PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

INTELIGENCIA ARTIFICIAL (INF371)

BUSQUEDA ADVERSARIAL (JUEGOS)

Dr. Edwin Villanueva Talavera

Contenido

- Busqueda Adversarial
  - Juegos
  - Algoritmo MINIMAX
  - Algoritmo ALPHA-BETA

Bibliografía:

Capitulo 5.1-5.3 del libro:

Stuart Russell & Peter Norvig "Artificial Intelligence: A modern Approach", Prentice Hall, Third Edition, 2010

Juegos

- Los algoritmos de búsqueda estudiados hasta ahora son apropiados para implementar agentes que actuarán en ambientes sin interacción con otros agentes y que poseen total control de sus acciones y de sus efectos
- En entornos multiagentes, donde las acciones de los agentes afectan los objetivos de los otros agentes (entornos conocidos como **juegos**) se necesita otro tipo de algoritmos para guiar la toma de decisiones de los agentes : **algoritmos de búsqueda adversarial**
- Un tipo común de juego que abordaremos son los llamados **juegos de suma-zero** (*zero-sum games*), donde la suma de la utilidad de los agentes (**jugadores**) al final del juego es constante

Juegos

Formulación del Juego:

- Estado Inicial S_0: Situación del juego al inicio
- Player*(s)*: Define que jugador tiene la movida en un estado dado *s*
- Actions*(s)*: Retorna el conjunto de movidas válidas a partir del estado *s*
- Result*(s,a)*: Función sucesora (o modelo de transición). Define a que estado se arriba aplicando acción *a* al estado *s*
- Terminal-Test*(s)*: Determina si estado *s* es un estado final de juego (estado terminal)
- Utility*(s,p)*: Función de utilidad. Retorna el valor de utilidad que recibe jugador *p* al final del juego finalizado en estado terminal *s*

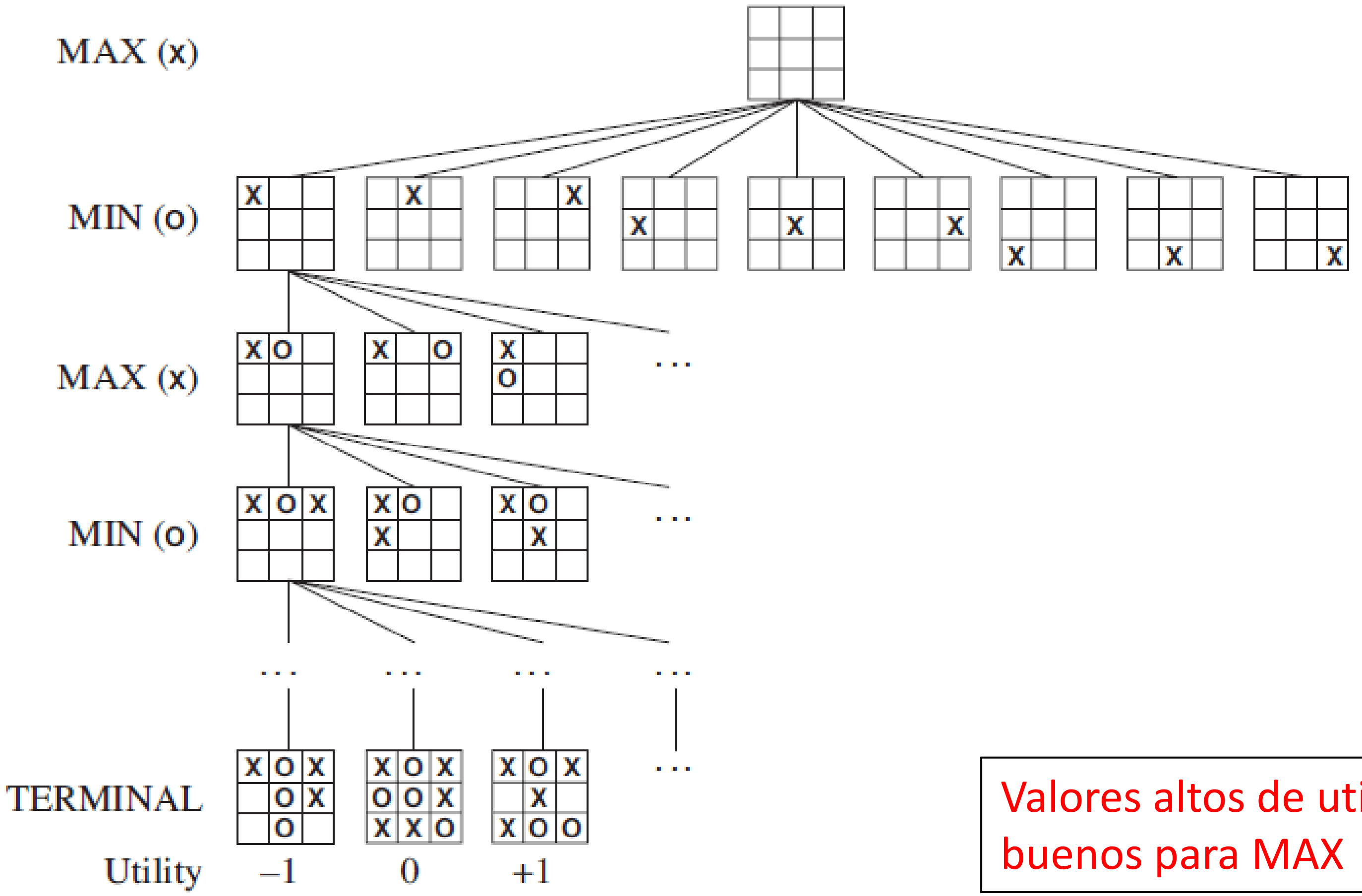
Juegos

Consideraciones en juegos de dos jugadores:

- El agente que nos toca programar lo llamamos MAX, al oponente lo llamamos MIN.
- MAX hace el primer movimiento, turnándose con MIN hasta el final
- La movida del oponente MIN es "imprevisible": MAX tiene que considerar todos los movimientos posibles del MIN
- Al final del juego, cada jugador recibe un valor de utilidad que refleja su estado al final del juego (ej. ganador, perdedor, empate)

Arbol de Juego (2 jugadores):

- Arbol donde los nodos son estados del juego y aristas son movidas. Nodo raiz es el estado inicial del juego

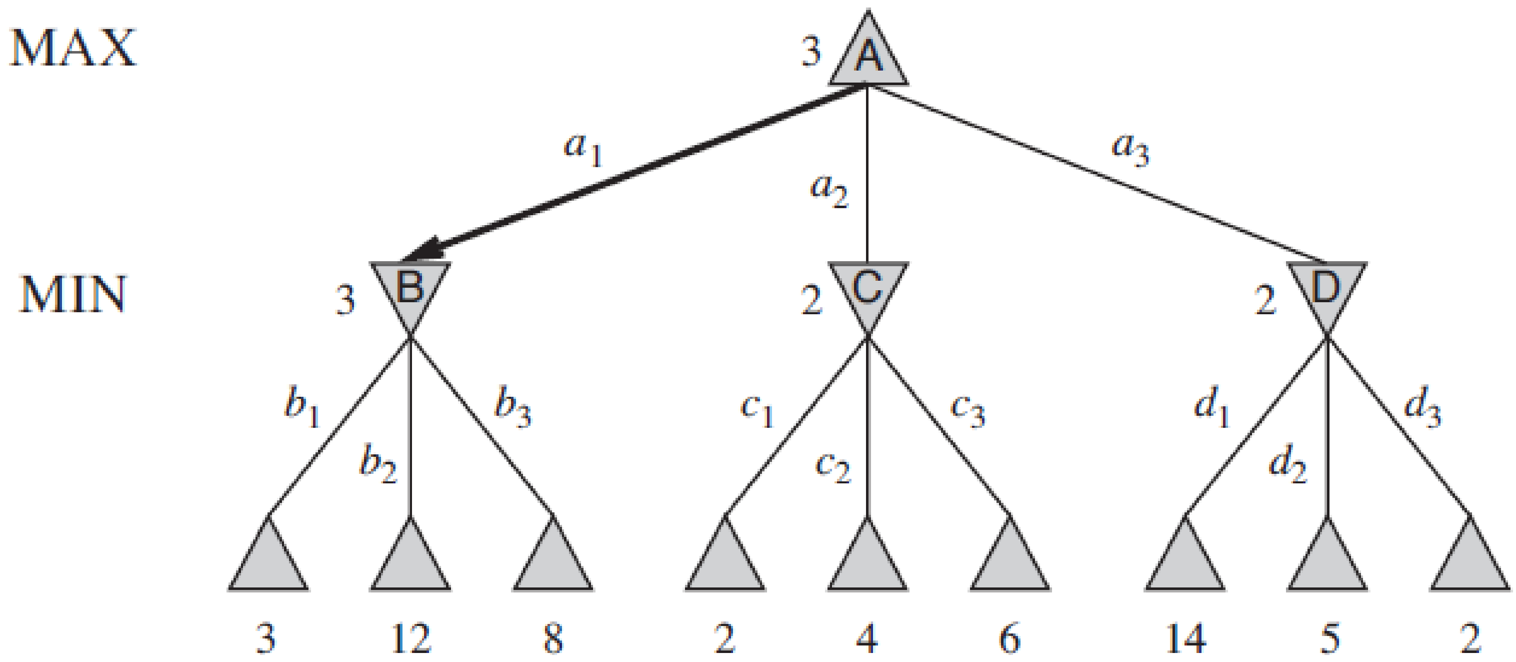


Valores altos de utilidad son buenos para MAX

Decisiones Óptimas en Juegos (2 jugadores):

- La solución óptima para MAX seria uma sequencia de movidas que MAX haría para llegar a un estado terminal donde el sea ganador
- MAX debe encontrar una **estrategia** de contingencia la cual debe especificar el movimiento en el estado inicial y luego el movimiento en los estados resultantes de cada posible movimiento de MIN y así sucesivamente
- Para encontrar esta estratégia se asume que el oponente MIN hace sus movimientos de forma óptima (queriendo llegar a un estado terminal donde MIN es el ganador)

Decisiones Óptimas en Juegos (2 jugadores):



Algoritmo MINIMAX

Decisiones Óptimas en Juegos (2 jugadores): MINIMAX

- Dado un árbol de juego, la estrategia óptima puede ser determinada a partir del **valor Minimax de cada nodo:**

