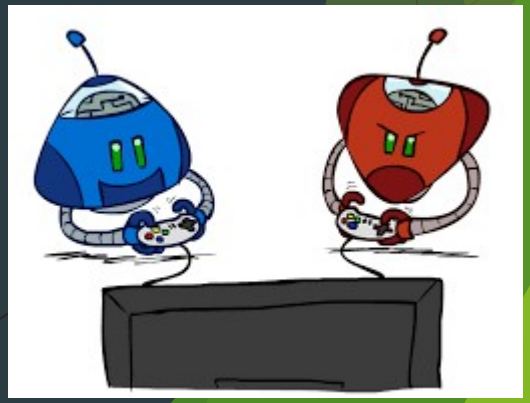
Técnicas de Inteligencia Artificial

Tema 3.2

Búsqueda adversaria en juegos

Búsqueda adversaria

- Los métodos de búsqueda anteriores aplicaban para un único agente que trataba de encontrar una solución óptima en forma de secuencia de acciones.
- En la búsqueda adversaria dos agentes realizan acciones de manera alternativa.
- Los valores de utilidad para cada agente son el opuesto del agente contrario.
- Las búsquedas en las que **dos o más jugadores** con **objetivos en conflicto** intentan explorar el mismo espacio para obtener la solución se llaman **búsquedas adversarias**, también conocidas como **juegos**.
- Los juegos son modelados como un problema de búsqueda y una función heurística de evaluación, que son los dos aspectos clave para resolverlos.



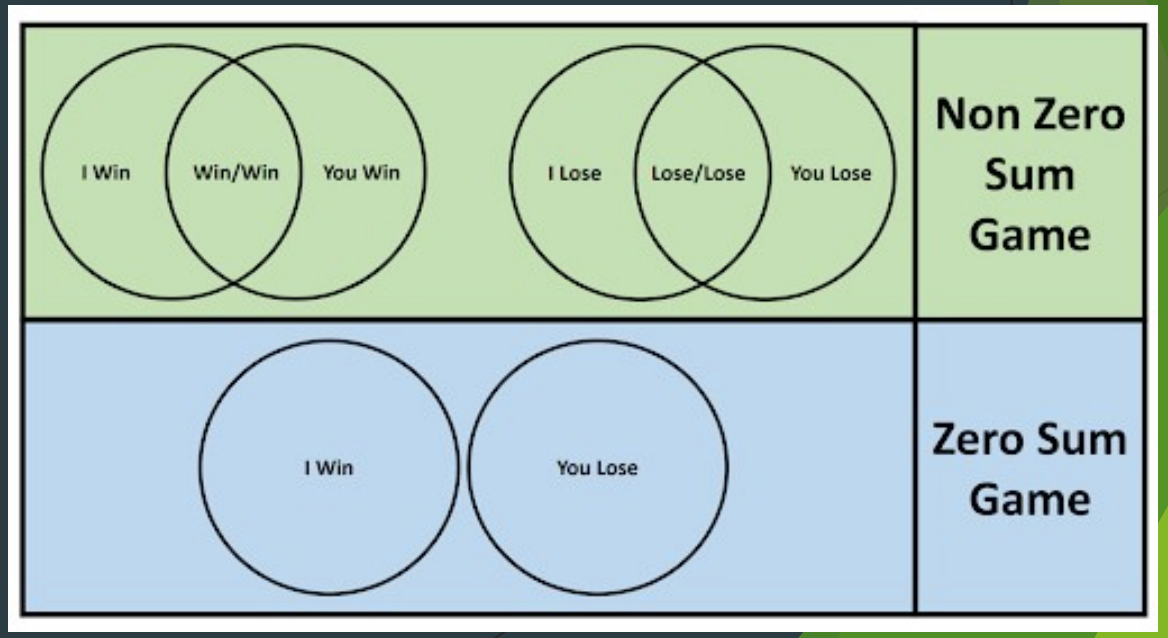
Tipos de juegos en IA

- **Información perfecta**. Los agentes tienen acceso completo al tablero, toda la información del juego y pueden ver el movimiento de sus oponentes.
- **Información imperfecta**. Los agentes no tienen acceso a toda la información sobre el juego.
- **Juegos deterministas**. Siguen un patrón rígido y un conjunto de reglas, sin elementos de azar.
- **Juegos no deterministas**. Contienen una serie de elementos impredecibles y un factor de suerte, en forma de dados o cartas.

	Determinista	**Probabilístico**
Información perfecta	Ajedrez, damas, go	Backgammon, monopoly
Información imperfecta	Hundir la flota	Bridge, poker

Juegos de suma cero

- Los juegos de suma cero son búsquedas adversarias donde solo hay un ganador.
- Cada ganancia o pérdida de utilidad de un agente en un juego de suma cero está exactamente balanceado por las pérdidas o ganancias de otro actor.
- Un jugador trata de maximizar un valor, mientras que el otro trata de minimizarlo.
- Los jugadores deben encontrar estrategias contingentes. Una estrategia óptima asume un oponente infalible.

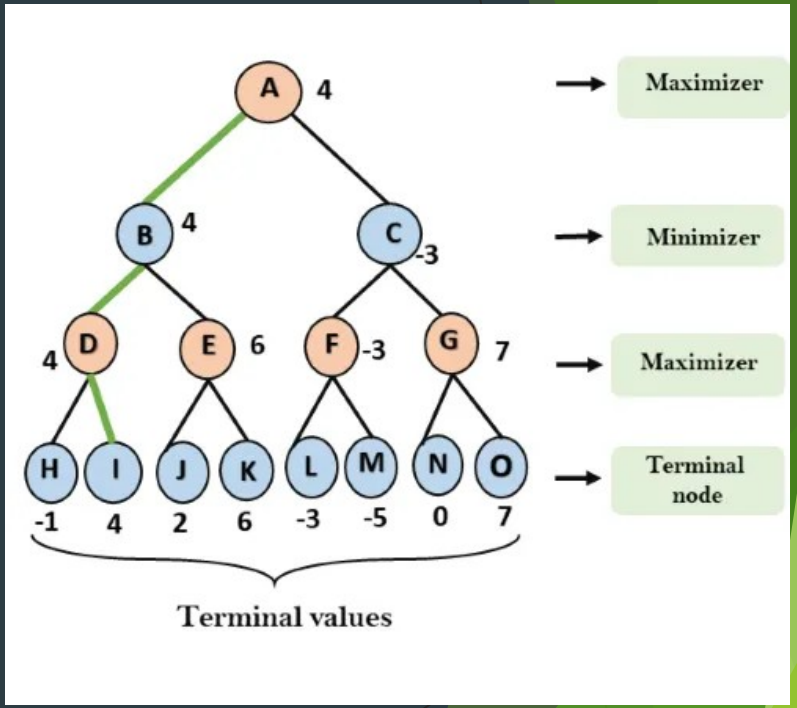


Juegos de suma cero

- Formalización del problema:
  - **Estado inicial**. Configuración inicial de la partida.
  - **Jugador**(s). Indica que jugador ha movido en el espacio de estados.
  - **Acción**(s). Devuelve el conjunto de movimientos permitidos en el espacio de estados como acción.
  - **Resultado**(s, a). El modelo de transición define el resultado de los movimientos en el espacio de estados.
  - **Test terminal**(s). Si el juego ha terminado, el terminal es verdadero; si no, es falso. Se trata de estados en los que el juego finaliza.
  - **Utilidad**(s,p). El valor numérico para una partida que finaliza en un estado terminal s para un jugador p es devuelto por la función de utilidad.

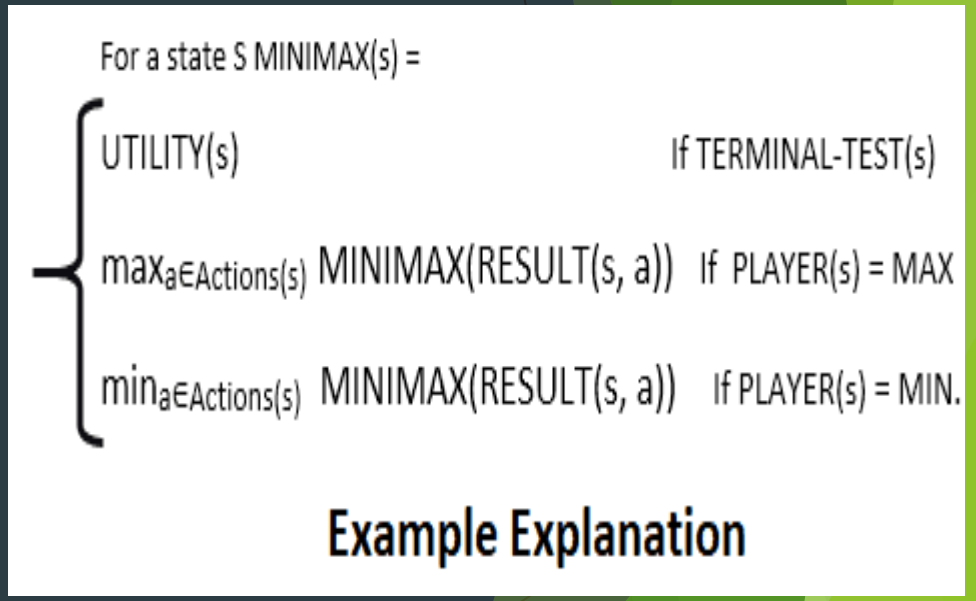
Algoritmo minimax

- Tipo de algoritmo de backtracking para encontrar el mejor movimiento para un jugador, asumiendo que el oponente juega de forma óptima.
- Especialmente útil cuando toda la información del juego está disponible.
- Cada movimiento es representado en términos de pérdida o ganancia para uno de los jugadores.
- Usado en juegos de suma cero.
- Se conoce como ply un movimiento individual hecho por un jugador.



