Tema 5: Técnicas heurísticas en juegos

José A. Alonso Jiménez
Francisco J. Martín Mateos

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Sevilla

Juegos: Motivaciones para su estudio

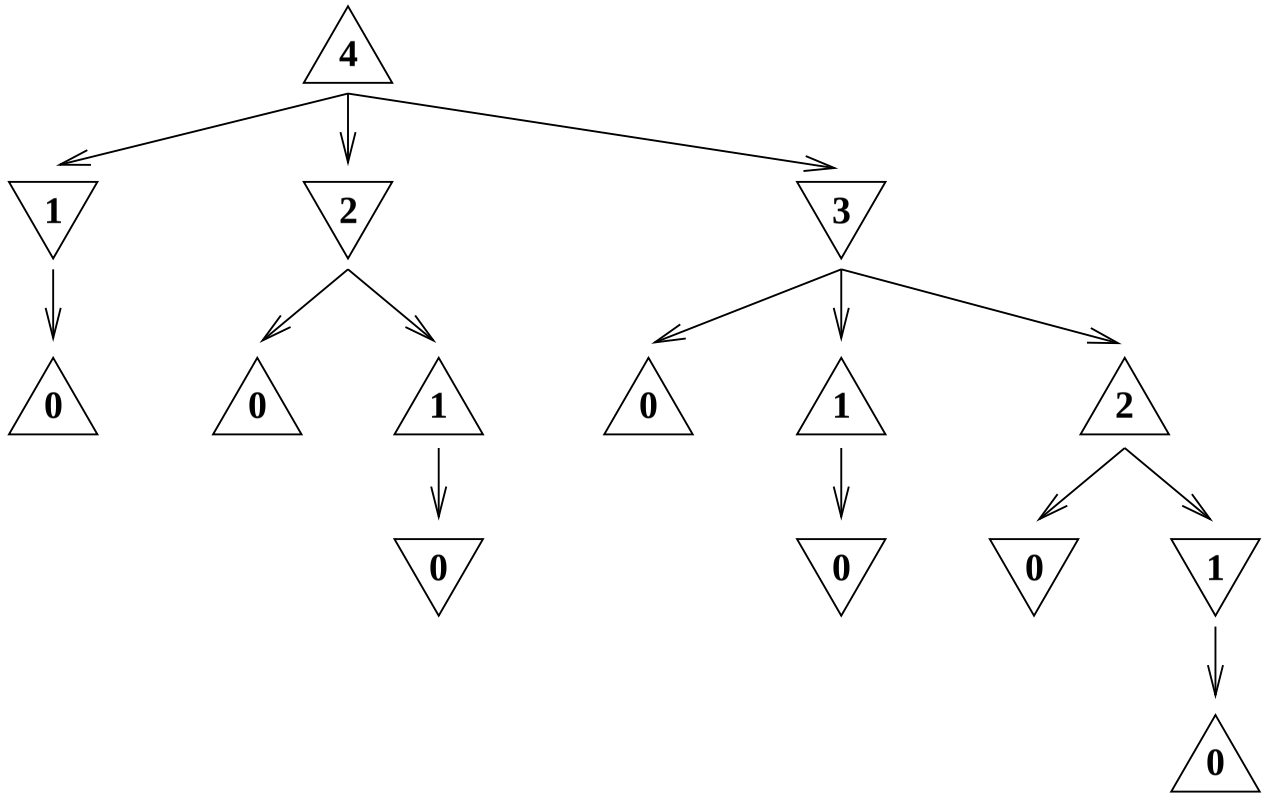
- **Crear programas de ordenador para jugar.**
- **Emular el razonamiento humano en un ordenador.**
- **Construir sistemas que sean capaces de tomar decisiones en un entorno adverso.**

Juegos: Características y ejemplos

- **Características de los juegos que vamos a estudiar:**
  - **Juegos bipersonales.**
  - **Los jugadores mueven alternativamente.**
  - **La ventaja para un jugador es desventaja para el otro.**
  - **Los jugadores poseen toda la información sobre el estado del juego.**
  - **Hay un número finito de estados y decisiones.**
  - **No interviene el azar (dados, cartas).**
- **Ejemplos de juegos válidos:**
  - **Ajedrez, damas, go, otelo, 3 en raya, nim, ...**
- **Ejemplos de juegos que no son válidos:**
  - **Backgammon, poker, bridge, ...**