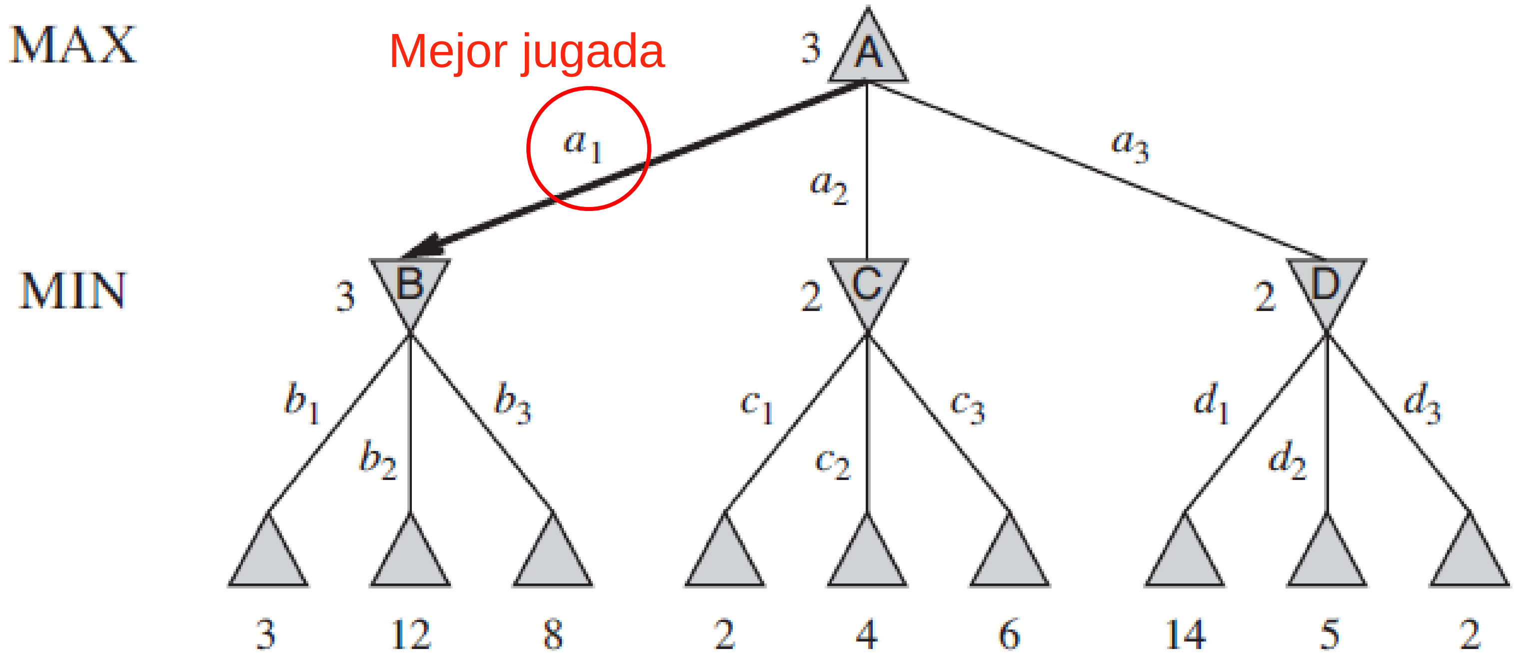
$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

- El valor Minimax(s) (**para MAX**) es la utilidad de MAX de estar en estado s asumiendo que MIN escogerá los estados más ventajosos para el mismo hasta el final del juego (es decir, los estados con menor valor de utilidad para MAX)

Algoritmo MINIMAX

Ejemplo de cálculo del valor de MINIMAX:

- Cada jugador hace un solo movimiento



Algoritmo MINIMAX

PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

```
function MINIMAX-DECISION(state) returns an action
   return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for each a in ACTIONS(state) do
      v ← MAX(v, MIN-VALUE(RESULT(s, a)))
   return v

function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for each a in ACTIONS(state) do
      v ← MIN(v, MAX-VALUE(RESULT(s, a)))
   return v
```

Algoritmo MINIMAX

Propiedades del algoritmo MINIMAX:

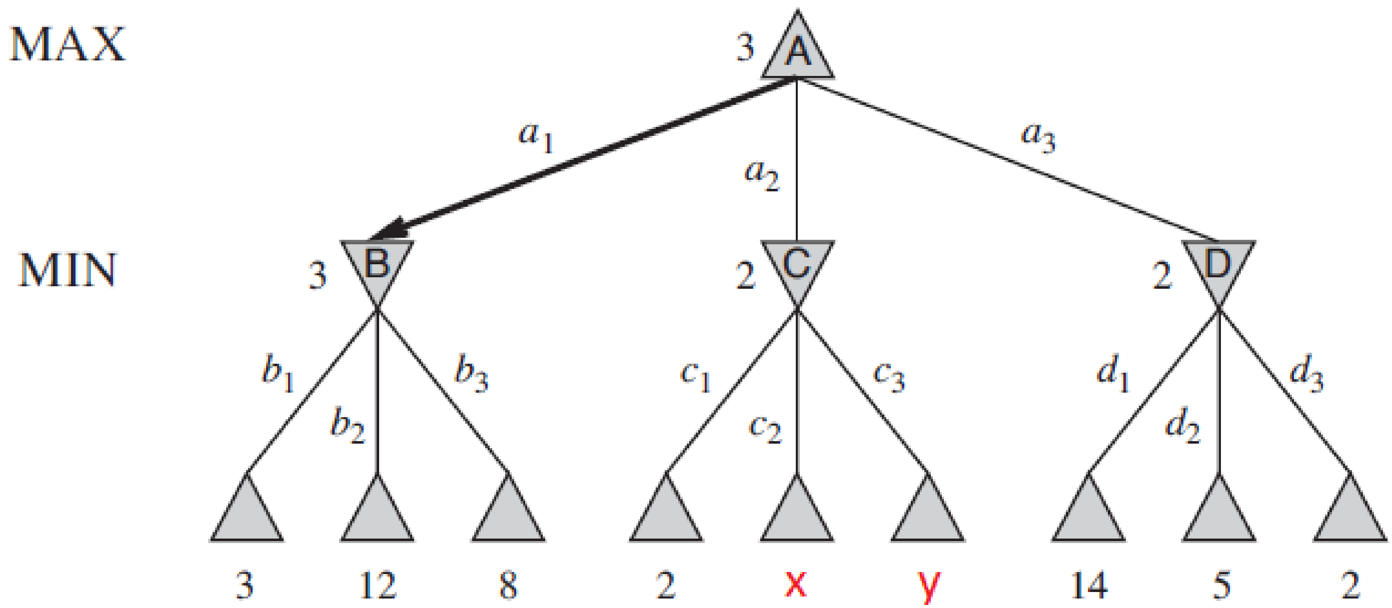
- Equivale a una búsqueda completa en profundidad en el árbol del juego.
  - m: profundidad máxima del árbol
  - b: movimientos válidos en cada estado