Algoritmo minimax

- Descripción del algoritmo:
  - Dos jugadores: MAX y MIN.
  - Los jugadores tienen turnos alternativos empezando por MAX.
  - MAX maximiza el resultado del árbol de juego.
  - MIN minimiza el resultado.
  - Generalmente, utiliza una estrategia de búsqueda en profundidad.
  - El nodo hoja terminal puede aparecer en cualquier nivel del árbol.
  - Los valores de minimax deben ser propagados por el árbol hasta encontrar el nodo terminal.

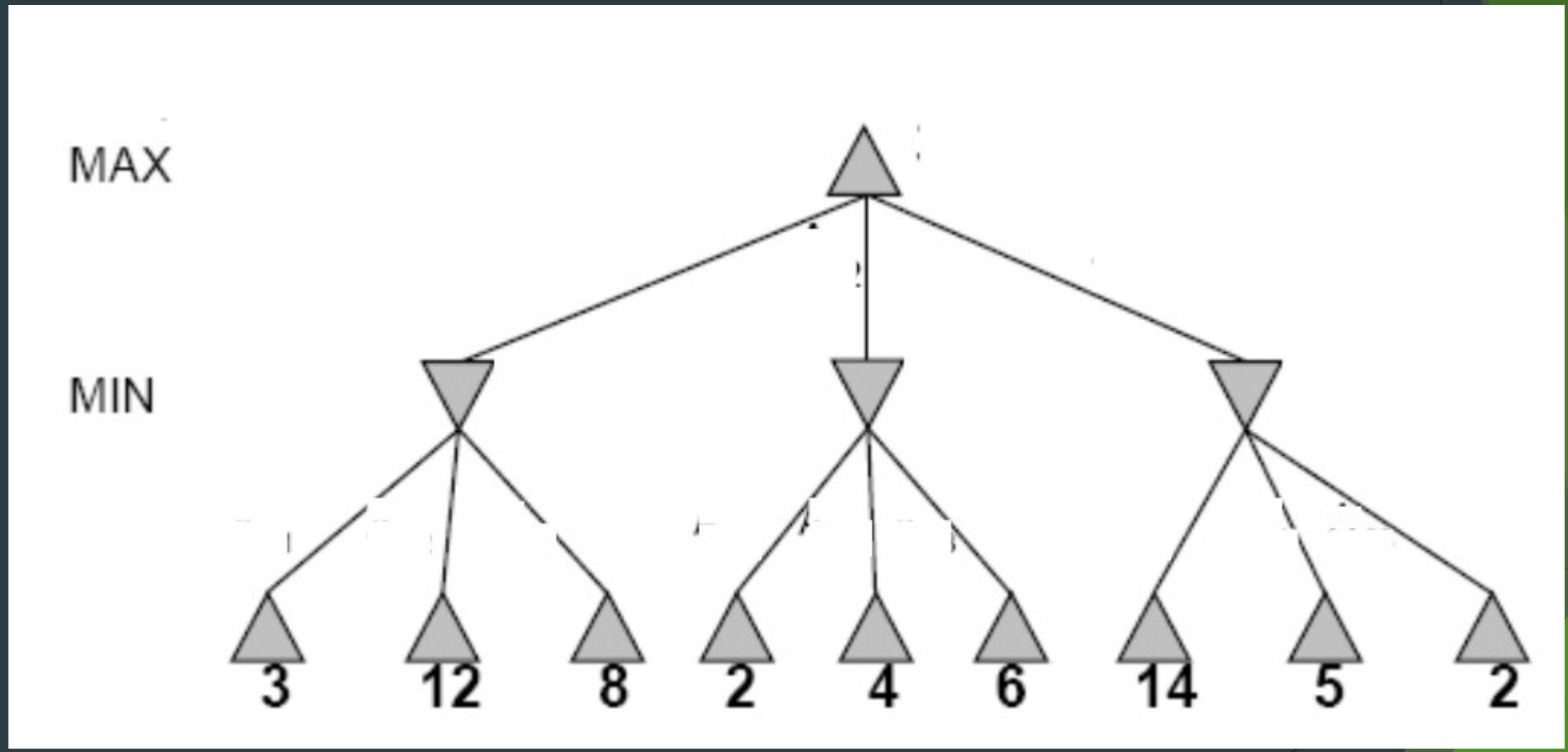


Algoritmo minimax

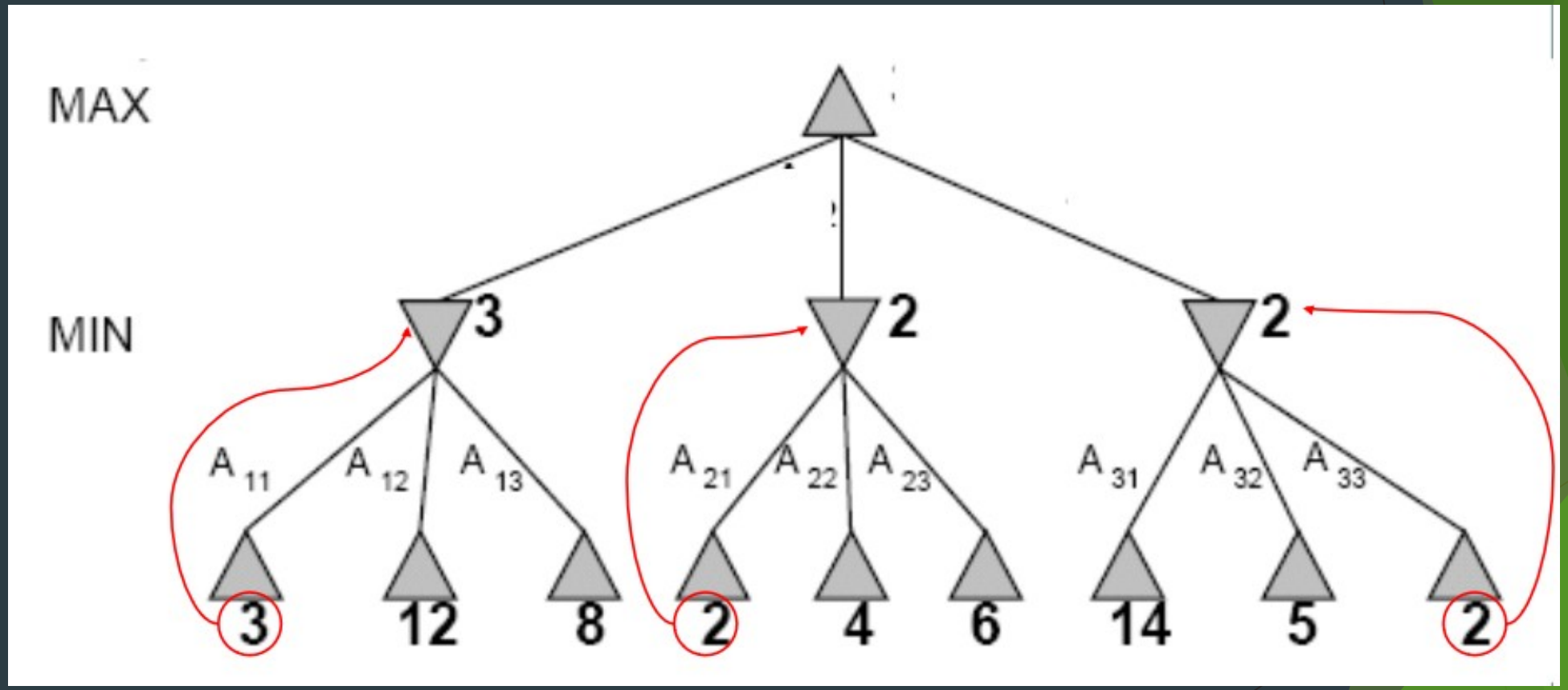
Algorithm 1: Recursive Minimax

Data: Starting state node S
Function `Minimax`$(S,\ Maximizing = True)$**:**
    if S *is terminal* **then**
        return Utility(S)
    if $Maximizing = True$ **then**
        $v \leftarrow -\infty$;
        for ***each*** *child* ***in*** S **do**
            $v \leftarrow Max(v, Minimax(child, False))$;
        end
        return v;
    else
        $v \leftarrow +\infty$;
        for ***each*** *child* ***in*** S **do**
            $v \leftarrow Min(v, Minimax(child, True))$;
        end
        return v;
    end

