Tema 7: Búsqueda con adversario (juegos)

José Luis Ruiz Reina
José Antonio Alonso
Franciso J. Martín Mateos
María José Hidalgo

Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Inteligencia Artificial I, 2012

Índice

Juegos: introducción

- Programación de las máquinas para juegos de estrategia
  - Basados en búsqueda (enormes espacios de búsqueda)
  - La existencia de un oponente introduce *incertidumbre*
  - Existen limitaciones en el tiempo de respuesta
  - Pueden competir contra la inteligencia humana
- En general, se trata de construir sistemas que sean capaces de tomar decisiones en un entorno adverso.
- Tipos de juegos:
  - Información completa, deterministas: ajedrez, damas
  - Información completa, con azar: backgammon
  - Información incompleta, con azar: juegos de cartas

Juegos: características y ejemplos

- Características de los juegos que vamos a estudiar en este tema:
  - 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.
- Ejemplos de juegos de ese tipo:
  - Ajedrez, damas, go, otelo, 3 en raya, nim, ...
- Ejemplos de juegos que no son de ese tipo:
  - Backgammon, poker, bridge, ...

Ejemplo de juego: Nim

- Situación inicial: una pila con N fichas.
- Jugadas: tomar 1, 2 ó 3 fichas de la pila.
- Objetivo: obligar al adversario a coger la última ficha.
- Ejemplo de jugada en el Nim, comenzando con 9 fichas
  - Jugador 1 coge 3 fichas (quedan 6)
  - Jugador 2 coge 1 ficha (quedan 5)
  - Jugador 1 coge 2 fichas (quedan 3)
  - Jugador 2 coge 2 fichas (queda 1)
  - Jugador 1 coge 1 ficha (no quedan fichas)
  - Por tanto, jugador 2 gana

Elementos de un juego

- Jugadores:
  - Máquina (al que llamaremos MAX)
  - Humano (al que llamaremos MIN)
- Estados: situaciones por las que puede pasar el juego
- Estado inicial (comienzo del juego)
- Estados finales (final del juego)
- En los estados finales, especificaremos qué jugador gana
- Movimientos: operadores que se aplican a los estados, cambiando las situaciones de juego
- Función de utilidad: valoración (respecto de MAX) basada en las reglas del juego, que se asigna a un estado final y a un turno

Elementos del juego en el Nim

- Estados: número fichas que quedan en la mesa
- Estado inicial: número de fichas al comienzo
- Un único estado final: 0
- El estado final es ganador para un jugador si es su turno
- Movimientos: tomar 1, 2 ó 3 fichas
- Función de utilidad (para el estado final): 1 si le toca a MAX y -1 si le toca a MIN

3 en raya

- En un tablero 3x3, un jugador posee fichas “X” y otro fichas “O”. En cada turno el jugador coloca una ficha en el tablero. Gana el que consigue colocar tres de sus fichas en línea

X	O	
	X	
	X	O

X	O	X
O	X	O
X	X	O

Elementos del juego en el 3 en raya

- Estados: tablero + ficha que se pondrá a continuación
- Estado inicial: tablero vacío + ficha de salida
- Estados finales: tableros completos o con línea ganadora
- Estados ganadores para un jugador: estados finales con línea ganadora en los que no le toca poner

Elementos del juego en el 3 en raya

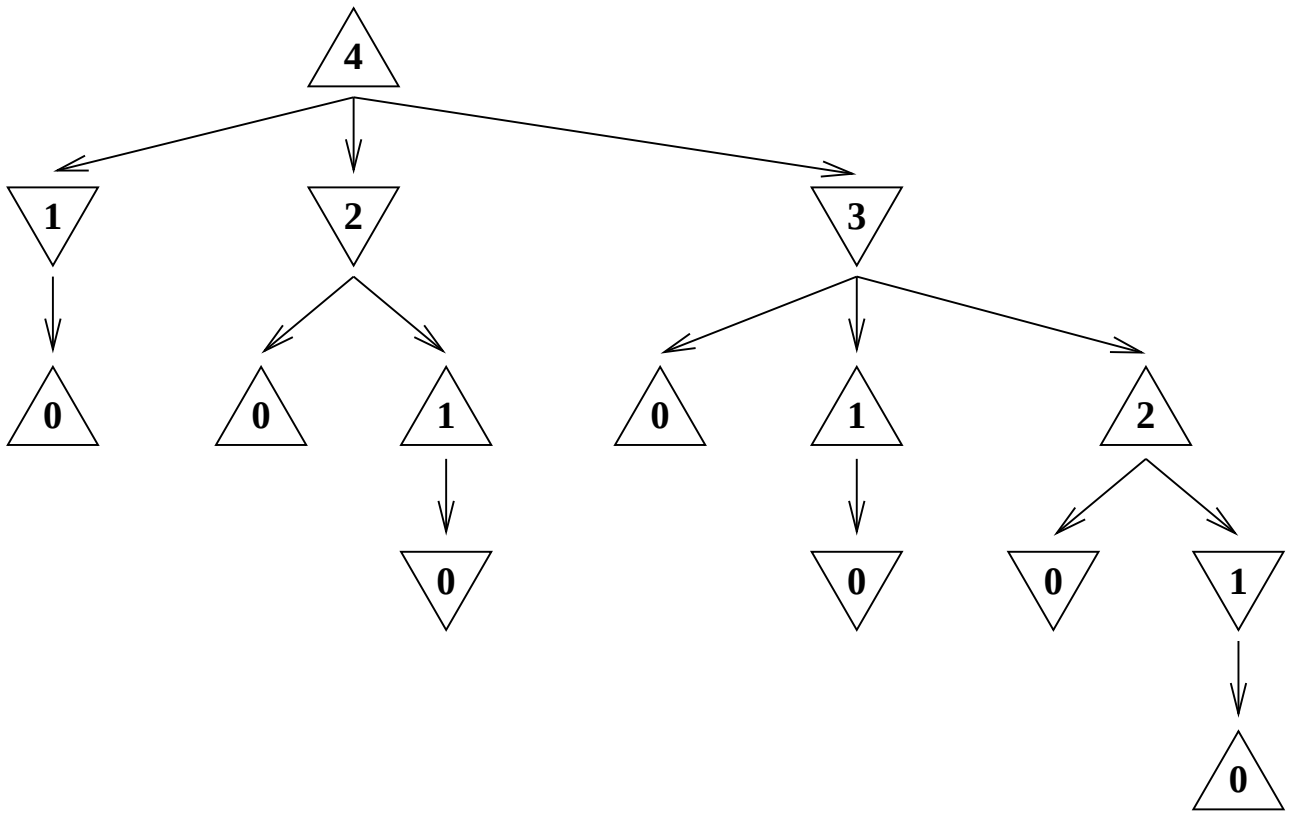
- Movimientos:
  - 9 movimientos posibles, uno por casilla
  - Colocar ficha en i ($\boldsymbol{i} = \mathbf{0}, \ldots, \mathbf{8}$)
- Aplicación de movimientos:
  - Aplicable si la casilla no está ocupada
  - Estado resultante: colocar la ficha que toca en la casilla especificada
- Función de utilidad:
  - 1 si es ganador para MAX
  - 0 si es tablas
  - -1 si es ganador para MIN