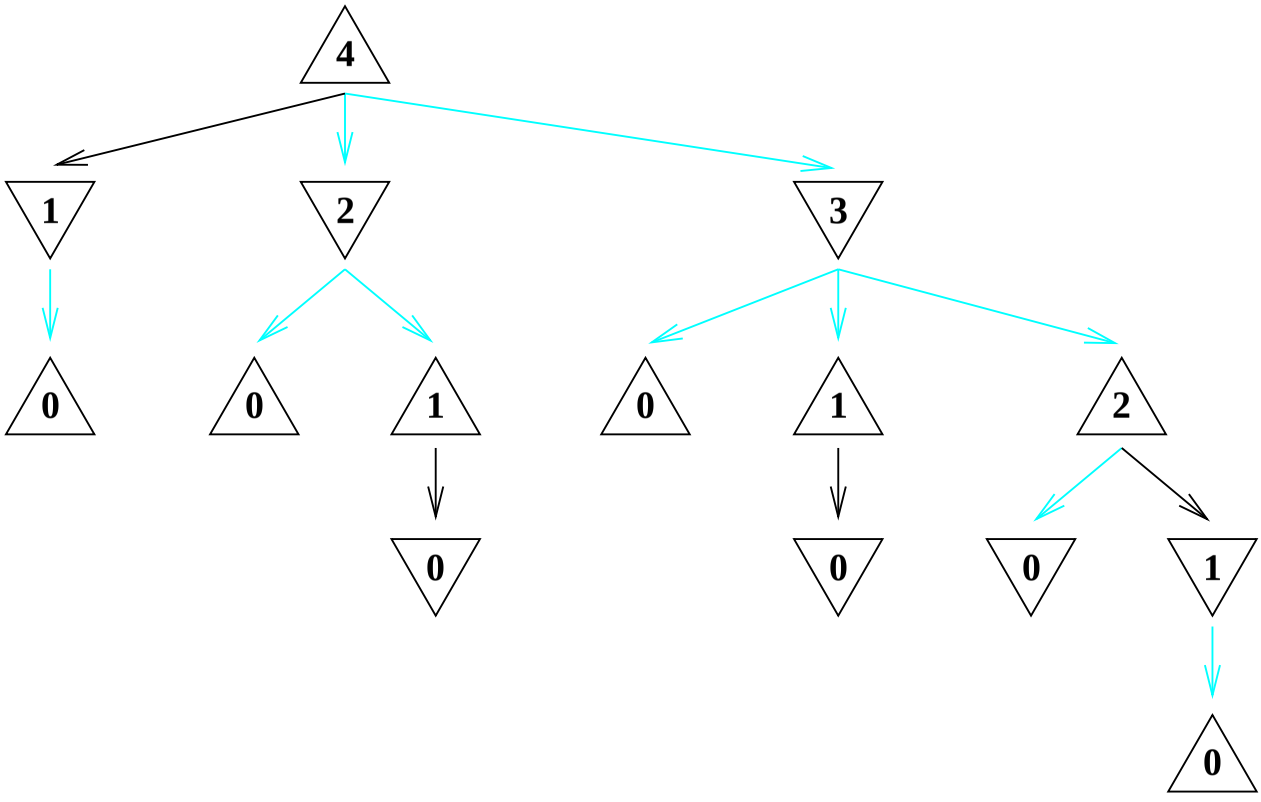
Ejemplo de juego: Nim

- **Situación inicial: Una pila con N fichas.**
- **Jugadas: Coger 1, 2 ó 3 fichas de la pila.**
- **Objetivo: Obligar al adversario a coger la última ficha. (Coger la última ficha).**
- **Desarrollo completo del juego con 4 piezas:**



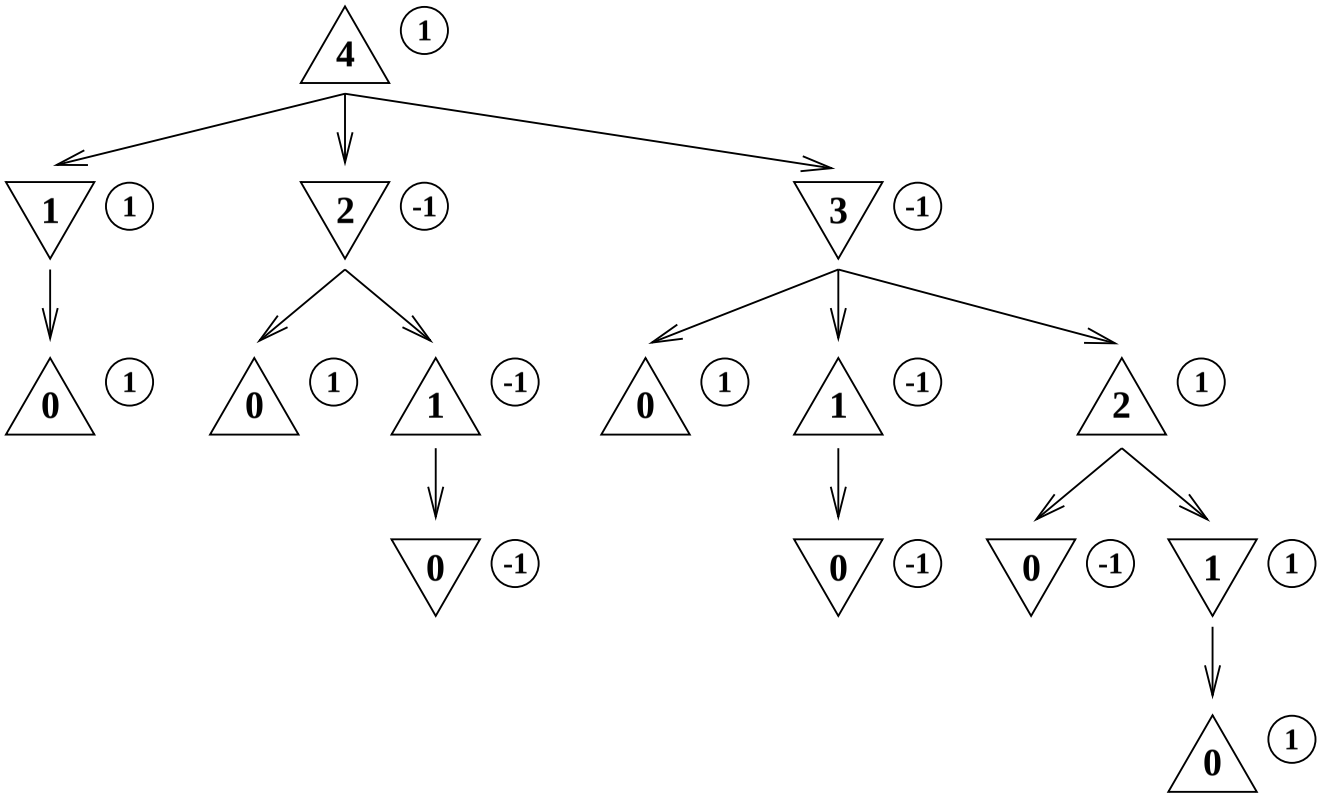
Estrategia ganadora: Nim

- **Estrategia ganadora: El movimiento que, haga lo que haga el adversario, nos lleve a una situación ganadora o a la que nos favorezca más.**
- **Estrategia ganadora en el Nim:**



Desarrollo incompleto

- **Función de evaluación estática.**
  - **Límites inferior y superior.**
- **Valores propagados, máximo y mínimo.**
- **Desarrollo del Nim con valores propagados.**



Elementos del juego

- **Jugadores:**
  - **MIN: humano**
  - **MAX: máquina**
- **Estados:**
  - **Estados iniciales.**
  - **Estados finales.**
  - **Estados ganadores.**
  - **Función de evaluación estática.**
- **Nodos:**
  - **Tablero + Jugador + Valor.**
- **Movimientos:**
  - **Generar movimientos.**
  - **Verificar movimientos del humano.**
  - **Procedimiento de decisión para la máquina.**

Representación del Nim.

- **Variables globales:**

```
(defvar *orden* 8)

(defvar *estado-inicial*)

(defvar *movimientos*)
```

- **Generador de juegos:**

```
(defun inicializa-nim (orden)
  (setf *orden* orden)
  (crea-estado-inicial)
  (list 'nim orden))
```

Representación del Nim.

- **Estado inicial:**

```
(defun crea-estado-inicial ()
  (setf *estado-inicial* *orden*))
```

- **Estados finales:**

```
(defun es-estado-final (estado)
  (= estado 0))
```

Representación del Nim.