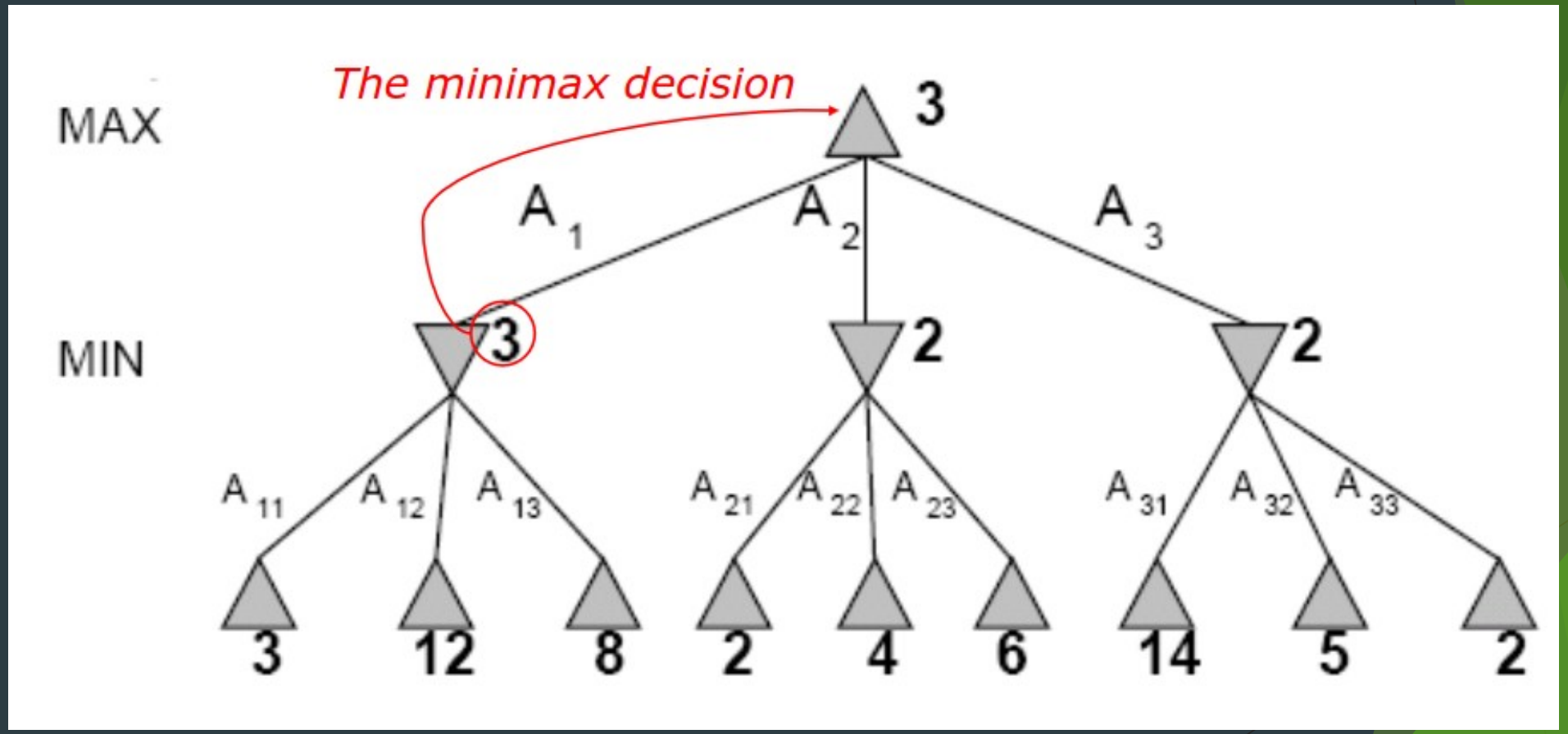
Algoritmo minimax



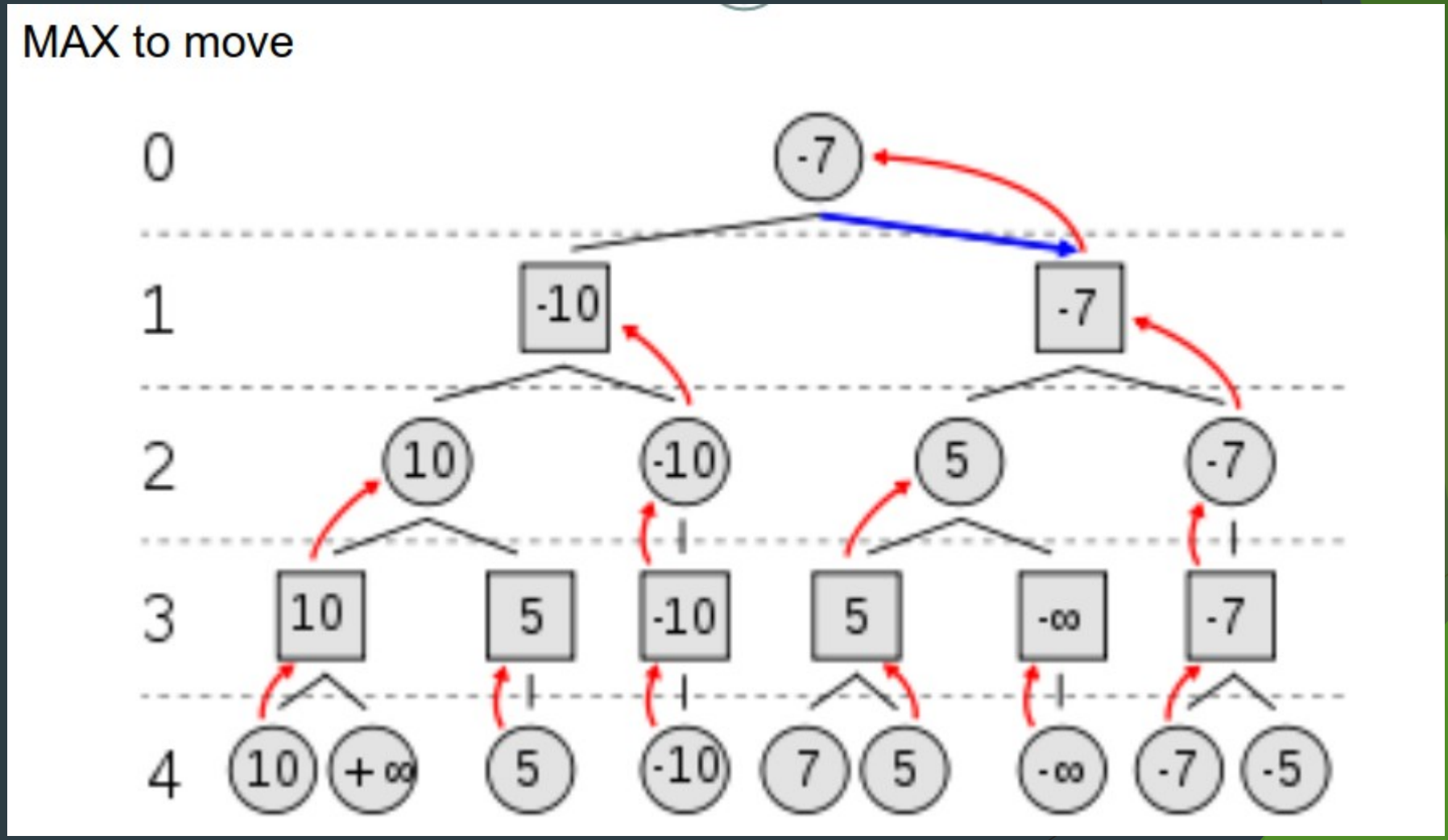
Algoritmo minimax



Algoritmo minimax



Algoritmo minimax

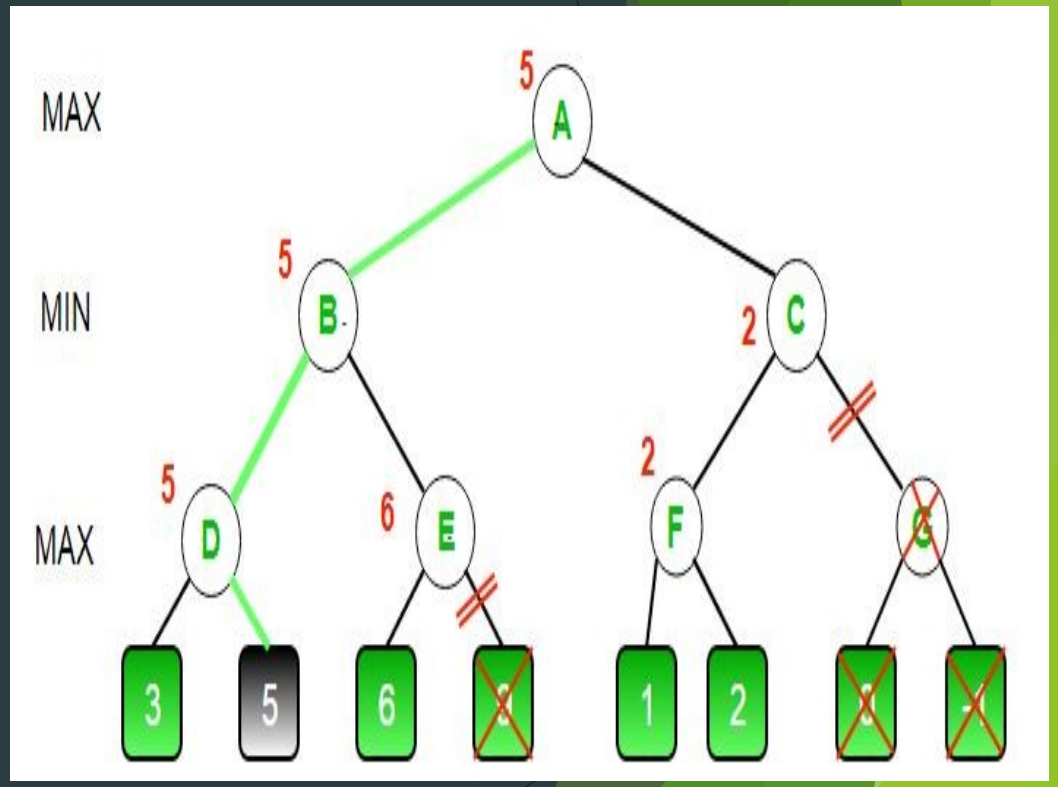


Algoritmo minimax

- El algoritmo original tiene una serie de limitaciones:
  - Se necesita llegar a un nodo terminal para generar una evaluación de utilidad.
  - El desarrollo del árbol de búsqueda puede implicar muchos niveles de profundidad, pudiendo ser potencialmente niveles infinitos en caso de bucles.
  - La evaluación y búsqueda de todos los nodos posibles y ramas degrada el rendimiento y la eficiencia del sistema.
  - Si hay restricciones de tiempo y espacio, no es posible explorar el árbol completo y obtener una solución en ciertas situaciones.

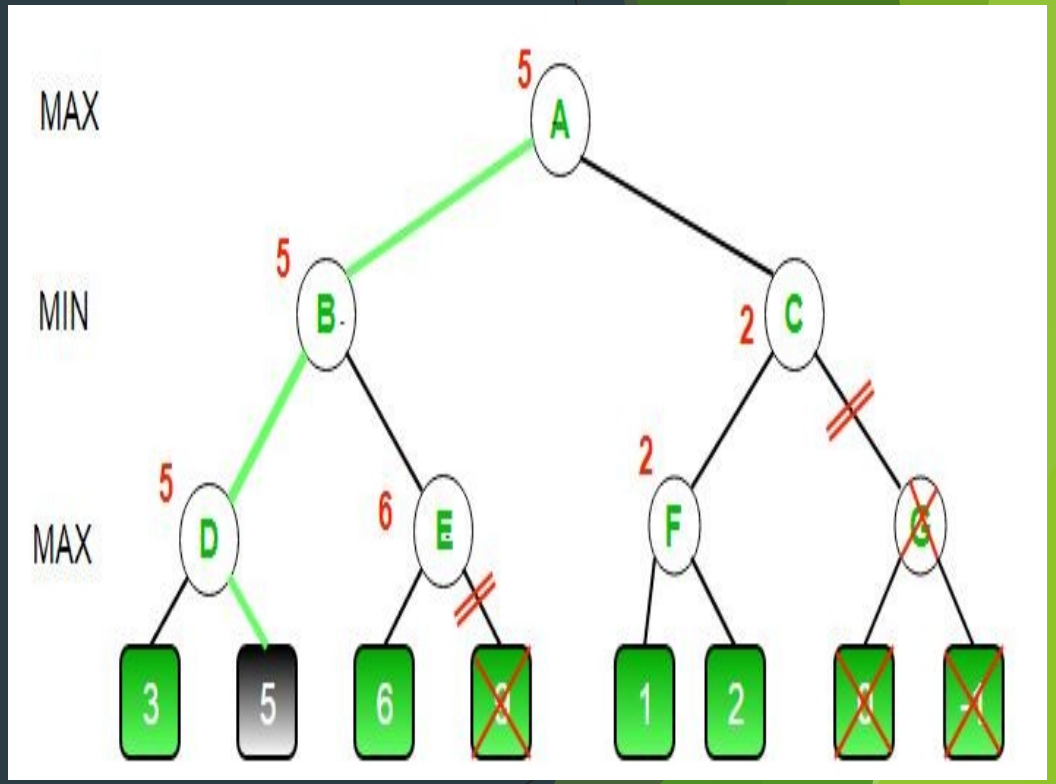
Poda alfa-beta