Implementación de la representación de un juego

- Estructuras de datos para estados y movimientos
- Funciones y variables:
  - Una variable **`*estado-inicial*`**
  - Una función **`es-estado-final(estado)`**
  - Una función **`es-estado-ganador(estado,turno,jugador)`**
  - Una lista **`*movimientos*`**
  - Una función **`aplica-movimiento(movimiento,estado)`**
  - Una función **`f-utilidad(estado,turno)`**
- **`aplica-movimiento(movimiento,estado)`**:
  - devuelve **`no-aplicable`** si **`movimiento`** no es aplicable a **`estado`**, o el estado resultante en otro caso.
- **`es-estado-ganador(estado,turno,jugador)`**:
  - Devuelve *verdad* si **`jugador`** gana el juego en el estado final **`estado`** cuando le toca al jugador **`turno`** (*falso* en caso contrario)

Árboles de juego

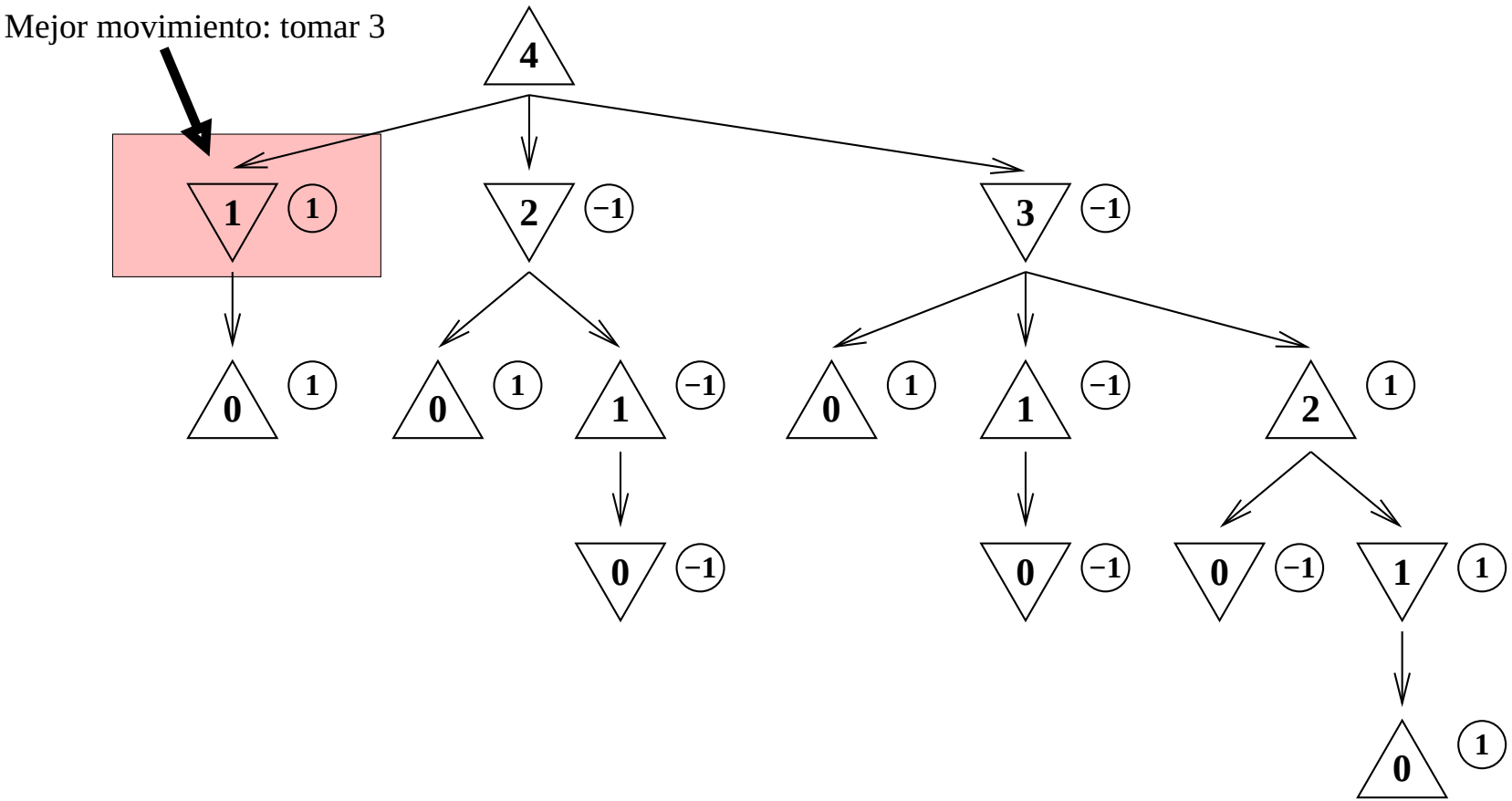
- Representación de *todas las posibles situaciones* que se pueden dar en el juego, a partir de un estado dado
  - Cada nivel corresponde a un jugador (el primero para MAX)
- Un árbol de juego en el Nim:



El problema de decidir el movimiento adecuado

- En su turno, la máquina debe decidir qué movimiento hacer. Idea:
  - En cada turno, construir el árbol de juego completo cuyo nodo raíz sea la situación actual, desarrollándolo hasta los estados finales
  - Valorar los finales según la función de utilidad
  - *Propagar* hacia arriba los valores de la función
  - Elegir el movimiento que lleve al estado sucesor del actual con mejor valoración
- La propagación de valores se hace según el principio *minimax*:
  - MAX siempre escogerá lo mejor para MAX y MIN lo peor para MAX
  - Un nodo de MAX toma el valor del sucesor con *mayor* valor
  - Un nodo de MIN toma el valor del sucesor con *menor* valor

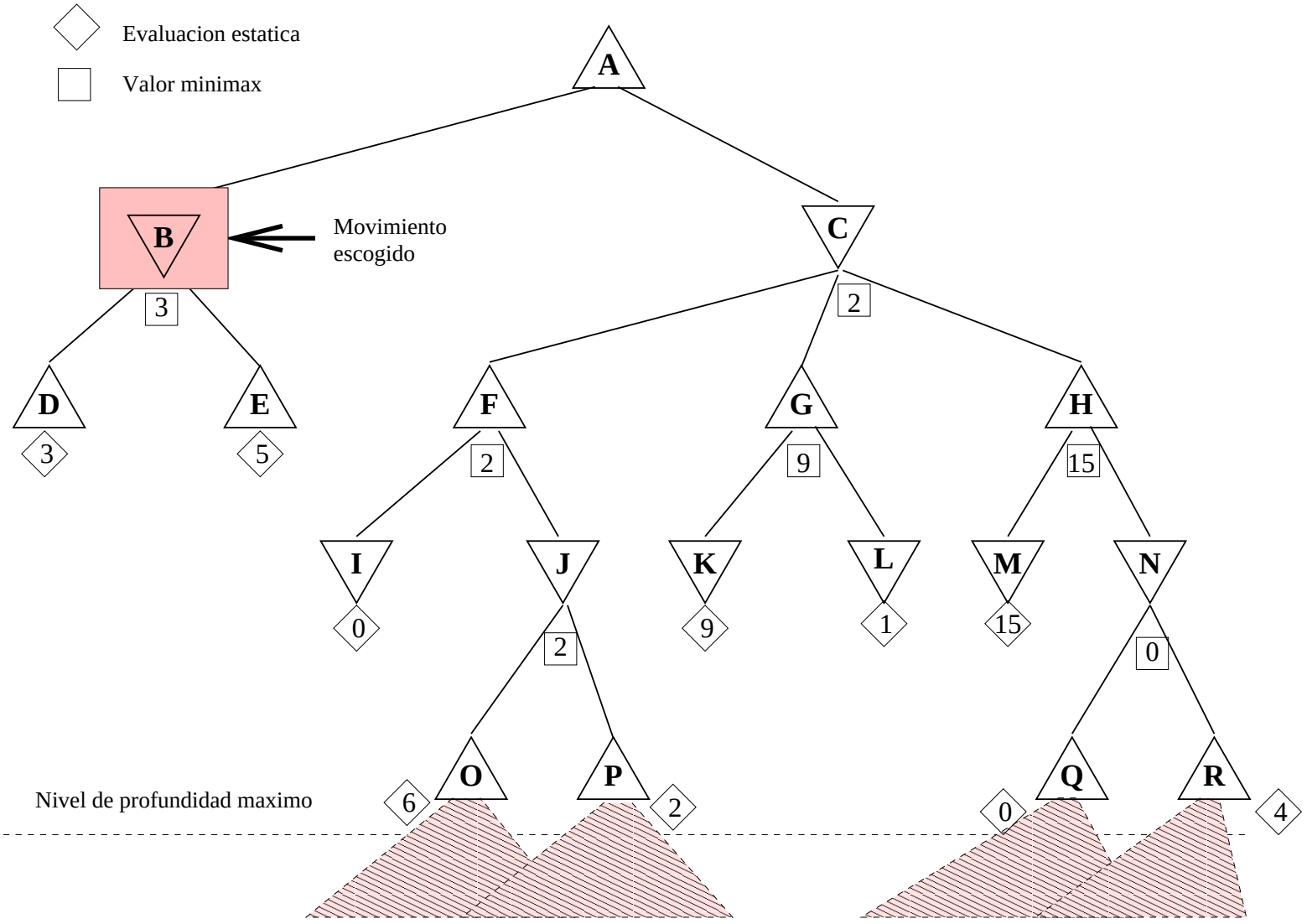
Un arbol minimax y completo para el Nim (4 de inicio)