- Completo? Si (Si el árbol es finito)
- Óptimo? Si
- Complejidad de tiempo? $O(b^m)$
- Complejidad de espacio? $O(bm)$

Para ajedrez, $b \approx 35$, $m \approx 100$ para juegos "razonables"
→ solución exacta no es posible

Algoritmo ALPHA-BETA

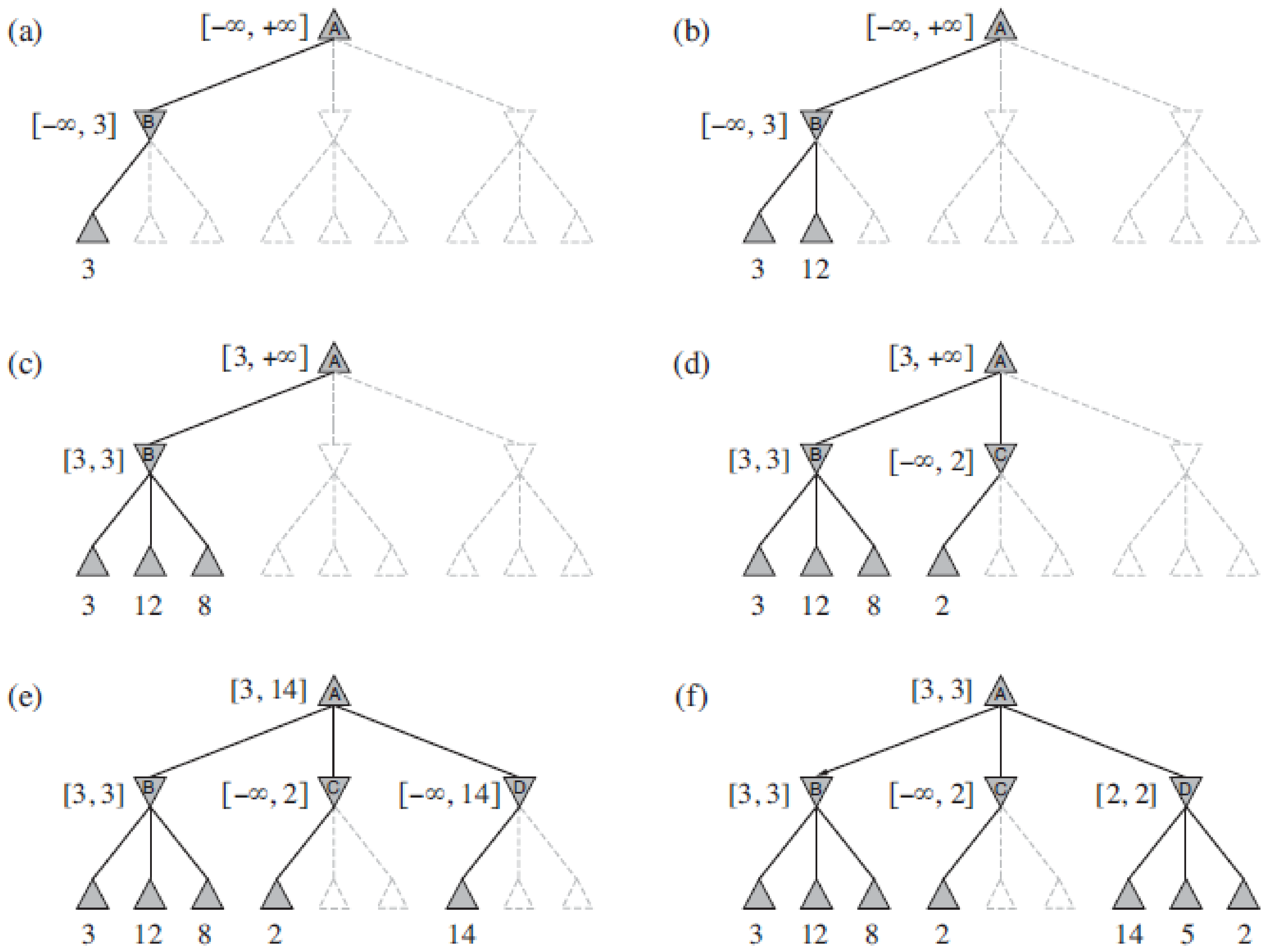
- Surge como una forma de aliviar la complejidad temporal de MINIMAX, la cual es exponencial en la profundidad del árbol
- Se basa en el hecho de que decisiones optimas pueden ser tomadas evitando explorar ramas que no influirían en la decisión final
- Ejemplo:



  - No se necesita saber los valores de x, y para tomar la decisión en A

Algoritmo ALPHA-BETA

- Ejemplo:



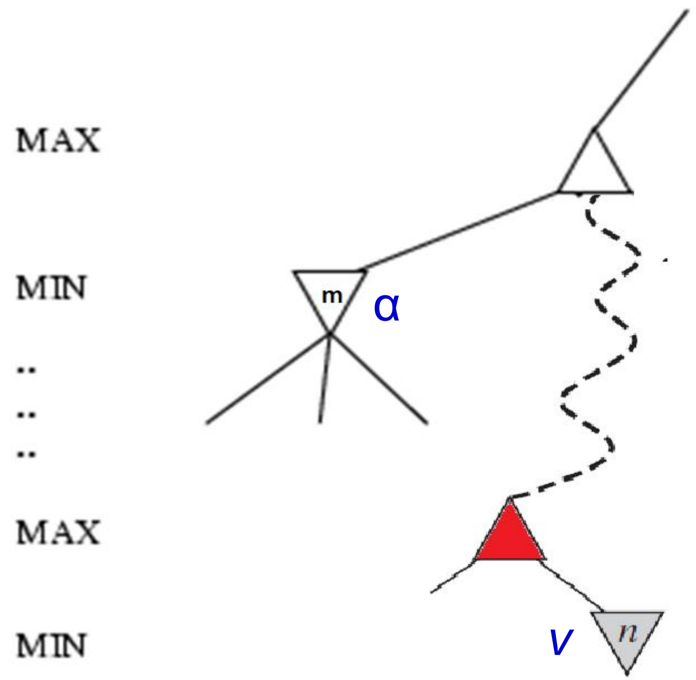
Algoritmo ALPHA-BETA

Implementación Poda α-β :