- Reduce el número de nodos que son evaluados por el algoritmo minimax.
- Deja de evaluar un movimiento cuando se ha descubierto al menos una posibilidad que demuestra que el movimiento actual es peor que uno previamente examinado.
- Cuando se aplica a un árbol minimax estándar devuelve el mismo movimiento que el algoritmo original, pero evaluando muchos menos movimientos.



Poda alfa-beta

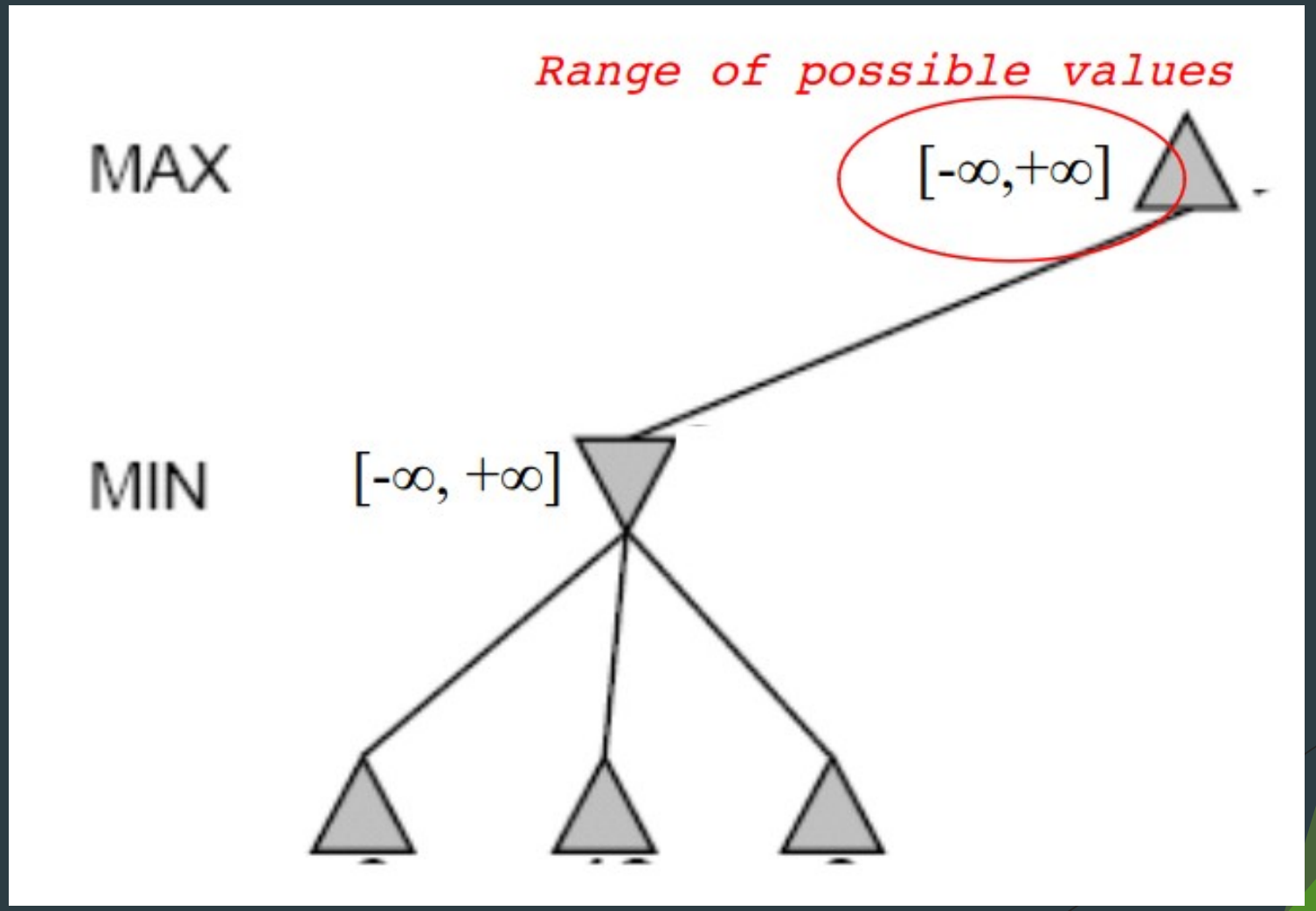
- Desarrollo del algoritmo:
  - α es el mayor valor que podemos garantizar para MAX en el subárbol actual. Inicialmente -INF.
  - β es el menor valor que podemos garantizar para MIN en el subárbol actual. Inicialmente INF.
  - Actualizamos valores de α y β durante la búsqueda y eliminamos las ramas pendientes tan pronto como el valor del subárbol actual es peor que el valor de α o β actual para MAX o MIN.
  - Condición de parada: α >= β