Complejidad de los árboles de prueba y heurística

- Problema: en la mayoría de los juegos, un árbol de juego completo es enorme
  - Complejidad exponencial y tiempo limitado para decidir
  - En la práctica es imposible generar todo el árbol
- Complejidad en el ajedrez
  - Número medio de alternativas de movimientos en un turno: 35
  - Número medio de movimientos de una partida: 100
  - ¡¡¡Árbol con $\mathbf{35^{100}} \sim \mathbf{10^{154}}$ nodos!!!
- Idea:
  - No expandir el árbol hasta los estados finales sino sólo hasta un determinado nivel de profundidad
  - Valorar los estados que sean hojas del árbol usando una función *heurística*
  - Propagar valores usando el principio minimax

Ejemplo: árbol minimax con evaluación estática



Heurística: función de evaluación estática

- Función de evaluación estática:
  - Dado un estado del juego y un turno, es una *estimación* de la bondad de tal situación respecto de MAX
  - En los estados finales, debería coincidir con la de utilidad
  - Esta función heurística codifica todo el *conocimiento* que poseemos acerca del juego
  - Cuanto mayor el valor, mejor el estado para MAX
  - Lo importante es la comparación del valor entre los estados
- Implementación:
  - En lo que sigue, asumiremos que en lugar de una función **`f-utilidad(estado,turno)`**, disponemos de una función **`f-e-estatica(estado,turno)`**, definida *sobre todos* los estados
  - También necesitaremos dos variables: **`*maximo-valor*`** y **`*minimo-valor*`** almacenando, respectivamente, cotas para el mayor y el menor valor que puede tomar la función de evaluación estática

Ejemplos de función de evaluación estática en el Nim

- ***MAXIMO-VALOR*=1** y ***MINIMO-VALOR*=-1**
- Primera función de evaluación estática (muy poco informada):

```
FUNCION F-E-ESTATICA(ESTADO,TURNO)
  Si ES-ESTADO-FINAL(ESTADO)
     Si TURNO=MAX, devolver *MAXIMO-VALOR*
     en otro caso, devolver *MINIMO-VALOR*
  en otro caso, devolver 0
```

- Segunda función de evaluación estática, estrategia ganadora (comenzando):

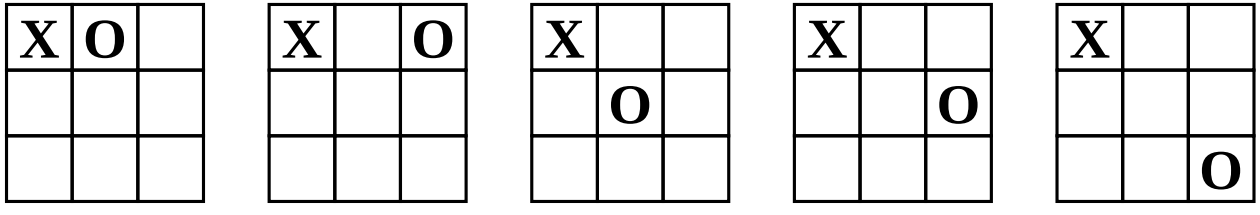
```
FUNCION F-E-ESTATICA(ESTADO,TURNO)
  Si TURNO=MAX,
     Si REM(ESTADO,4)=1, devolver *MINIMO-VALOR*
     en caso contrario, devolver *MAXIMO-VALOR*
  en caso contrario,
     Si REM(ESTADO,4)=1, devolver *MAXIMO-VALOR*
     en caso contrario, devolver *MINIMO-VALOR*
```

Una función de evaluación estática en el 3 en raya

- Función de evaluación estática:
  - Una línea potencialmente ganadora para un jugador es aquella que el jugador podría completar para ganar
  - ***MAXIMO-VALOR*=9999999** y ***MINIMO-VALOR*=-9999999**

```
FUNCION F-E-ESTATICA(ESTADO,TURNO)
1. Si ES-ESTADO-GANADOR(ESTADO,TURNO,MAX),
       devolver *MAXIMO-VALOR*
   si no, si ES-ESTADO-GANADOR(ESTADO,TURNO,MIN),
             devolver *MINIMO-VALOR*
   si no, si ES-ESTADO-FINAL(ESTADO), devolver 0
   si no, devolver la diferencia entre el número de
          posibles líneas ganadoras para MAX y el
          número de posibles líneas ganadoras para MIN
```

- Ejemplos (suponiendo que MAX juega con “X”):



Implementación del procedimiento de decisión minimax

- Nodo de juego: estado + jugador turno
  - Funciones de acceso: **ESTADO(NODO)** y **JUGADOR(NODO)**
- Sucesores de un nodo:

```
FUNCION SUCESOR(NODO,MOVIMIENTO)
1. Hacer ESTADO-SUCESOR igual a
    APLICA-MOVIMIENTO(MOVIMIENTO,ESTADO(NODO))
2. Si ESTADO-SUCESOR=NO-APLICABLE
      devolver NO-APLICABLE
   en caso contrario,
      devolver un nodo cuyo estado es ESTADO-SUCESOR,
       y cuyo jugador es el contrario de JUGADOR(NODO)

FUNCION SUCESORES(NODO)
1. Hacer SUCESORES vacío
2. Para cada MOVIMIENTO en *MOVIMIENTO*,
       si SUCESOR(NODO,MOVIMIENTO) =/= NO-APLICABLE,
          incluir SUCESOR(NODO,OPERADOR) en SUCESORES
3. Devolver SUCESORES
```