- **Movimientos:**

```
(defun quita (fichas estado)
  (when (>= estado fichas)
        (- estado fichas)))

(defparameter *movimientos*
  '((quita 3)
    (quita 2)
    (quita 1)))

(defun aplica-movimiento (movimiento estado)
  (funcall (symbol-function (first movimiento))
           (second movimiento)
           estado))
```

Representación del Nim.

- **Estados ganadores:**

```
(defun es-estado-ganador (estado turno jugador)
  (and (= estado 0) (equal turno jugador)))
```

  - **Devuelve `T` si y sólo si en el `ESTADO`, que describe la situación del juego, cuando le toca mover al jugador `TURNO`, el `JUGADOR` ha ganado la partida. En caso contrario devuelve `NIL`.**

Representación del Nim.

- **Límites de la función de evaluación estática:**

```
(defparameter *minimo-valor* -1)

(defparameter *maximo-valor* 1)
```

- **Primera función de evaluación estática:**

```
(defun f-e-estatica-1 (estado jugador)
  (if (es-estado-final estado)
      (if (eq jugador 'max)
          *maximo-valor*
          *minimo-valor*)
      0))
```

- **Ejemplos:**

```
(f-e-estatica-1 0 'max))  =>  1
(f-e-estatica-1 0 'min))  =>  -1
(f-e-estatica-1 5 'min))  =>  0
(f-e-estatica-1 5 'max))  =>  0
```

Representación del Nim.

- **Segunda función de evaluación estática:**

```
(defun f-e-estatica-2 (estado jugador)
  (case jugador
        (max (if (= (rem estado 4) 1)
                 *minimo-valor*
                 *maximo-valor*))
        (min (if (= (rem estado 4) 1)
                 *maximo-valor*
                 *minimo-valor*))))
```

- **Ejemplos:**

```
(f-e-estatica-2 0 'max)  =>  1
(f-e-estatica-2 0 'min)  =>  -1
(f-e-estatica-2 5 'min)  =>  1
(f-e-estatica-2 5 'max)  =>  -1
```

- **Función `f-e-estatica`:**

```
(defun f-e-estatica (estado jugador)
  (f-e-estatica-2 estado jugador))
```

Procedimientos para definir juegos

```
*ESTADO-INICIAL*
(ES-ESTADO-FINAL ESTADO)
(ES-ESTADO-GANADOR ESTADO TURNO JUGADOR)
*MOVIMIENTOS*
(APLICA-MOVIMIENTO MOVIMIENTO ESTADO)
(F-E-ESTATICA ESTADO)
*MINIMO-VALOR*
*MAXIMO-VALOR*
```

Procedimiento de control de juegos.

- **Comienzo del juego:**

```
(defvar *procedimiento*)

(defun juego (&key (empieza-la-maquina? nil)
                   (procedimiento '(minimax 5)))
  (setf *procedimiento* procedimiento)
  (cond (empieza-la-maquina? (crea-nodo-j-inicial 'max)
                             (if (es-estado-final *estado-inicial*)
                                 (analiza-final *nodo-j-inicial*)
                                 (jugada-maquina *nodo-j-inicial*)))
        (t (crea-nodo-j-inicial 'min)
           (if (es-estado-final *estado-inicial*)
               (analiza-final *nodo-j-inicial*)
               (jugada-humana *nodo-j-inicial*)))))
```

Procedimiento de control de juegos.

- **Nodos del árbol de análisis:**

```
(defstruct (nodo-j (:constructor crea-nodo-j)
                   (:conc-name nil)
                   (:print-function escribe-nodo-j))
  estado
  jugador
  valor)

(defun escribe-nodo-j (nodo-j &optional (canal t) profundidad)
  (format canal "~%Estado  : ~a~%Jugador : ~a"
          (estado nodo-j)
          (jugador nodo-j)))
```

Procedimiento de control de juegos.

- **Nodo inicial.**

```
(defvar *nodo-j-inicial*)

(defun crea-nodo-j-inicial (jugador)
  (setf *nodo-j-inicial*
    (crea-nodo-j :estado *estado-inicial*
                 :jugador jugador)))
```

Procedimiento de control de juegos.

- **Controlar el final del juego.**

```
(defun analiza-final (nodo-j-final)
  (escribe-nodo-j nodo-j-final)
  (cond ((es-estado-ganador (estado nodo-j-final)
                            (jugador nodo-j-final) 'max)
         (format t "~&La maquina ha ganado"))
        ((es-estado-ganador (estado nodo-j-final)
                            (jugador nodo-j-final) 'min)
         (format t "~&El humano ha ganado"))
        (t (format t "~&Empate"))))
```

Procedimiento de control de juegos.

- **Procesar jugada máquina:**

```
(defun jugada-maquina (nodo-j)
  (escribe-nodo-j nodo-j)
  (format t "~%Mi turno.~&")
  (let ((siguiente (aplica-decision *procedimiento* nodo-j)))
    (if (es-estado-final (estado siguiente))
        (analiza-final siguiente)
        (jugada-humana siguiente))))
```

Procedimiento de control de juegos.

- **Procesar jugada humana:**

```
(defun escribe-movimientos ()
  (format t "~%Los movimientos permitidos son:")
  (let ((numero 0))
    (loop for i in *movimientos* do
          (format t "~%      ~a (~a)" i numero)
          (setf numero (+ numero 1)))))
```

Procedimiento de control de juegos.

```
(defun jugada-humana (nodo-j)
  (escribe-nodo-j nodo-j)
  (escribe-movimientos)
  (format t "~%Tu turno: ")
  (let ((m (read)))
    (cond
     ((and (integerp m) (< -1 m (length *movimientos*)))
      (let ((nuevo-estado
             (aplica-movimiento (nth m *movimientos*)
                                (estado nodo-j))))
        (cond
         (nuevo-estado
          (let ((siguiente
                 (crea-nodo-j
                  :estado nuevo-estado
                  :jugador 'max)))
            (if (es-estado-final nuevo-estado)
                (analiza-final siguiente)
                (jugada-maquina siguiente))))
         (t (format t "~&   ~a no se puede usar. " m)
            (jugada-humana nodo-j)))))
     (t (format t "~&   ~a es ilegal. " m)
        (jugada-humana nodo-j)))))
```

Decisiones de la máquina. Procedimiento minimax

- **Descripción del procedimiento de decisión:**
  - `(minimax 5)`: **Aplicar el procedimiento minimax con profundidad de análisis 5.**
- **Aplicación del procedimiento de decisión:**
  - `(aplica-decision *procedimiento* nodo-j)`: **Aplicar el** `*PROCEDIMIENTO*` **de decisión al** `NODO-J`**.**

```
(defun aplica-decision (procedimiento nodo-j)
  (funcall (symbol-function (first procedimiento))
           nodo-j
           (second procedimiento)))
```