Se mantiene valores [α, β]:

- α es el mayor valor encontrado hasta ahora para cualquier punto de elección de MAX
- β es el menor valor encontrado hasta ahora para cualquier punto de elección de MIN

Si en algun momento explorando las ramas de n se tiene que v (mayor valor minimax para n) es menor a α entonces poda el subarbol n



Algoritmo ALPHA-BETA

function ALPHA-BETA-SEARCH($state$) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Algoritmo ALPHA-BETA

Orden de examinación de nodos en Poda α-β :

- La efectividad de la poda α-β depende del orden en que los sucesores son examinados
- Con el mejor orden posible la complejidad de tiempo es $O(b^{m/2})$ (puede resolver um arbol del doble de profundidad al mismo costo)
- Usando orden aleatoria se espera explorar $O(b^{3m/4})$
- Usando conocimiento a priori de juegos pasados puede ayudar a escoger un orden (ej. Primero capturar pieza, luego amenazar y luego avanzar)
- Esquemas con tablas de transposición pueden mejorar el costo en arboles con estados repetidos (en juegos donde diferentes secuencias de movidas pueden generar el mismo estado). Similar a Explored Set.

Monte-Carlo Tree Search (MCTS)

- Minimax necesita explorar todo el árbol del juego para decidir cada movida, lo cual lo limita a juegos simples, con bajo factor de ramificación y arboles poco profundas
- Versiones de Minimax con heurísticas son difíciles de implementar en la practica, ya que se requiere definir una heurística que se requiere um conocimiento profundo del juego
- Búsqueda con árboles Monte-Carlo (propuesto en 2007 [1]) consigue lidiar eficientemente con juegos complejos, desde que no se desarrolla todo el arbol del juego, sólo las partes que ofrecen mejores chances de ganar
- MCTS es un método anytime, puede ser parado en cualquier momento y obtener una movida aceptable y mejora con el tiempo

Monte-Carlo Tree Search (MCTS)

- MCTS realiza ciclos repetidos compuestos de 4 fases:

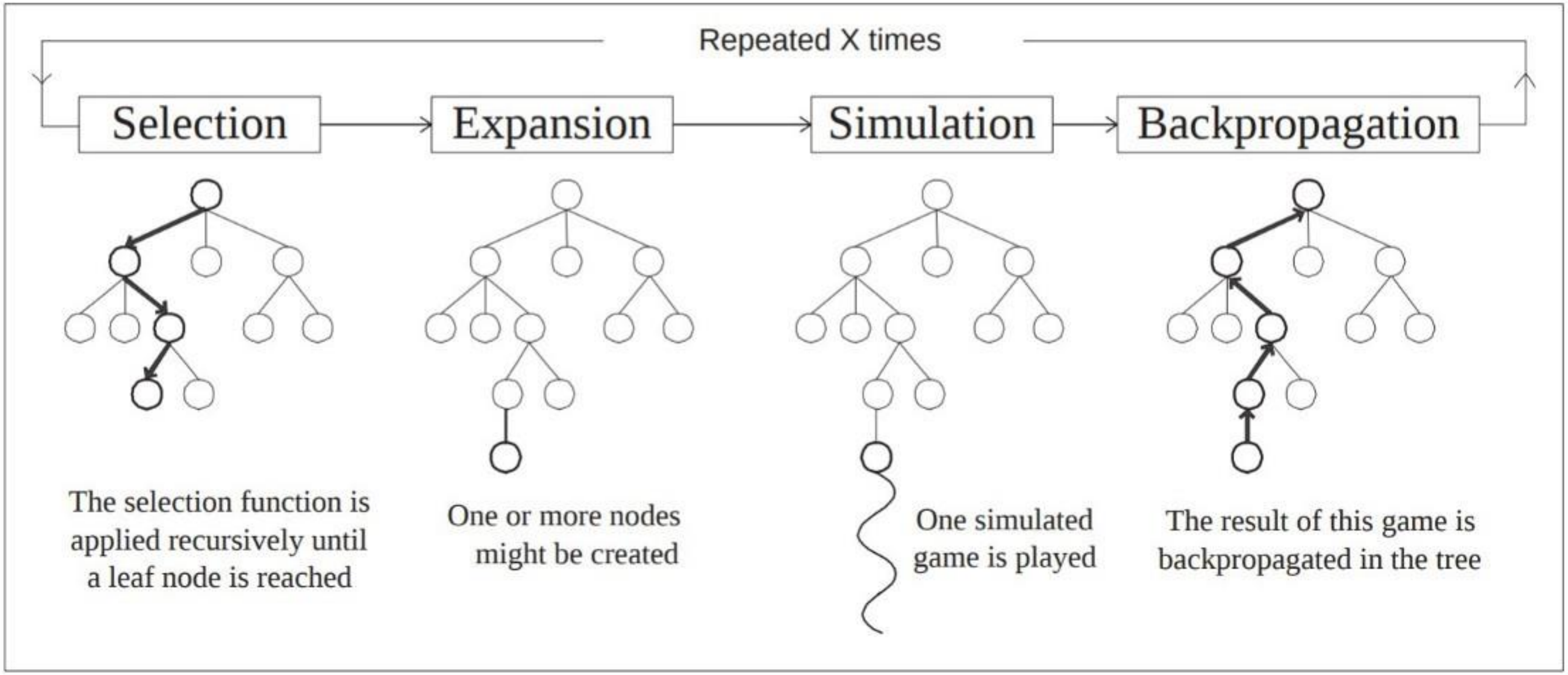


Diagrama tomado del paper de Chaslot [1]