Implementación de minimax

- Decisión minimax:

```
FUNCION DECISION-MINIMAX(ACTUAL,PROFUNDIDAD)
1. Hacer MAX-VAL igual a *MINIMO-VALOR*
2. Por cada NODO en SUCESORES(ACTUAL),
       2.1 Hacer VALOR-ACTUAL igual
            a VALOR-MINIMAX(NODO,PROFUNDIDAD-1)
       2.2 Si VALOR-ACTUAL >= MAX-VAL,
                 hacer MAX-VAL igual a VALOR-ACTUAL  y
                 hacer MAX-NODO igual a NODO
3. Devolver MAX-NODO
```

- Cálculo del valor minimax:

```
FUNCION VALOR-MINIMAX(NODO,PROFUNDIDAD)
1. Si ES-ESTADO-FINAL(ESTADO(NODO)) o
      PROFUNDIDAD=0 o
      SUCESORES(NODO) igual a vacío,
       devolver F-E-ESTATICA(ESTADO(NODO),JUGADOR(NODO))
   si no, si JUGADOR(NODO)=MAX, devolver
       MAXIMIZADOR(SUCESORES(NODO),PROFUNDIDAD-1)
   si no, devolver MINIMIZADOR(SUCESORES(NODO),PROFUNDIDAD-1)
```

Implementación de minimax

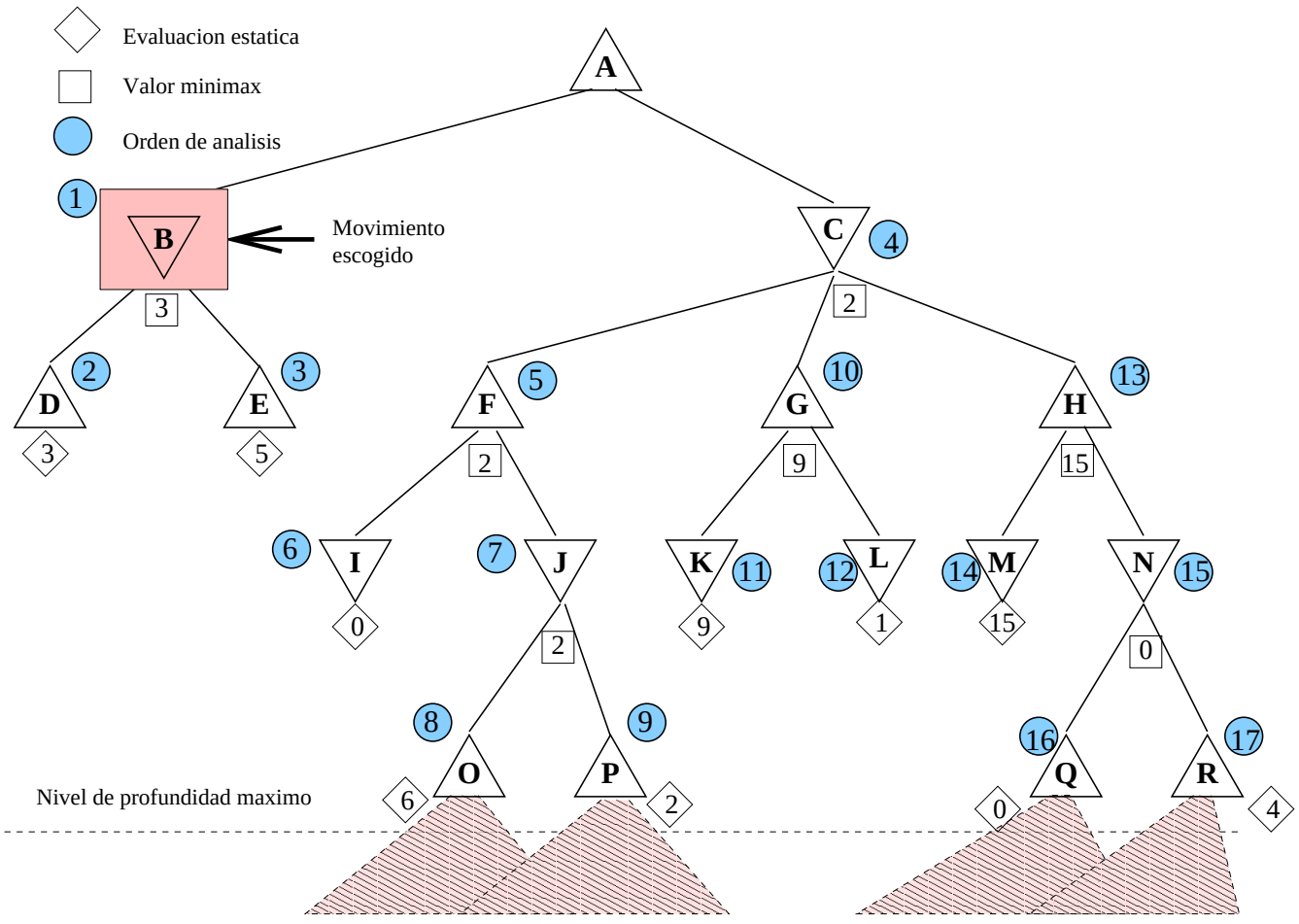
- Cálculo de máximos y mínimos:

```
FUNCION MAXIMIZADOR(SUCESORES,PROFUNDIDAD)
1. Hacer MAX-VAL igual a *MINIMO-VALOR*
2. Por cada NODO en SUCESORES,
        2.1 Hacer VALOR-ACTUAL igual a
                VALOR-MINIMAX(NODO,PROFUNDIDAD)
        2.2 Si VALOR-ACTUAL >= MAX-VAL,
                hacer MAX-VAL igual a VALOR-ACTUAL
3. Devolver MAX-VAL

FUNCION MINIMIZADOR(SUCESORES,PROFUNDIDAD)
1. Hacer MIN-VAL igual a *MAXIMO-VALOR*
2. Por cada NODO en SUCESORES,
        2.1 Hacer VALOR-ACTUAL igual a
                VALOR-MINIMAX(NODO,PROFUNDIDAD)
        2.2 Si VALOR-ACTUAL <= MIN-VAL,
                hacer MIN-VAL igual a VALOR-ACTUAL
3. Devolver MIN-VAL
```