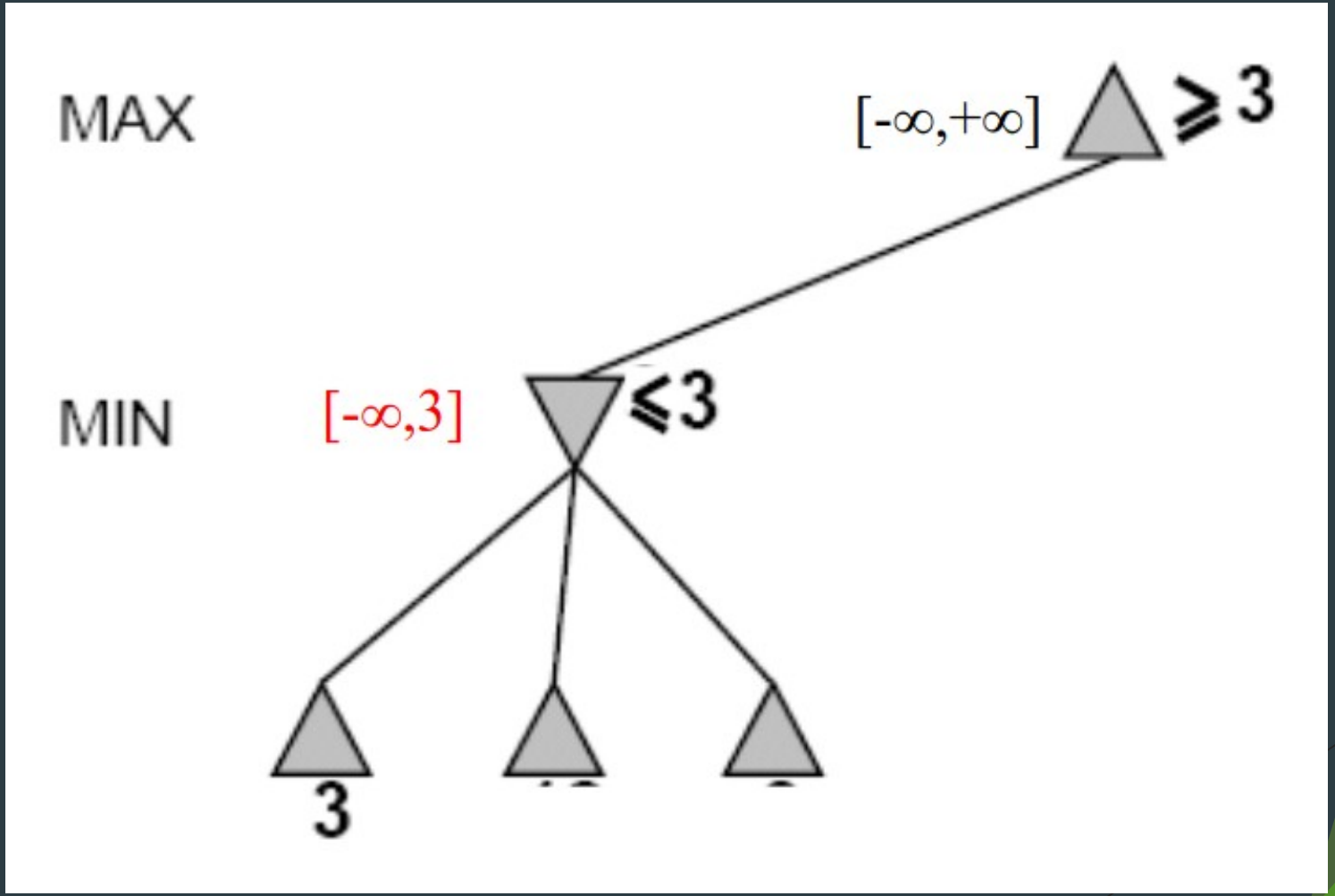
Poda alfa-beta

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) 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), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) 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) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