Monte-Carlo Tree Search (MCTS)

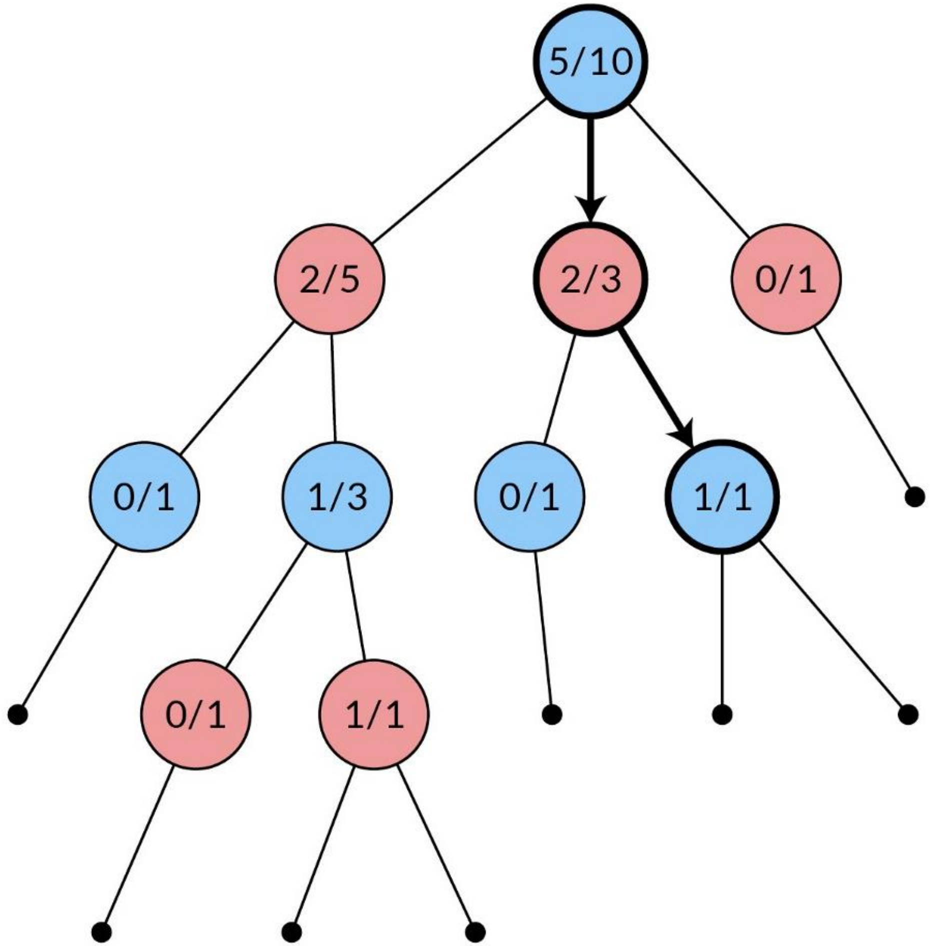
Fase de Selección

- Cada nodo (game state) mantiene 2 estadísticas:
  - **N**: # de simulaciones envolviendo al nodo
  - **U**: # de simulaciones (envolviendo al nodo) ganadas por el jugador que generó el nodo
- Comenzando en el nodo raíz se va seleccionando cada vez el nodo hijo con el mayor valor UCB1 hasta que se encuentre un nodo hoja (sin hijos)

$$\mathrm{UCB1} = U/N + c\sqrt{\ln(N^{pa})\,/N}$$

Incentiva explotacion

Incentiva exploracion



N^{pa} es el valor N del nodo padre. c es normalmente $\sqrt{2}$

Diagrama tomado del [2]

Monte-Carlo Tree Search (MCTS)

Fase de Expansión

- Se genera un (o varios) nodo hijo del nodo hoja seleccionado en la fase anterior. Los valores N y U de los hijos generados serán 0.
- Si se generan varios nodos hijos se escoge uno aleatoriamente