- **Importante:** el proceso recursivo descrito implementa una *generación en profundidad* de los nodos del árbol de juego

Recorrido en profundidad del árbol de juego

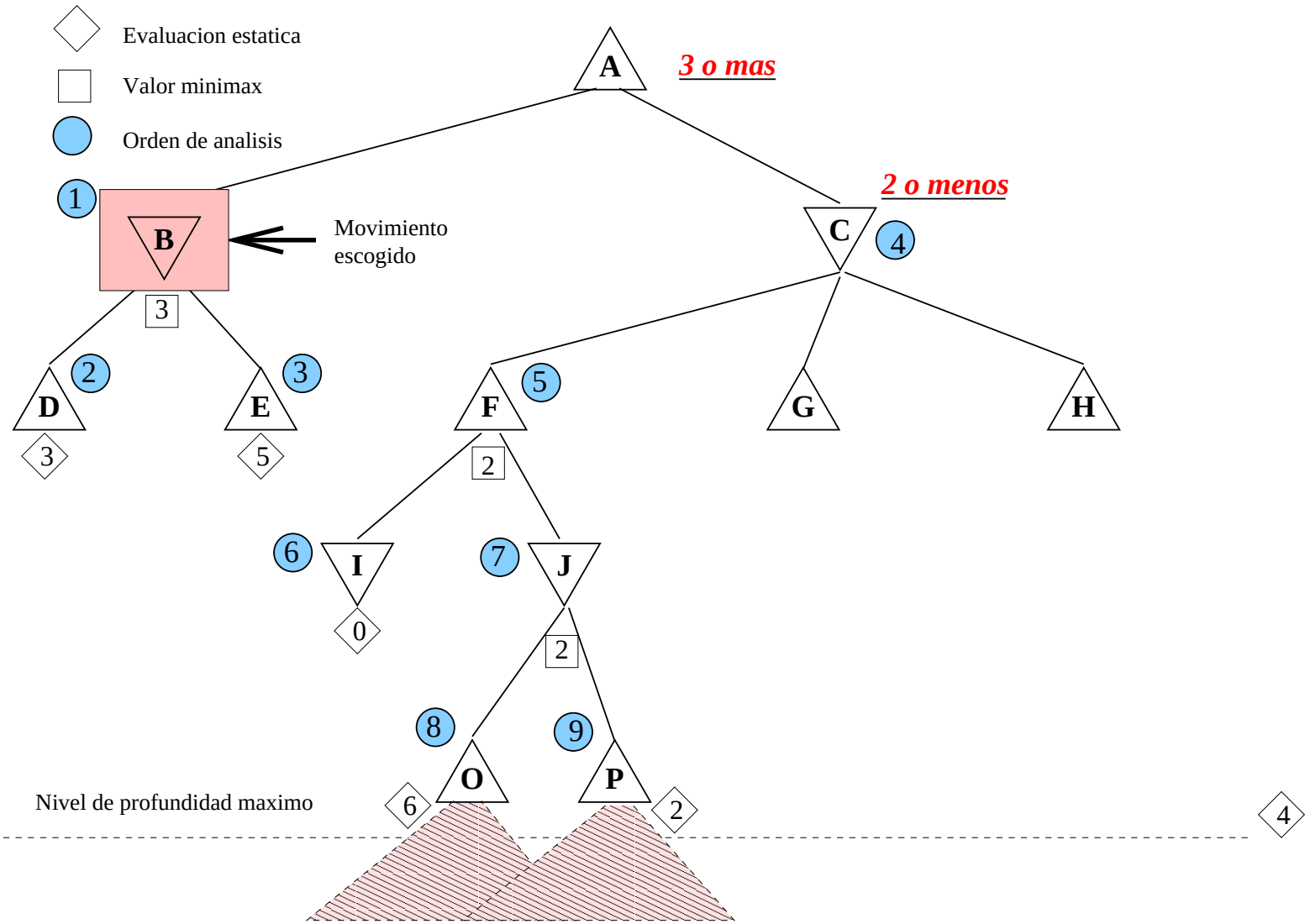


Complejidad de minimax

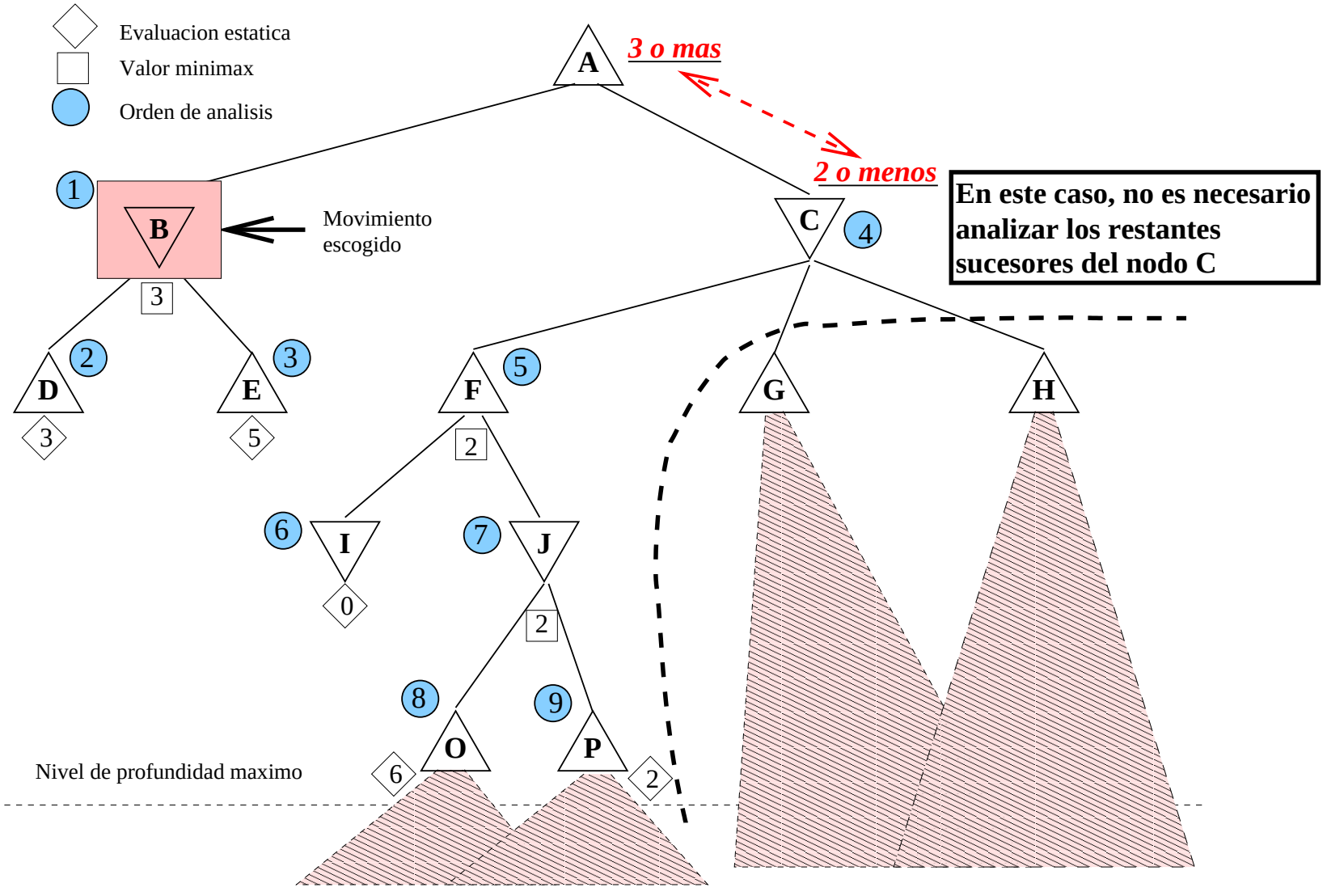
- Complejidad:
  - Sea r es el factor de ramificación y m el nivel de profundidad.
  - Complejidad en tiempo: $O(r^m)$.
  - Complejidad en espacio: $O(rm)$.
- Aún es ineficiente en la práctica:
  - En el ajedrez, con 150 segundos por turno, sólo será posible analizar $\sim$ 150000 nodos, luego sólo se podrá generar un árbol de profundidad menor que 4
  - Por tanto, la calidad del juego de la máquina sería muy baja (un jugador humano medio puede planificar al menos 6 jugadas por adelantado)
- La *poda alfa-beta* mejora :
  - En algunos casos, es posible podar el árbol *sin que afecte a la decisión que se toma finalmente*

Ejemplo de poda alfa-beta

- Situación justo después de analizar el subárbol de raíz F:



Ejemplo de poda alfa-beta



Minimax con poda alfa-beta

- Idea:
  - Cada nodo se analiza teniendo en cuenta el valor que *por el momento* tiene y el valor que *por el momento* tiene su padre