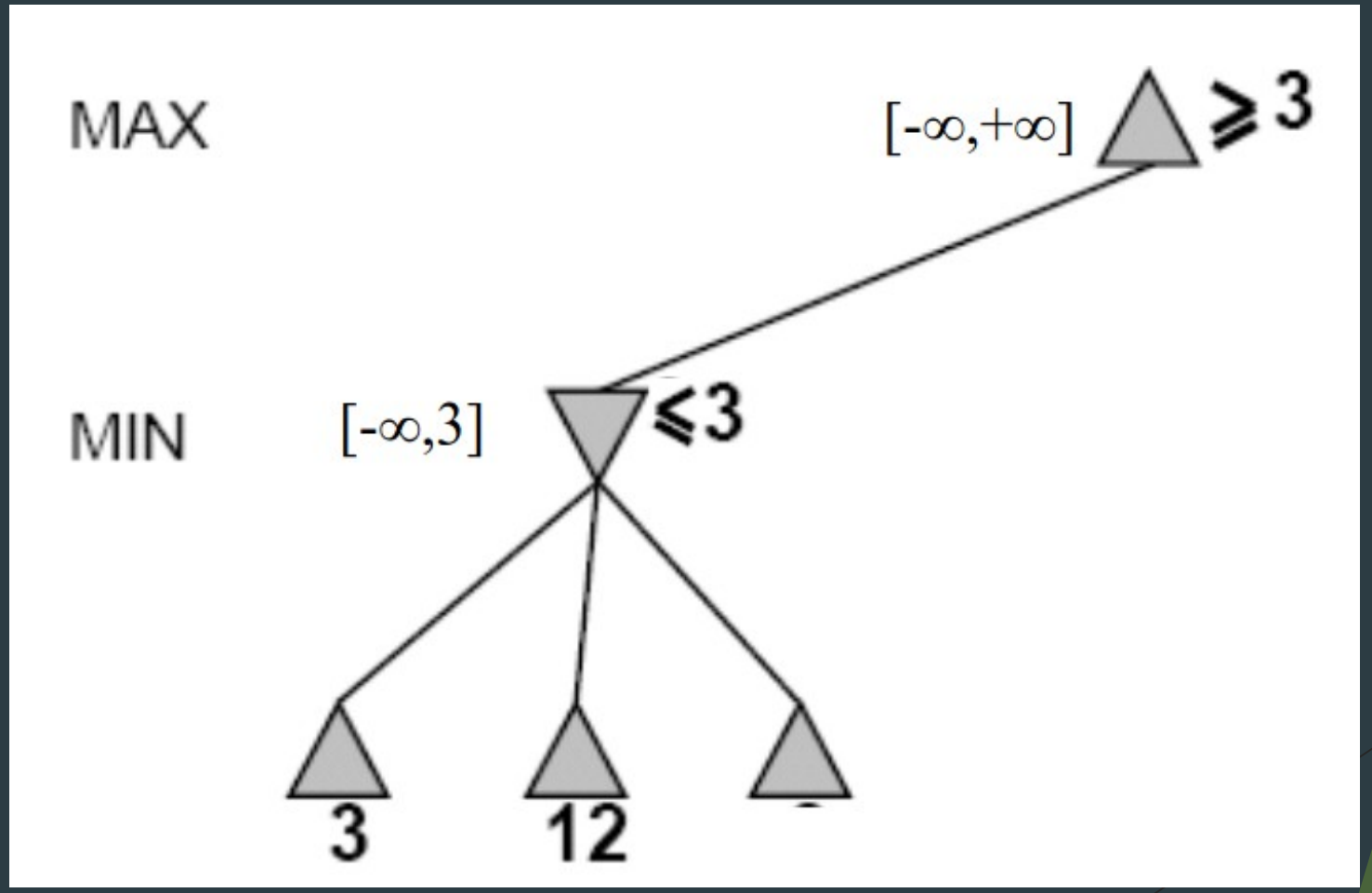
Poda alfa-beta



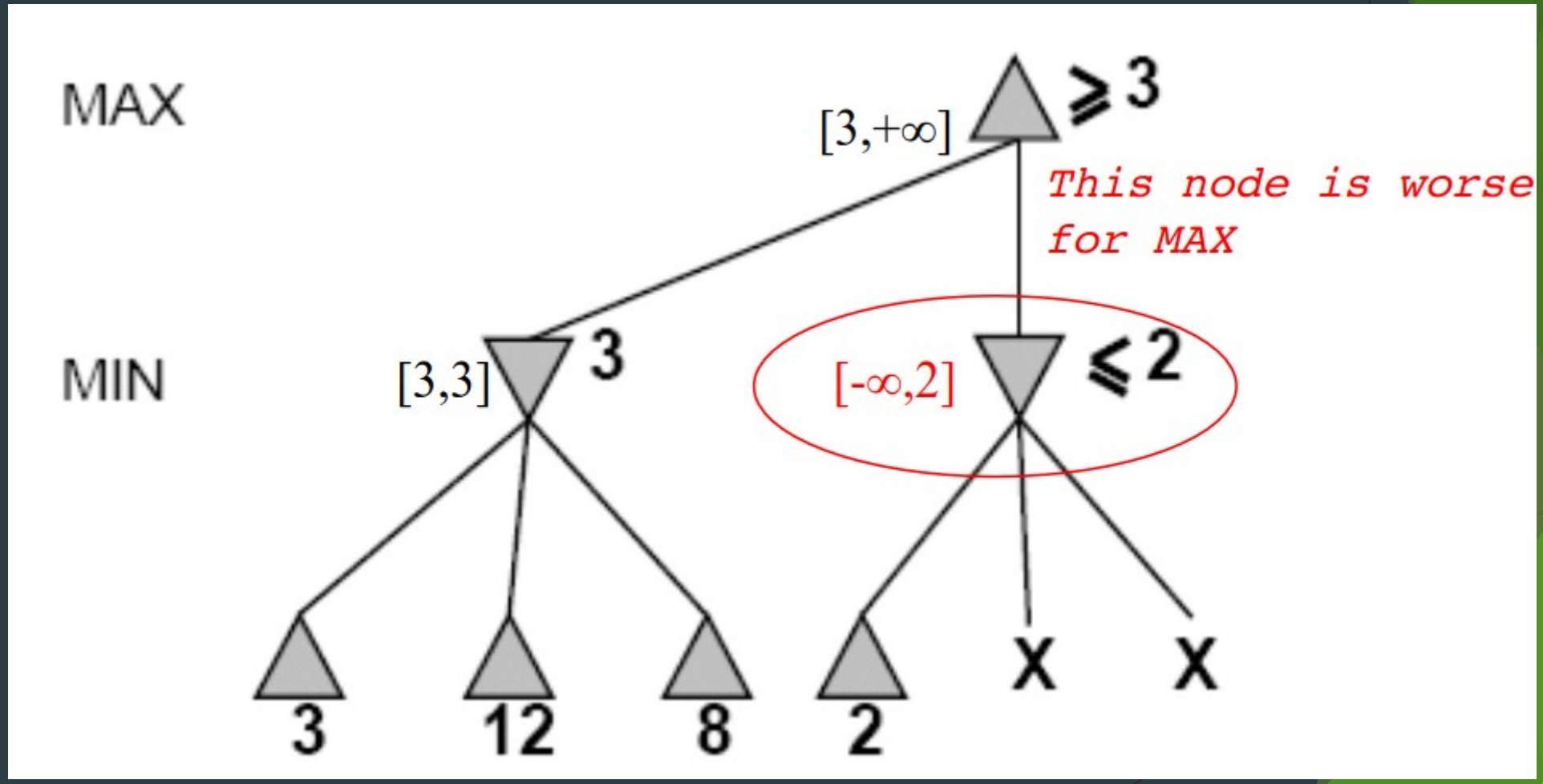
Poda alfa-beta



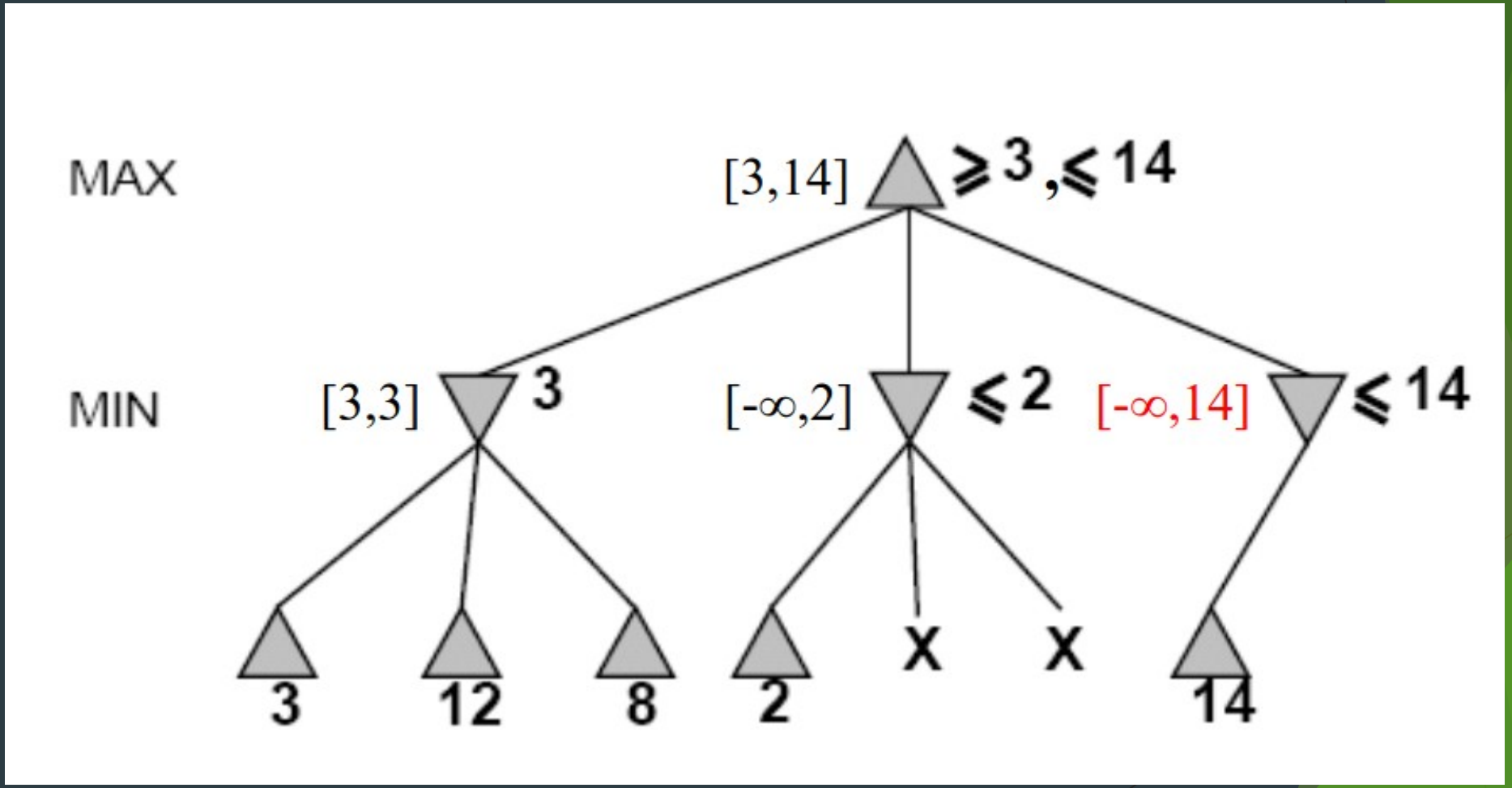
Poda alfa-beta



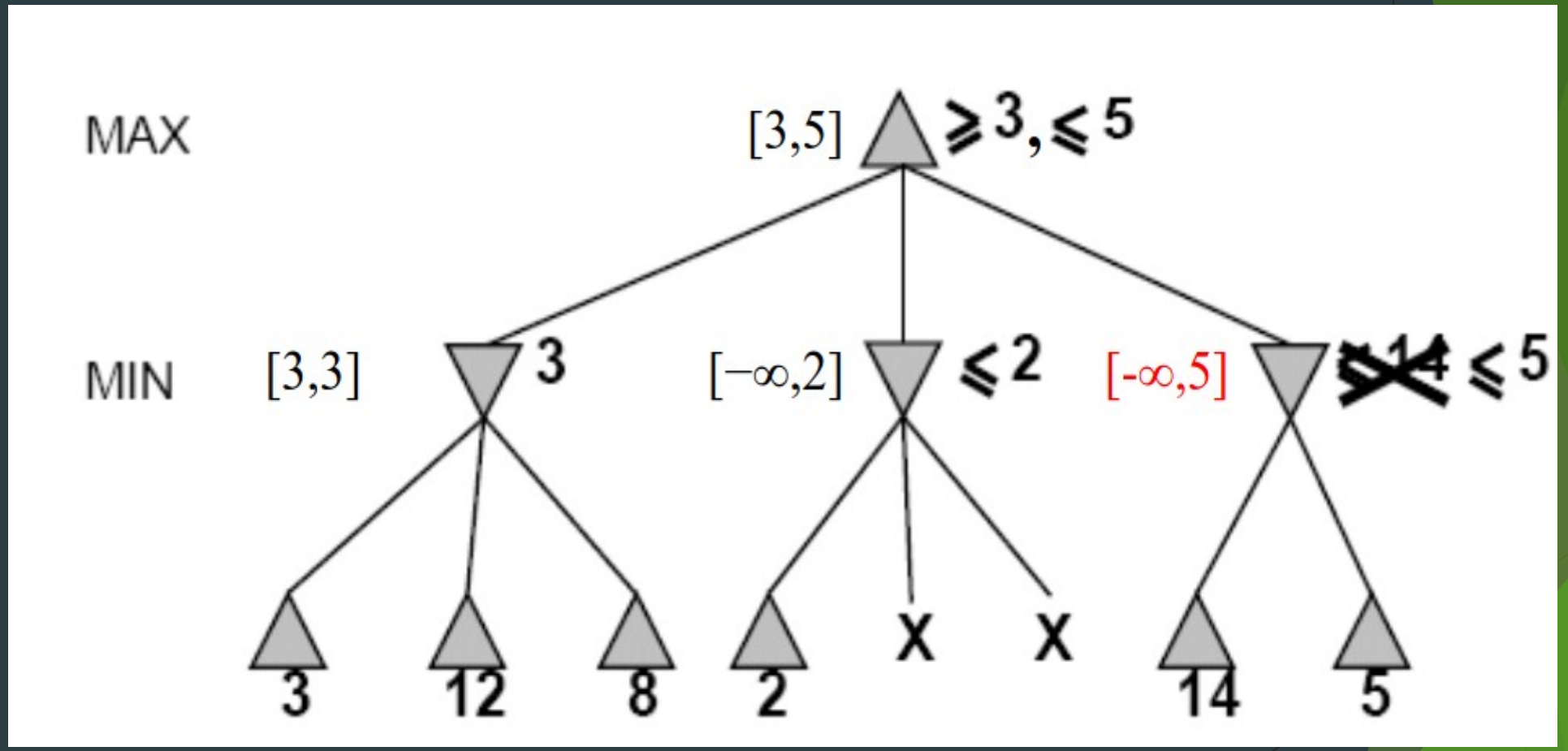
Poda alfa-beta



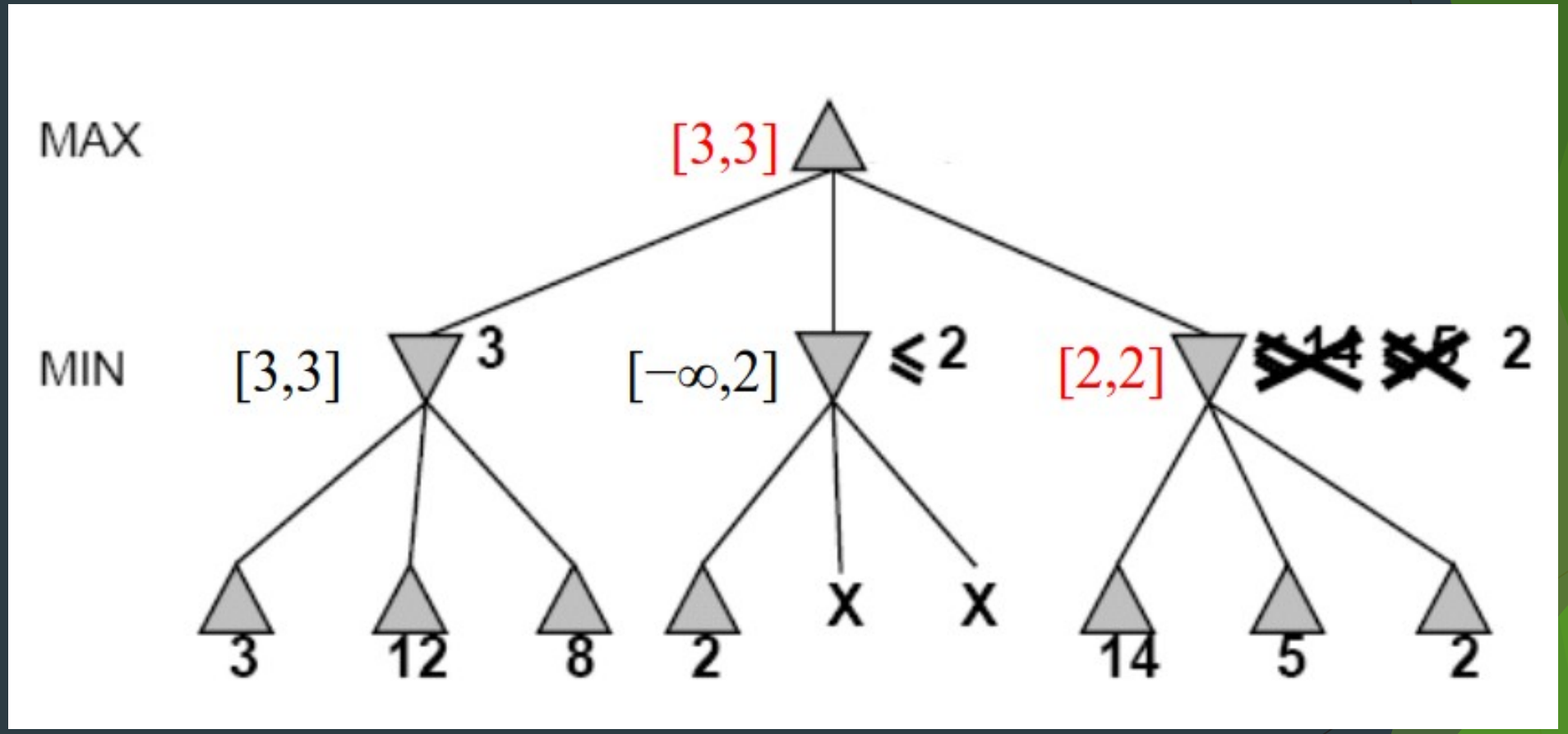
Poda alfa-beta



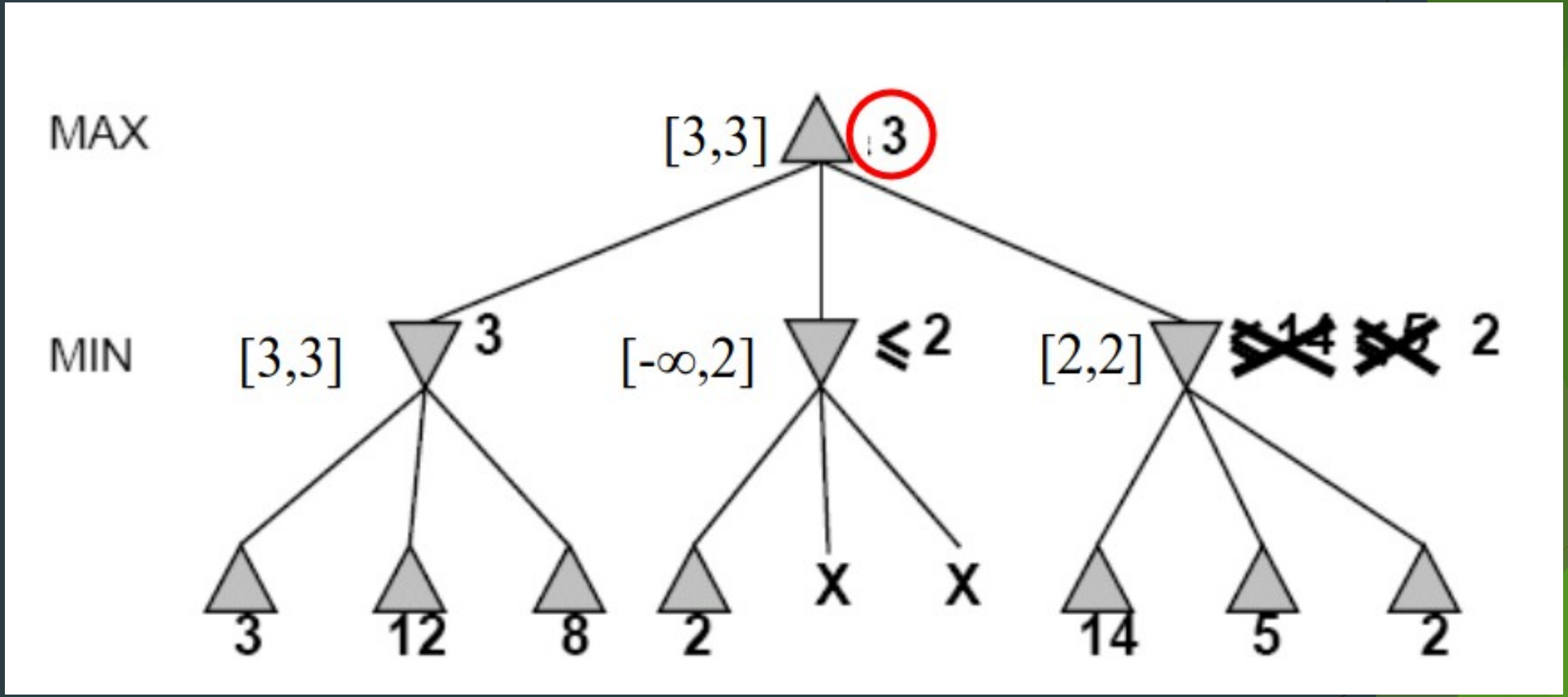
Poda alfa-beta



Poda alfa-beta

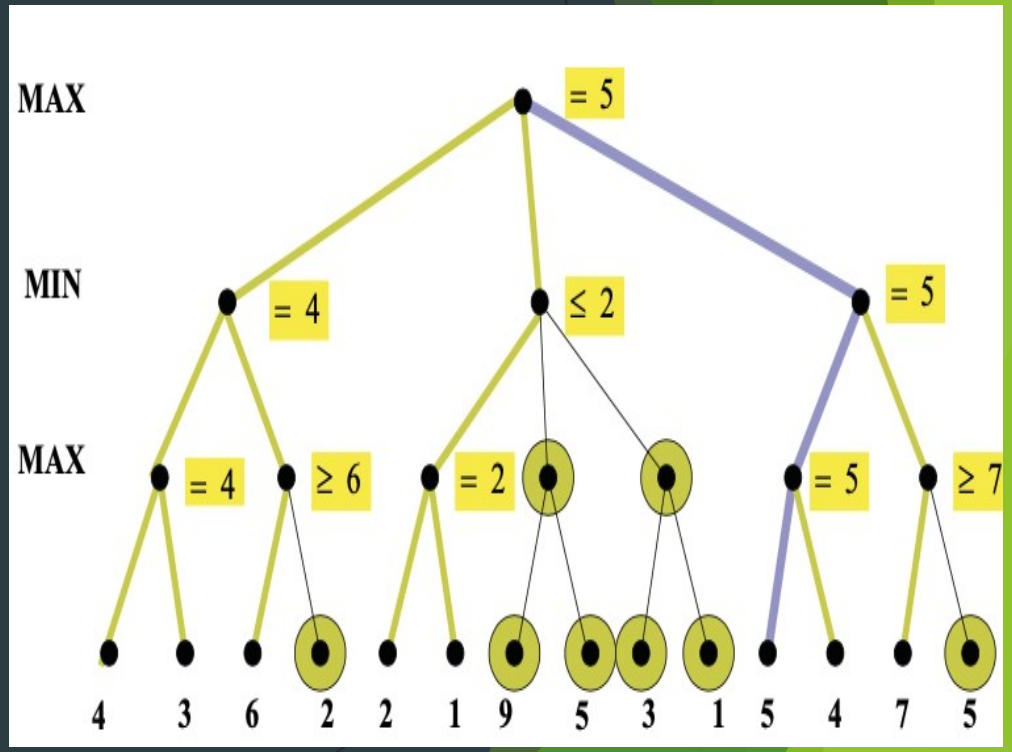


Poda alfa-beta



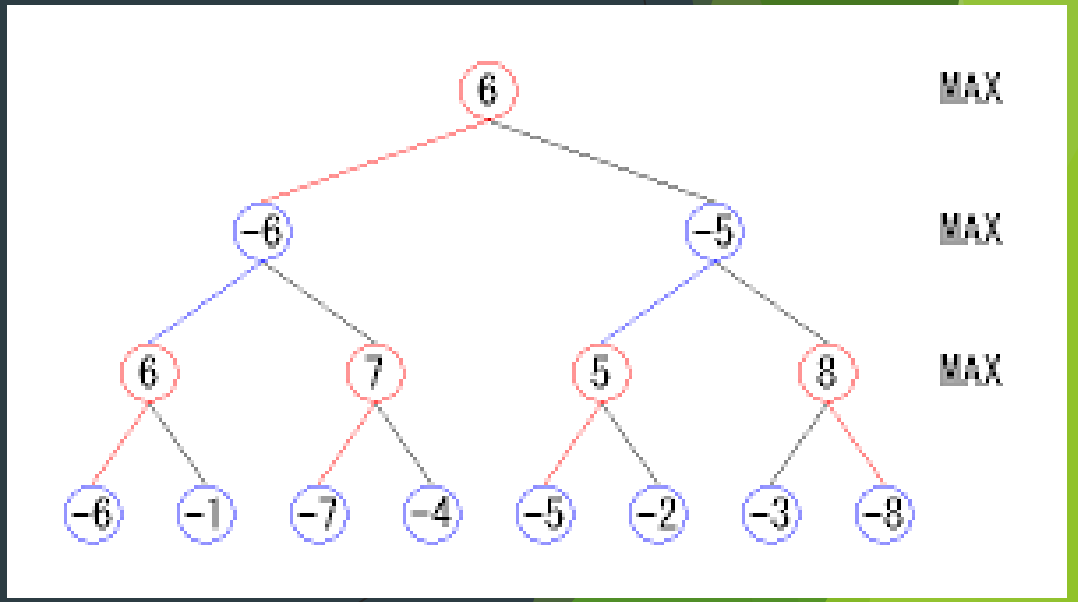
Poda alfa-beta

- Optimizaciones:
  - La poda depende mayoritariamente del orden en el que se evalúan los nodos de un determinado nivel.
  - Habitualmente, se utiliza una heurística que nos permita anticipar de alguna forma el valor esperado de cada nodo para realizar la búsqueda en el orden más conveniente.
  - Por ejemplo, en ajedrez se podría utilizar una heurística de ordenación del siguiente estilo:
    - Jaques.
    - Capturas.
