((children (suc (fourth state) turn))) ;third state =
board
          (let ((result (dfs2 children turn (- depth 1))))
              (and result (cons state result)))))))

(defun dfs2 (states turn depth)
  (cond
    ((null states) nil)
    ((dfs1 (car states) turn depth))
    ((dfs2 (cdr states) turn depth))))

(defun minimax-ab (tree &optional (alpha (- *infinity*)) (beta
*infinity*)(mult 1))
  (if (null (car tree))
    *infinity*
    (if (numberp (car tree))
      (* mult (car tree))
      (do* ((branches tree (cdr branches))
            (branch (car branches) (car branches)))
          ((or (null branches) ; no more branches
              (>= alpha beta)) ; cutoff
            (cond (branches
                  (when *report-cutoffs*
                    (format t "~&Cutting off branches ~{~& ~A~}"
branches))
                  beta) ; if a >= b, return beta
                (t alpha))) ; if last node, return max of the nodes
          ;(print (second branch))
          (setq alpha
                (max alpha
                      (- (minimax-ab branch (- beta) (- alpha) (- mult))))))))))

(defun eval-state(col row state turn)
  (let ((n (search-north col row state turn))
        (s (search-south col row state turn))
        (e (search-east col row state turn))
        (w (search-west col row state turn))
        (ne (search-neast col row state turn))
        (nw (search-nwest col row state turn))
        (se (search-seast col row state turn))
        (sw (search-swast col row state turn)) (temp state))

```

```

(if n (setf temp (move-north (list col row ) n temp turn)))
(if s (setf temp (move-south (list col row ) s temp turn)))
(if e (setf temp (move-east (list col row ) e temp turn)))
(if w (setf temp (move-west (list col row ) w temp turn)))
(if ne (setf temp (move-neast (list col row ) ne temp turn)))
(if nw (setf temp (move-nwest (list col row ) nw temp turn)))
(if se (setf temp (move-seast (list col row ) se temp turn)))
(if sw (setf temp (move-swast (list col row ) sw temp turn)))

(loop for x from 0 to 7
      sum (loop for y from 0 to 7
                sum (if (eq (aref temp y x) turn)
                        (aref corner-heuristic y x) 0) ))))

(defun static-eval-state(col row state turn)
  (let ((n (search-north col row state turn))
        (s (search-south col row state turn))
        (e (search-east col row state turn))
        (w (search-west col row state turn))
        (ne (search-neast col row state turn))
        (nw (search-nwest col row state turn))
        (se (search-seast col row state turn))
        (sw (search-swast col row state turn))
        (temp state))

    (if n (setf temp (move-north (list col row ) n temp turn)))
    (if s (setf temp (move-south (list col row ) s temp turn)))
    (if e (setf temp (move-east (list col row ) e temp turn)))
    (if w (setf temp (move-west (list col row ) w temp turn)))
    (if ne (setf temp (move-neast (list col row ) ne temp turn)))
    (if nw (setf temp (move-nwest (list col row ) nw temp turn)))
    (if se (setf temp (move-seast (list col row ) se temp turn)))
    (if sw (setf temp (move-swast (list col row ) sw temp turn)))

    (loop for x from 0 to 7
          sum (loop for y from 0 to 7
                    sum (if (eq (aref temp y x) turn)
                            (aref *heuristic* y x)
                            (if (eq (aref temp y x) (inverse turn))
                                (- (aref *heuristic* y x)
                                    0))))))

(defparameter corner-heuristic ; suggested by peter norvig
  (make-array '(8 8)
              :initial-contents
              '(( 120 -20 20 5 5 20 -20 120 )
                (-20 -40 -5 -5 -5 -5 -40 -20 )
                ( 20 -5 15 3 3 15 -5 20 )
                ( 5 -5 3 3 3 3 -5 5 )
                ( 5 -5 3 3 3 3 -5 5 )
                ( 20 -5 15 3 3 15 -5 20 )
                (-20 -40 -5 -5 -5 -5 -40 -20 )
                ( 120 -20 20 5 5 20 -20 120 )))

(defparameter static-heuristic
  (make-array '(8 8)
              :initial-contents
              '(( 1 0 1 1 1 1 0 1 ) ( 0 -3 1 1 1 1 1 -3 0 )
                ( 1 1 1 1 1 1 1 1 ) ( 1 1 1 1 1 1 1 1 )
                ( 1 1 1 1 1 1 1 1 ) ( 1 1 1 1 1 1 1 1 )
                ( 0 -3 1 1 1 1 1 -3 0 ) ( 1 0 1 1 1 1 0 1 )))

```