  - Esto determina en cada momento un intervalo (α, β) de posibles valores que podría tomar el nodo
- Significado intuitivo de α y β en cada momento:
  - Nodos MAX: α es el valor actual del nodo (que tendrá eso o más) y β es el valor actual del padre (que tendrá eso o menos)
  - Nodos MIN: β es el valor actual del nodo (que tendrá eso o menos) y α es el valor actual del padre (que tendrá eso o más)
- La poda se produce si en algún momento $\alpha \geq \beta$:
  - Y no hace falta analizar los restantes sucesores del nodo
  - En nodos MIN, se denomina poda β y en los nodos MAX, poda α

Implementación de minimax con poda alfa-beta

- Decisión minimax con poda alfa-beta:

```
FUNCION DECISION-MINIMAX-A-B(ACTUAL,PROFUNDIDAD)
1. Hacer ALFA igual a *MINIMO-VALOR*
2. Por cada NODO en SUCESORES(ACTUAL),
   2.1 Hacer VALOR-ACTUAL igual a
        VALOR-MINIMAX-A-B(NODO,PROFUNDIDAD-1,ALFA,*MAXIMO-VALOR*)
   2.2 Si VALOR-ACTUAL > ALFA,
        hacer ALFA igual a VALOR-ACTUAL
        y hacer MAX-NODO igual a NODO
   2.3 Si ALFA >= *MAXIMO-VALOR*,
        salir del bucle ;;; <--- poda alfa
3. Devolver MAX-NODO
```