Decisiones de la máquina. Función minimax

```
1. Si el estado de NODO-J es un final o la profundidad de análisis restante es
   cero,
   1.1. devolver un nodo-j cuyo valor sea el valor estático del estado de NODO-J.
   1.2. en caso contrario, hacer
        1.2.1 Calcular los SUCESORES del NODO-J
        1.2.2 Si la lista de SUCESORES es vacía
              1.2.2.1 devolver un nodo-j cuyo valor sea el valor estático del
              estado de NODO-J.
              1.2.2.2 en caso contrario, si el jugador del NODO-J es MAX
                      1.2.2.2.1 devolver el nodo de la lista de SUCESORES con
                                mayor valor minimax
                      1.2.2.2.2 devolver el nodo de la lista de SUCESORES con
                                menor valor minimax
```

Decisiones de la máquina. Función minimax

```
(defun minimax (nodo-j profundidad)
  (if (or (es-estado-final (estado nodo-j))
          (= profundidad 0))                                    ;1
      (crea-nodo-j :valor (f-e-estatica (estado nodo-j)
                                        (jugador nodo-j)))      ;1.1
      (let ((sucesores (sucesores nodo-j)))                     ;1.2.1
        (if (null sucesores)                                    ;1.2.2
            (crea-nodo-j :valor (f-e-estatica (estado nodo-j)
                                              (jugador nodo-j))) ;1.2.2.1
            (if (eq (jugador nodo-j) 'max)                      ;1.2.2.2
                (maximizador sucesores profundidad)             ;1.2.2.2.1
                (minimizador sucesores profundidad))))))        ;1.2.2.2.2
```

Decisiones de la máquina. Sucesores

- **Generación de sucesores:**

```
(defun sucesores (nodo-j)
  (let ((resultado ()))
    (loop for movimiento in *movimientos* do
      (let ((siguiente
             (aplica-movimiento movimiento
                                (estado nodo-j))))
        (when siguiente
          (push
            (crea-nodo-j
               :estado siguiente
               :jugador (contrario (jugador nodo-j)))
            resultado))))
    (nreverse resultado)))
```

- **Determinar el jugador contrario a uno dado.**

```
(defun contrario (jugador)
  (if (eq jugador 'max)
      'min
      'max))
```

Decisiones de la máquina. Función maximizadora

1. Crear las siguientes variables locales
   1.1. MEJOR-SUCESOR (para almacenar el mejor sucesor encontrado hasta el momento), cuyo valor es el primer elemento de la lista de SUCESORES
   1.2. MEJOR-VALOR (para almacenar el mejor valor encontrado hasta el momento), cuyo valor es el *MINIMO-VALOR* que puede llegar a tomar la función de evaluación estática.
2. Para todo SUCESOR de la lista de SUCESORES hacer
   2.1. Calcular el VALOR minimax de SUCESOR, disminuyendo en 1 la profundidad de análisis
   2.2. Cuando el VALOR calculado sea mayor que el MEJOR-VALOR, hacer
      2.2.1 Actualizar el MEJOR-VALOR, que pasa a ser VALOR
      2.2.2 Actualizar el MEJOR-SUCESOR, que pasa a ser SUCESOR
3. Poner el MEJOR-VALOR como valor del MEJOR-SUCESOR.
4. Devolver el MEJOR-SUCESOR.

Decisiones de la máquina. Función maximizadora

```
(defun maximizador (sucesores profundidad)
  (let ((mejor-sucesor (first sucesores))                          ;1.1
        (mejor-valor *minimo-valor*))                              ;1.2
    (loop for sucesor in sucesores do                              ;2
          (setf valor (valor (minimax sucesor (1- profundidad))))  ;2.1
          (when (> valor mejor-valor)                              ;2.2
                (setf mejor-valor valor)                           ;2.2.1
                (setf mejor-sucesor sucesor)))                     ;2.2.2
    (setf (valor mejor-sucesor) mejor-valor)                       ;3
    mejor-sucesor))                                                ;4
```

Decisiones de la máquina. Función minimizadora

```
1. Crear las siguientes variables locales
   1.1. MEJOR-SUCESOR (para almacenar el mejor sucesor encontrado hasta el
        momento), cuyo valor es el primer elemento de la lista de SUCESORES
   1.2. MEJOR-VALOR (para almacenar el mejor valor encontrado hasta el
        momento), cuyo valor es el *MAXIMO-VALOR* que puede llegar a tomar
        la función de evaluación estática.
2. Para todo SUCESOR de la lista de SUCESORES hacer
   2.1. Calcular el VALOR minimax de SUCESOR, disminuyendo en 1 la
        profundidad de análisis
   2.2. Cuando el VALOR calculado sea menor que el MEJOR-VALOR, hacer
        2.2.1 Actualizar el MEJOR-VALOR, que pasa a ser VALOR
        2.2.2 Actualizar el MEJOR-SUCESOR, que pasa a ser SUCESOR
3. Poner el MEJOR-VALOR como valor del MEJOR-SUCESOR.
4. Devolver el MEJOR-SUCESOR.
```

Decisiones de la máquina. Función minimizadora

```
(defun minimizador (sucesores profundidad)
  (let ((mejor-sucesor (first sucesores))                          ;1.1
        (mejor-valor *maximo-valor*))                              ;1.2
    (loop for sucesor in sucesores do                              ;2
          (setf valor (valor (minimax sucesor (1- profundidad))))  ;2.1
          (when (< valor mejor-valor)                              ;2.2
                (setf mejor-valor valor)                           ;2.2.1
                (setf mejor-sucesor sucesor)))                     ;2.2.2
    (setf (valor mejor-sucesor) mejor-valor)                       ;3
    mejor-sucesor))                                                ;4
```