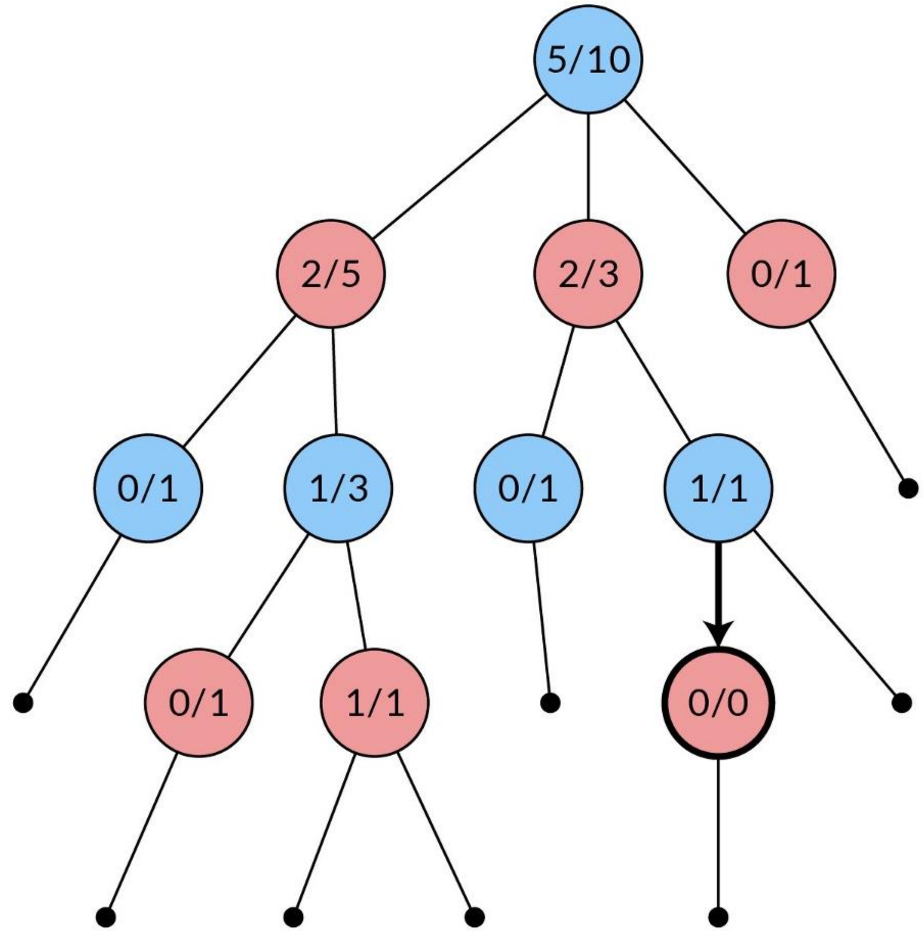


Diagrama tomado del [2]

Monte-Carlo Tree Search (MCTS)

Fase de Simulación

- Continuando desde el nuevo nodo generado (s) en la fase de expansión se realiza una simulación del juego hasta llegar a un estado terminal:

```
p = Player(s)
while not Terminal_test(s):
    a = random.choice (Actions(s))
    s = Result(s, a)
v = Utility(s, p)
```

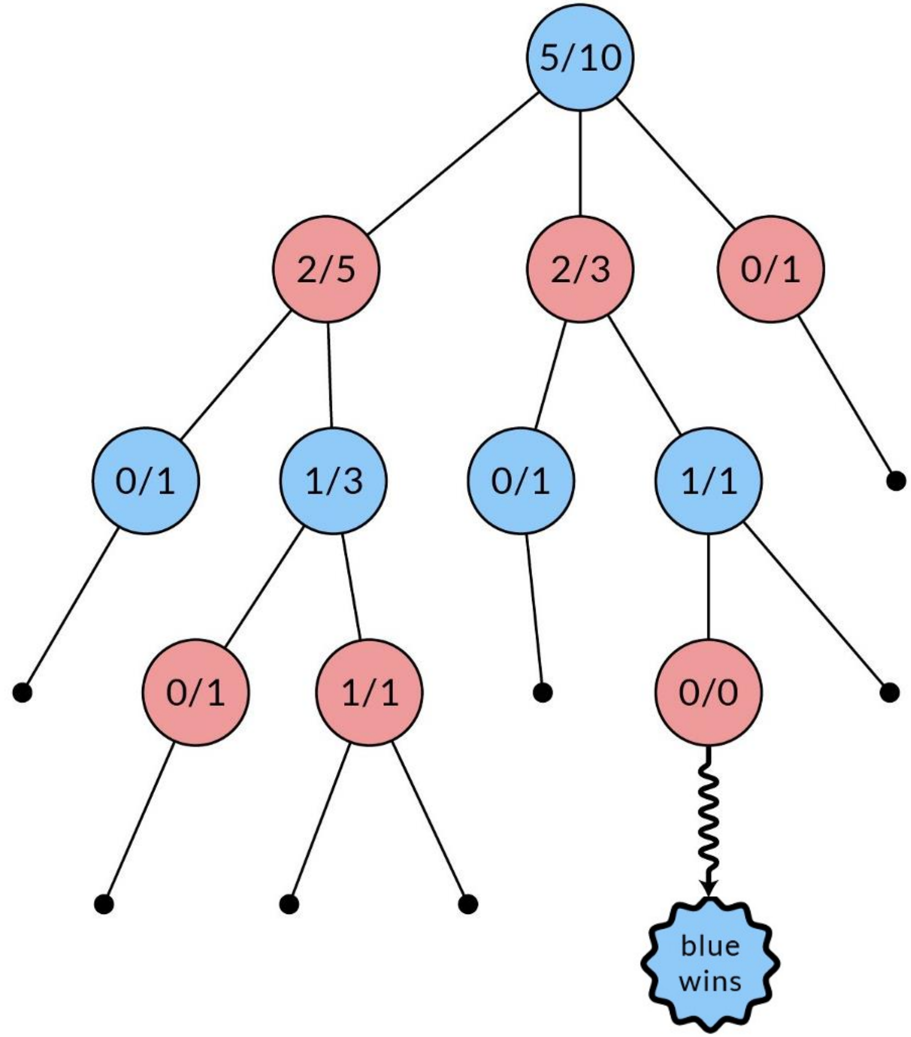


Diagrama tomado del [2]

Monte-Carlo Tree Search (MCTS)

Fase Backpropagation

- Las estadísticas de todos los nodos visitados de la simulación son actualizadas