- Cálculo del valor minimax con poda alfa-beta:

```
FUNCION VALOR-MINIMAX-A-B(NODO,PROFUNDIDAD,ALFA,BETA)
1. Si ES-ESTADO-FINAL(ESTADO(NODO)) o
      PROFUNDIDAD=0 o
      SUCESORES(NODO) igual a vacío,
      devolver F-E-ESTATICA(ESTADO(NODO),JUGADOR(NODO))
   si no, si JUGADOR(NODO)=MAX, devolver
      MAXIMIZADOR-A-B(SUCESORES(NODO),PROFUNDIDAD-1,ALFA,BETA)
   si no, devolver
      MINIMIZADOR-A-B(SUCESORES(NODO),PROFUNDIDAD-1,ALFA,BETA)
```

Implementación de minimax con poda alfa-beta

- Cálculo del máximos y mínimos:

```
FUNCION MAXIMIZADOR-A-B(SUCESORES,PROFUNDIDAD,ALFA,BETA)
2. Por cada NODO en SUCESORES,
    2.1 Hacer VALOR-ACTUAL igual a
         VALOR-MINIMAX-A-B(NODO,PROFUNDIDAD,ALFA,BETA)
    2.2 Si VALOR-ACTUAL > ALFA,
         hacer ALFA igual a VALOR-ACTUAL
    2.3 Si ALFA>=BETA, salir del bucle  ;;; <--- poda alfa
3. Devolver ALFA

FUNCION MINIMIZADOR-A-B(SUCESORES,PROFUNDIDAD,ALFA,BETA)
2. Por cada NODO en SUCESORES,
    2.1 Hacer VALOR-ACTUAL igual a
         VALOR-MINIMAX-A-B(NODO,PROFUNDIDAD,ALFA,BETA)
    2.2 Si VALOR-ACTUAL < BETA,
         hacer BETA igual a VALOR-ACTUAL
    2.3 Si ALFA>=BETA, salir del bucle   ;;; <--- poda beta
3. Devolver BETA
```

Complejidad de alfa-beta

- Complejidad:
  - Sea r es el factor de ramificación y m el nivel de profundidad.
  - Complejidad en tiempo: $O(r^{\frac{3m}{4}})$.
  - Complejidad en espacio: $O(rm)$.
- En la práctica, si los sucesores se exploran en orden de valor minimax (creciente o decreciente dependiendo de si es un nodo MIN o MAX, resp.), se produce la máxima poda, y en ese caso la complejidad temporal es $O(r^{\frac{m}{2}})$

Eficiencia y heurística de la poda alfa-beta

- Según lo anterior, en el mismo tiempo se pueden considerar hasta *el doble* de jugadas por adelantado que sin poda (haciendo a la máquina más competitiva)
  - Sin perder precisión
- Se pueden introducir heurísticas para intentar obtener la máxima poda
- Métodos heurísticos para generar los sucesores en el orden adecuado:
  - Usar el valor de la función de evaluación estática para ordenar
  - Usar búsqueda en profundidad iterativa y guardar los valores minimax calculados en cada iteración

Cuestiones técnicas

- Existen numerosas mejoras a las técnicas descritas, que permiten crear máquinas que juegan a nivel competitivo
- Decisiones sobre la profundidad (¿dónde parar la búsqueda?):
  - Efecto *horizonte*
  - Búsqueda de posiciones *quiescentes*
- Diseño de funciones de evaluación estática:
  - Combinación lineal de características
  - *Pesos* para ponderar la importancia de cada característica
- Aprendizaje automático:
  - Aprendizaje de los pesos, tras una serie de partidas consigo misma
  - Redes neuronales
- Biblioteca de movimientos (aperturas, finales,. . .)

3 en raya: ejemplo de juego

```
[12]> (load "juego.lsp")
...
[13]> (load "minimax.lsp")
...
[14]> (load "3-en-raya-2.lsp")
...
[15]> (inicializa-3-en-raya 'x)
(3-EN-RAYA. EMPIEZA X)
[16]> (juego :empieza-la-maquina? t
             :procedimiento '(minimax-a-b 6))

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
```

3 en raya: ejemplo de juego

```
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 5) (4)      (PON-FICHA-EN 6) (5)
      (PON-FICHA-EN 7) (6)      (PON-FICHA-EN 8) (7)
Tu turno: 1

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
```

3 en raya: ejemplo de juego

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

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

Jugador : MAX
Mi turno.

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

3 en raya: ejemplo de juego

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

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


Jugador : MAX
Mi turno.

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

Jugador : MIN
La maquina ha ganado
NIL
```

Bibliografía

- Russell, S. y Norvig, P. *Artificial Intelligence (A Modern Approach)* (Prentice–Hall, 2010). Third Edition
  - Cap. 5: “Adversarial search”.
- Russell, S. y Norvig, P. *Inteligencia Artificial (un enfoque moderno)* (Pearson Educación, 2004). Segunda edición)
  - Cap. 6: “Búsqueda con adversario”.

Bibliografía

- Winston, P.R. *Inteligencia artificial (3a. ed.)* (Add–Wesley, 1994).
  - Cap. 6: “Arboles y búsqueda con adversario”.
  - Cap. 4.7.1: “Juegos: la estrategia minimax y sus modificaciones”.
- Rich, E. y Knight, K. *Inteligencia artificial (segunda edición)* (McGraw–Hill Interamericana, 1994).
  - Cap. 12: “Los juegos”.