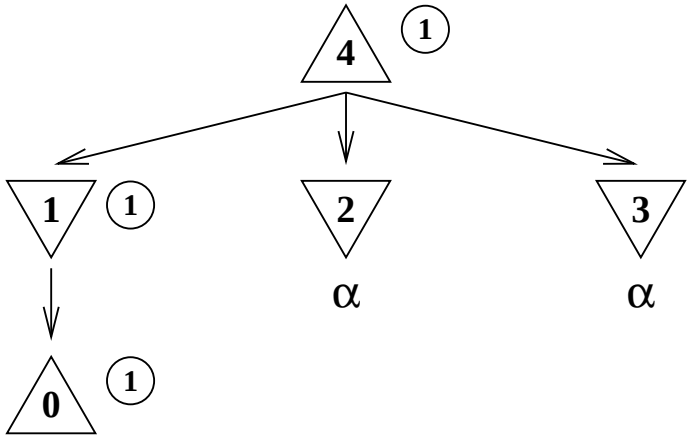
Nim con minimax y f-e-estatica-1

```
> (inicializa-nim 7)
(NIM 7)
> (juego :empieza-la-maquina? t
         :procedimiento '(minimax 1))

Estado  : 7
Jugador : MAX
Mi turno.

Estado  : 4
Jugador : MIN
Los movimientos permitidos son:
      (QUITA 3) (0)
      (QUITA 2) (1)
      (QUITA 1) (2)
Tu turno: 0

Estado  : 1
Jugador : MAX
Mi turno.

Estado  : 0
Jugador : MIN
El humano ha ganado
NIL
```

Nim con minimax y f-e-estatica-2

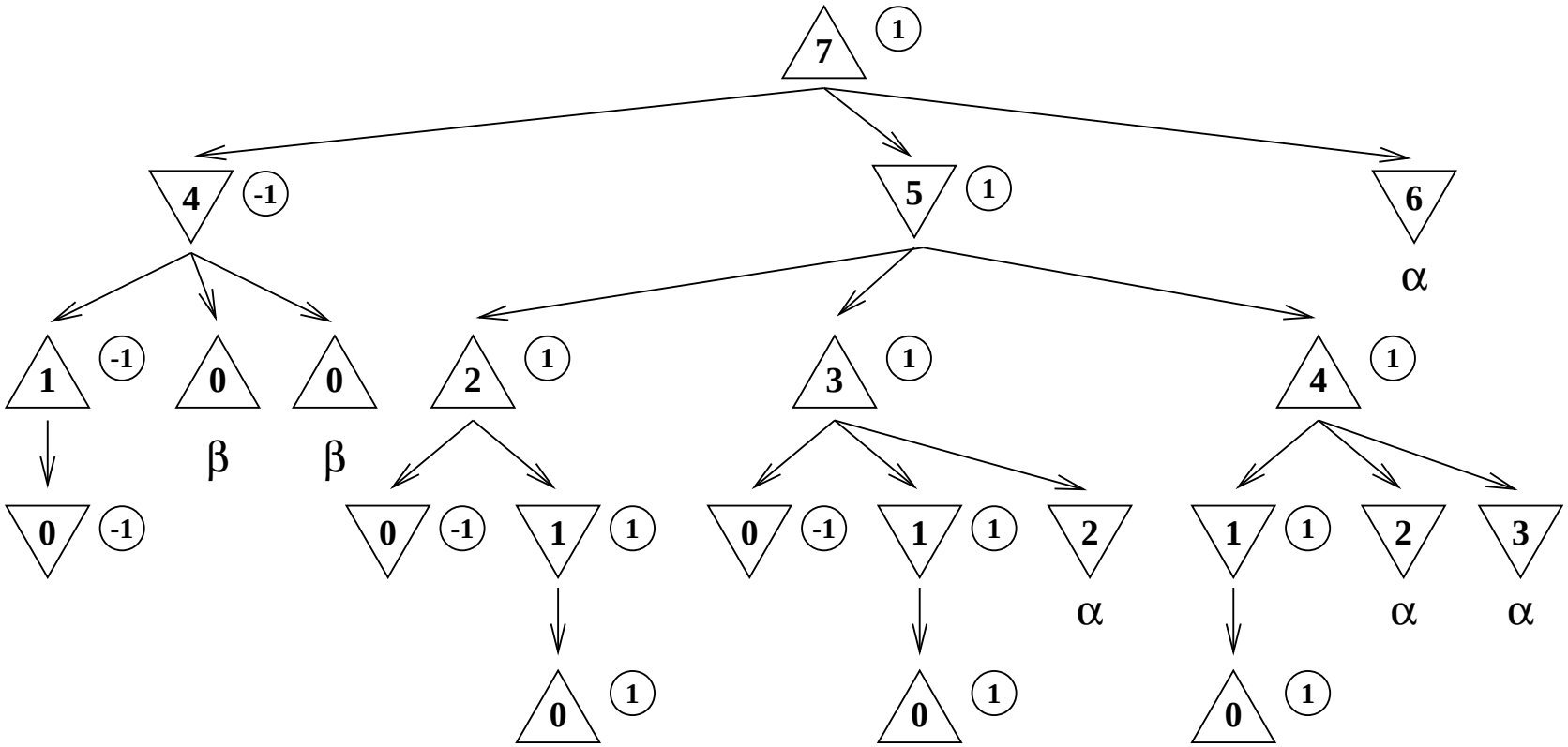
```
> (juego :empieza-la-maquina? t
         :procedimiento '(minimax 1))

Estado  : 7
Jugador : MAX
Mi turno.

Estado  : 5
Jugador : MIN
Los movimientos permitidos son:
      (QUITA 3) (0)
      (QUITA 2) (1)
      (QUITA 1) (2)
Tu turno: 0

Estado  : 2
Jugador : MAX
Mi turno.

Estado  : 1
Jugador : MIN
Los movimientos permitidos son:
      (QUITA 3) (0)
      (QUITA 2) (1)
      (QUITA 1) (2)
Tu turno: 2

Estado  : 0
Jugador : MAX
La maquina ha ganado
NIL
```

Ejemplo de poda alfa–beta: Nim 4



- **Principio alfa–beta: Si se tiene una buena (mala) idea, no perder el tiempo en averiguar lo buena (mala) que es.**

Ejemplo de poda alfa–beta: Nim 7



Decisiones de la máquina. Minimax con poda alfa–beta

```
1. Si el estado de NODO-J es un final o la profundidad de análisis restante es
   cero,
   1.1. devolver un nodo-j cuyo valor sea el valor estático del estado de NODO-J.
   1.2. en caso contrario, hacer
        1.2.1 Calcular los SUCESORES del NODO-J
        1.2.2 Si la lista de SUCESORES es vacía
              1.2.2.1 devolver un nodo-j cuyo valor sea el valor estático del
                      estado de NODO-J.
              1.2.2.2 en caso contrario, si el jugador del NODO-J es MAX
                      1.2.2.2.1 devolver el nodo de la lista de SUCESORES
                                obtenido con la función MAXIMIZADOR-A-B
                      1.2.2.2.2 devolver el nodo de la lista de SUCESORES
                                obtenido con la función MINIMIZADOR-A-B
```