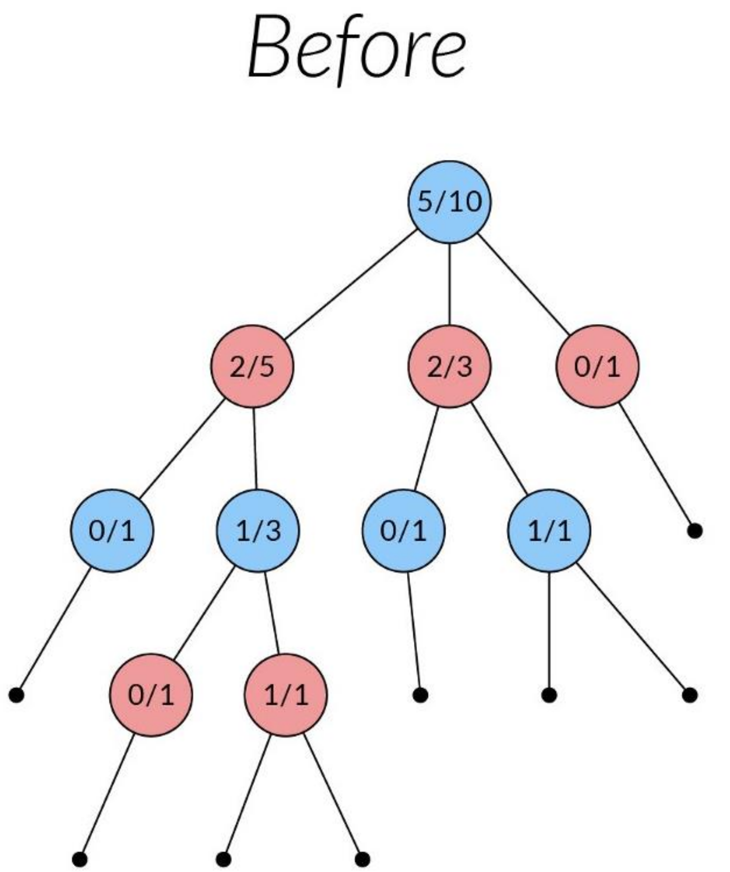
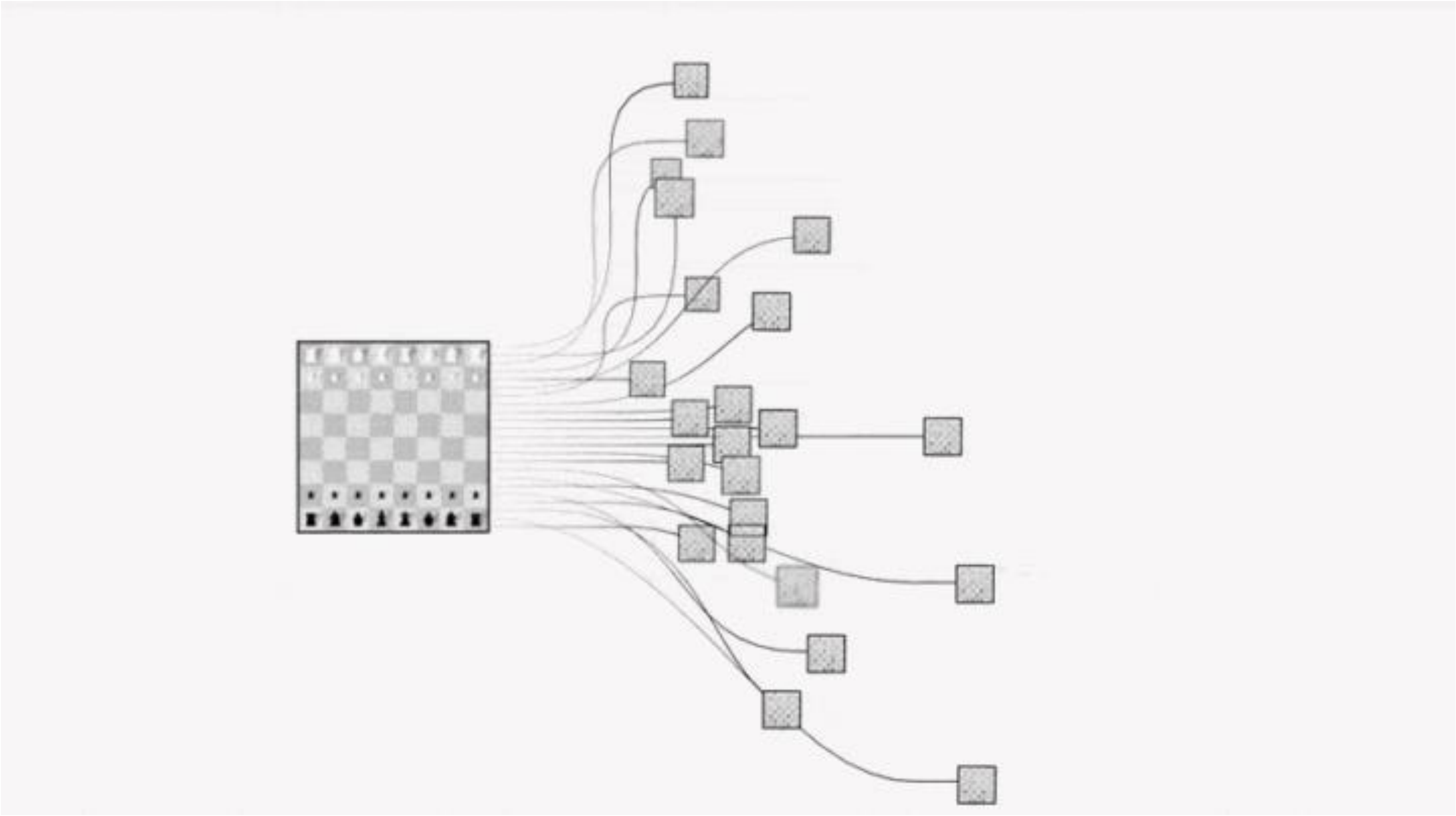


Diagrama tomado del [2]

Monte-Carlo Tree Search (MCTS)

Animacion del comportamiento de MCST



Animación tomado del [3]

Referencias

- [1] Chaslot et al. (2008) Monte-Carlo Tree Search: A New Framework for Game AI. In: Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'08). Pages 216–217
- [2] https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238
- [3] http://www.cs.us.es/~fsancho/?e=189

Preguntas?