Decisiones de la máquina. Minimax con poda alfa–beta

```
(defun minimax-a-b (nodo-j profundidad
                            &optional (alfa *minimo-valor*)
                            (beta *maximo-valor*))
  (if (or (es-estado-final (estado nodo-j))
          (= profundidad 0))
      (crea-nodo-j :valor (f-e-estatica (estado nodo-j)
                                        (jugador nodo-j)))
      (let ((sucesores (sucesores nodo-j)))
        (if (null sucesores)
            (crea-nodo-j :valor (f-e-estatica (estado nodo-j)
                                              (jugador nodo-j)))
            (if (eq (jugador nodo-j) 'max)
                (maximizador-a-b sucesores profundidad alfa beta)
                (minimizador-a-b sucesores profundidad alfa beta))))))
```

Decisiones de la máquina. Minimax con poda alfa–beta

1. Crear las siguientes variables locales
   1.1. MEJOR-SUCESOR (para almacenar el mejor sucesor encontrado hasta el momento o el sucesor por defecto en caso de no encotrar ninguno que mejore la cota ALFA), cuyo valor es el primer elemento de la lista de SUCESORES
   1.2. VALOR (para almacenar el mejor valor encontrado hasta el momento), cuyo valor inicial es 0.
2. Para todo SUCESOR de la lista de SUCESORES hacer
   2.1. Calcular el VALOR minimax-a-b de SUCESOR, disminuyendo en 1 la profundidad de análisis y utilizando los valores de ALFA y BETA.
   2.2. Cuando el VALOR calculado sea mayor que ALFA, hacer
        2.2.1 Actualizar ALFA, que pasa a ser VALOR
        2.2.2 Actualizar el MEJOR-SUCESOR, que pasa a ser SUCESOR
   2.3. Cuando el valor de ALFA supere o iguale al de BETA, terminar el bucle: se ha producido un corte alfa.
3. Poner ALFA como valor del MEJOR-SUCESOR.
4. Devolver el MEJOR-SUCESOR.

Decisiones de la máquina. Minimax con poda alfa–beta

```
(defun maximizador-a-b (sucesores profundidad alfa beta)
  (let ((mejor-sucesor (first sucesores))
        (valor 0))
    (loop for sucesor in sucesores do
          (setf valor
                (valor (minimax-a-b sucesor (1- profundidad) alfa beta)))
          (when (> valor alfa)
                (setf alfa valor)
                (setf mejor-sucesor sucesor))
          (when (>= alfa beta)
                (return)))
    (setf (valor mejor-sucesor) alfa)
    mejor-sucesor))
```

Decisiones de la máquina. Minimax con poda alfa–beta

1. Crear las siguientes variables locales
   1.1. MEJOR-SUCESOR (para almacenar el mejor sucesor encontrado hasta el momento o el sucesor por defecto en caso de no encotrar ninguno que mejore la cota BETA), cuyo valor es el primer elemento de la lista de SUCESORES
   1.2. VALOR (para almacenar el mejor valor encontrado hasta el momento), cuyo valor inicial es 0.
2. Para todo SUCESOR de la lista de SUCESORES hacer
   2.1. Calcular el VALOR minimax-a-b de SUCESOR, disminuyendo en 1 la profundidad de análisis y utilizando los valores de ALFA y BETA.
   2.2. Cuando el VALOR calculado sea menor que BETA, hacer
        2.2.1 Actualizar BETA, que pasa a ser VALOR
        2.2.2 Actualizar el MEJOR-SUCESOR, que pasa a ser SUCESOR
   2.3. Cuando el valor de ALFA supere o iguale al de BETA, terminar el bucle: se ha producido un corte beta.
3. Poner BETA como valor del MEJOR-SUCESOR.
4. Devolver el MEJOR-SUCESOR.

Decisiones de la máquina. Minimax con poda alfa–beta

```
(defun minimizador-a-b (sucesores profundidad alfa beta)
  (let ((mejor-sucesor (first sucesores))
        (valor 0))
    (loop for sucesor in sucesores do
          (setf valor
                (valor (minimax-a-b sucesor (1- profundidad) alfa beta)))
          (when (< valor beta)
                (setf beta valor)
                (setf mejor-sucesor sucesor))
          (when (>= alfa beta)
                (return)))
    (setf (valor mejor-sucesor) beta)
    mejor-sucesor))
```

Complejidad de minimax y alfa–beta

- **Complejidad:**
  - r: factor de ramificación.
  - p: profundidad de la búsqueda.
  - Complejidad en tiempo de minimax: $O(r^p)$.
  - Complejidad en tiempo de minimax con poda alfa–beta, en el mejor caso: $O(r^{p/2})$.
- **Aplicación al ajedrez**
  - Factor de ramificación: 35
  - Número de movimientos en una partida media: 50.
  - Número de nodos analizados por minimax: $35^{100} \simeq 10^{154.4}$.
  - Número de nodos analizados por minimax con poda alfa–beta, en el mejor caso: $35^{50} \simeq 10^{77.2}$.
  - Número de posiciones legales: 10^{40}.

Complejidad en el Nim

- Estados evaluados
  - empezando la máquina,
  - con profundidad 20 y
  - eligiendo el humano siempre 1.

	8	12	16	20
minimax	192	2223	25472	291551
minimax-a-b	16	67	264	1023

- Tiempo y espacio para orden 16:

	Tiempo	Espacio
minimax	14.55 sec	3245720 bytes
minimax-a-b	0.32 sec	69812 bytes

3 en raya: Ejemplo de juego

```
> (juego :empieza-la-maquina? t
         :procedimiento '(minimax 2))

Estado  :
 . . .   0 1 2
 . . .   3 4 5
 . . .   6 7 8

Jugador : MAX
Mi turno.

Estado  :
 . . .   0 1 2
 . X .   3 4 5
 . . .   6 7 8

Jugador : MIN
Los movimientos permitidos son:
      (PON-FICHA-EN 0) (0)
      (PON-FICHA-EN 1) (1)
      (PON-FICHA-EN 2) (2)
      (PON-FICHA-EN 3) (3)
      (PON-FICHA-EN 4) (4)
      (PON-FICHA-EN 5) (5)
      (PON-FICHA-EN 6) (6)
      (PON-FICHA-EN 7) (7)
      (PON-FICHA-EN 8) (8)
Tu turno: 1
```

3 en raya: Ejemplo de juego

```
Estado  :
 . O .   0 1 2
 . X .   3 4 5
 . . .   6 7 8

Jugador : MAX
Mi turno.

Estado  :
 X O .   0 1 2
 . X .   3 4 5
 . . .   6 7 8

Jugador : MIN
Los movimientos permitidos son:
      (PON-FICHA-EN 0) (0)
      (PON-FICHA-EN 1) (1)
      (PON-FICHA-EN 2) (2)
      (PON-FICHA-EN 3) (3)
      (PON-FICHA-EN 4) (4)
      (PON-FICHA-EN 5) (5)
      (PON-FICHA-EN 6) (6)
      (PON-FICHA-EN 7) (7)
      (PON-FICHA-EN 8) (8)
Tu turno: 8
```

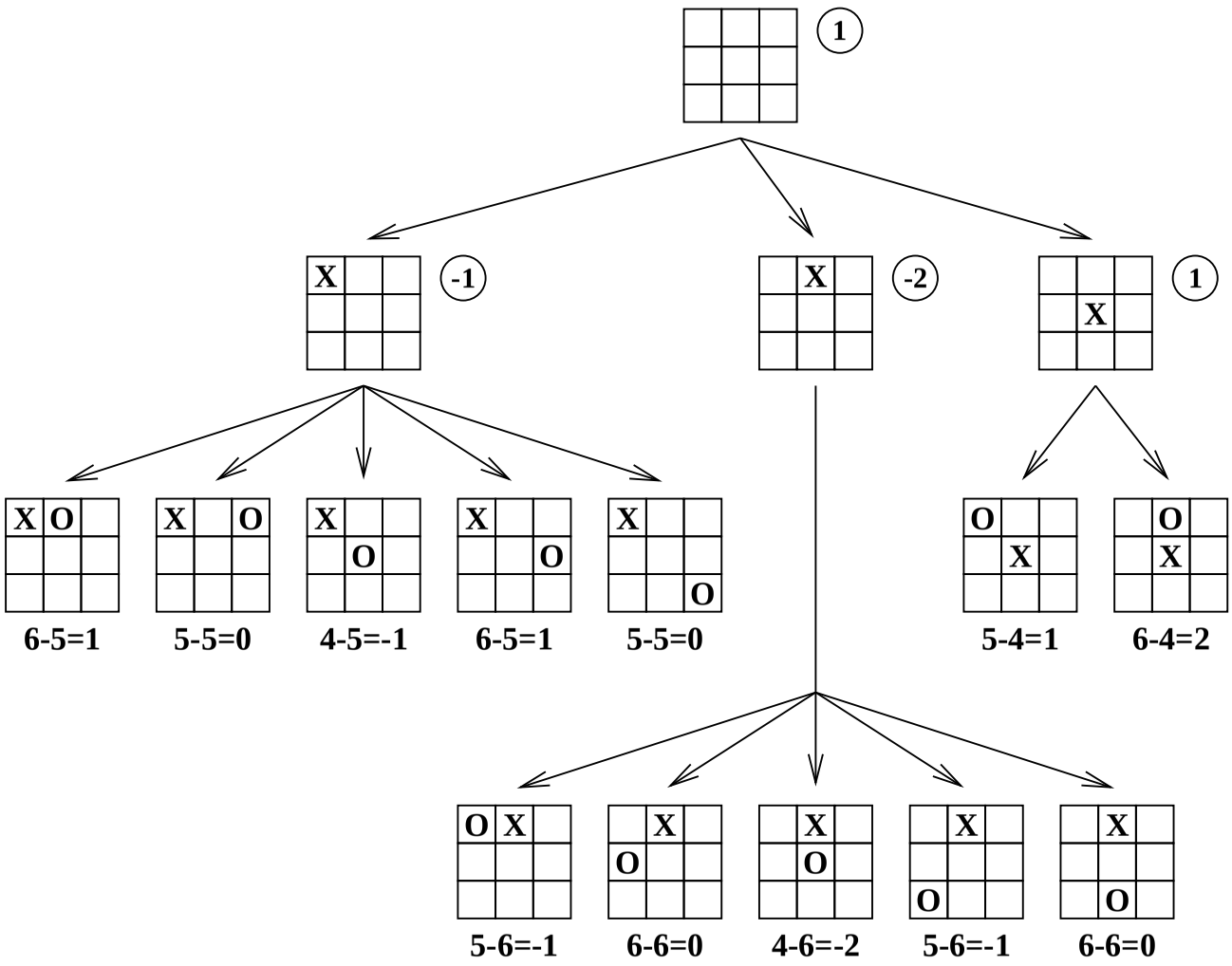
3 en raya: Ejemplo de juego

```
Estado  :
 X O .   0 1 2
 . X .   3 4 5
 . . O   6 7 8

Jugador : MAX
Mi turno.

Estado  :
 X O X   0 1 2
 . X .   3 4 5
 . . O   6 7 8

Jugador : MIN
Los movimientos permitidos son:
      (PON-FICHA-EN 0) (0)
      (PON-FICHA-EN 1) (1)
      (PON-FICHA-EN 2) (2)
      (PON-FICHA-EN 3) (3)
      (PON-FICHA-EN 4) (4)
      (PON-FICHA-EN 5) (5)
      (PON-FICHA-EN 6) (6)
      (PON-FICHA-EN 7) (7)
      (PON-FICHA-EN 8) (8)
Tu turno: 7
```

3 en raya: Ejemplo de juego

```
Estado  :
 X O X   0 1 2
 . X .   3 4 5
 . O O   6 7 8

Jugador : MAX
Mi turno.

Estado  :
 X O X   0 1 2
 . X .   3 4 5
 X O O   6 7 8

Jugador : MIN
La maquina ha ganado
NIL
```

3 en raya: Evaluación



3 en raya: Representación

- **Variables globales:**

```
(defvar *estado-inicial*)

(defvar *movimientos*)
```

- **Generador de juegos:**

```
(defun inicializa-3-en-raya (ficha)
  (crea-estado-inicial ficha)
  (list '3-en-raya. 'Empieza ficha))
```

- **Representación de estados:**

```
(defstruct (estado (:constructor crea-estado)
                   (:conc-name nil)
                   (:print-function escribe-estado))
  tablero
  ficha)
```

3 en raya: Representación

- **Escritura de estados:**

```
(defun escribe-estado (estado &optional (canal t) profundidad)
  (let ((tablero (tablero estado)))
    (format t "~% ~d ~d ~d   0 1 2~% ~d ~d ~d   3 4 5~% ~d ~d ~d   6 7 8~%~%"
            (or (nth 0 tablero) ".")
            (or (nth 1 tablero) ".")
            (or (nth 2 tablero) ".")
            (or (nth 3 tablero) ".")
            (or (nth 4 tablero) ".")
            (or (nth 5 tablero) ".")
            (or (nth 6 tablero) ".")
            (or (nth 7 tablero) ".")
            (or (nth 8 tablero) "."))))
```

3 en raya: Representación

- **Estado inicial:**

```
(defun crea-estado-inicial (ficha)
  (setf *estado-inicial*
    (crea-estado :tablero '(nil nil nil nil nil nil nil nil nil)
                 :ficha ficha)))
```

- **Estados finales:**

```
(defun es-estado-final (estado)
  (or (es-estado-completo estado)
      (tiene-linea-ganadora estado)))

(defun es-estado-completo (estado)
  (not (position nil (tablero estado))))
```

3 en raya: Representación

```
(defvar *lineas-ganadoras* '((0 1 2) (3 4 5) (6 7 8)
                             (0 3 6) (1 4 7) (2 5 8)
                             (0 4 8) (2 4 6)))

(defun tiene-linea-ganadora (estado)
  (loop for linea in *lineas-ganadoras* thereis
        (es-linea-ganadora linea (tablero estado))))

(defun es-linea-ganadora (linea tablero)
  (let ((valor (nth (first linea) tablero)))
    (when (and (not (null valor))
               (equalp valor (nth (second linea) tablero))
               (equalp valor (nth (third linea) tablero)))
          valor)))
```

3 en raya: Representación

- **Movimientos:**

```
(setf *movimientos* (loop for i from 0 to 8 collect (list 'pon-ficha-en i)))

(defun pon-ficha-en (posicion estado)
  (when (null (nth posicion (tablero estado)))
        (let ((nuevo-tablero (loop for i in (tablero estado) collect i)))
          (setf (nth posicion nuevo-tablero) (ficha estado))
          (crea-estado :tablero nuevo-tablero
                       :ficha (ficha-opuesta (ficha estado))))))

(defun ficha-opuesta (ficha)
  (if (eq ficha 'x)
      'o
      'x))
```

3 en raya: Representación

- **Aplicar movimientos:**

```
(defun aplica-movimiento (movimiento estado)
  (funcall (symbol-function (first movimiento))
           (second movimiento)
           estado))
```

- **Estados ganadores:**

```
(defun es-estado-ganador (estado turno jugador)
  (if (tiene-linea-ganadora estado)
      (not (equalp jugador turno))
      nil))
```

3 en raya: Representación

- **Límites de la función de evaluación estática:**

```
(defparameter *minimo-valor* -99999)

(defparameter *maximo-valor* 99999)
```

- **Función de evaluación estática:**

```
(defun f-e-estatica (estado jugador)
  (cond ((es-estado-ganador estado jugador 'max) *maximo-valor*)
        ((es-estado-ganador estado jugador 'min) *minimo-valor*)
        ((es-estado-completo estado) 0)
        (t (- (posibles-lineas-ganadoras estado jugador 'max)
              (posibles-lineas-ganadoras estado jugador 'min)))))
```

3 en raya: Representación

```
(defun posibles-lineas-ganadoras (estado turno jugador)
  (loop for linea in *lineas-ganadoras* counting
    (not (esta (tablero estado)
               (if (eq turno jugador)
                   (ficha-opuesta (ficha estado))
                   (ficha estado))
               linea))))

(defun esta (tablero ficha linea)
  (loop for i in linea
    thereis (eq (nth i tablero) ficha)))
```

3 en raya: Eliminación de simétricos

- **Sucesores:**

```
(defun sucesores (nodo-j)
  (elimina-simetricos (todos-los-sucesores nodo-j)))

(defun todos-los-sucesores (nodo-j)
  (let ((resultado ()))
    (loop for movimiento in *movimientos* do
          (let ((siguiente (aplica-movimiento movimiento
                                              (estado nodo-j))))
            (when siguiente
                  (push (crea-nodo-j :estado siguiente
                                     :jugador (contrario (jugador nodo-j)))
                        resultado))))
    (nreverse resultado)))
```

3 en raya: Eliminación de simétricos

```
(defun elimina-simetricos (nodos-sucesores)
  (delete-duplicates nodos-sucesores :test #'son-simetricos :from-end t))

(defun son-simetricos (nodo-1 nodo-2)
  (let ((tablero-1 (tablero (estado nodo-1)))
        (tablero-2 (tablero (estado nodo-2))))
    (or (son-simetricos-respecto-columna-central tablero-1 tablero-2)
        (son-simetricos-respecto-fila-central tablero-1 tablero-2)
        (son-simetricos-respecto-diagonal-principal tablero-1 tablero-2)
        (son-simetricos-respecto-diagonal-secundaria tablero-1 tablero-2))))
```

3 en raya: Eliminación de simétricos

```
(defun son-simetricos-respecto-fila-central (tablero-1 tablero-2)
  (loop for i from 0 to 2
    always (equal (fila i tablero-1)
                  (fila (- 2 i) tablero-2))))

(defun fila (i tablero)
  (loop for x from (* 3 i) to (+ (* 3 i) 2)
    collect (nth x tablero)))

(defun son-simetricos-respecto-columna-central (tablero-1 tablero-2)
  (loop for j from 0 to 2
    always (equal (columna j tablero-1)
                  (columna (- 2 j) tablero-2))))
```

3 en raya: Eliminación de simétricos

```
(defun columna (j tablero)
  (loop for x from j to (+ j (* 3 2)) by 3
    collect (nth x tablero)))

(defun son-simetricos-respecto-diagonal-principal (tablero-1 tablero-2)
  (loop for j from 0 to 2
    always (equal (columna j tablero-1)
                  (fila j tablero-2))))

(defun son-simetricos-respecto-diagonal-secundaria (tablero-1 tablero-2)
  (loop for j from 0 to 2
    always (equal (columna j tablero-1)
                  (reverse (fila (- 2 j) tablero-2)))))
```

Bibliografía

- **[Cortés–93]**
  Cap. 4.7.1: "Juegos: la estrategia minimax y sus modificaciones".
- **[Mira–95]**
  Cap. 4.4: "Búsqueda con adversarios".
- **[Rich–91]**
  Cap. 12: "Los juegos".
- **[Russell–97]**
  Cap. 5: "Juegos".
- **[Winston–94]**
  Cap. 5: "Arboles y búsqueda con adversario".