    - Amenazas.
    - Movimientos hacia delante.
    - Movimientos hacia atrás.



Negamax

- Variante de minimax que utiliza la propiedad de suma cero de los juegos de 2 jugadores.
- Simplifica la implementación usando $\min(a, b) = -\max(-b, -a)$
- El valor de la posición del jugador A es la negación del valor de la posición del jugador B.
- Con un único procedimiento se pueden evaluar ambas posiciones.



Negamax

```
function negamax(node, depth, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    value := −∞
    for each child of node do
        value := max(value, -negamax(child, depth - 1, -color))
    return value
```

```
(* Initial call for Player A's root node *)
negamax(rootNode, depth, 1)
```

```
(* Initial call for Player B's root node *)
negamax(rootNode, depth, -1)
```

Búsqueda en ventana nula

- Variante de la poda alfa-beta donde la ventana de valores se establece como [α, α+1].
  - Solo se requiere el valor de alfa. β = α+1
  - La ventana de valores intermedios es 0, de ahí el nombre de ventana nula o ventana cero.
- Se establece un valor máximo de profundidad, en el que el algoritmo debe realizar una evaluación de la posición y dar una respuesta.
  - Respuesta booleana: v <= α (fallo bajo) o v > α (fallo alto)
- No produce una puntuación exacta, pero permite saber si una rama se puede cortar.
- Se utiliza para realizar cortes en las búsquedas de manera rápida.
- Presentado por primera vez en el algoritmo Scout.

Scout

- Presentado en 1980 por Judea Pearl.
- Se basa en la idea de que es posible verificar si un valor de una rama es mayor o no que un valor “v” de una manera más rápida que sabiendo su valor exacto.
- Se basa en una función de TEST para probar si todos los hijos del primer hermano están por debajo o igual a MAX, o por encima o igual a MIN.
- Si la condición no se cumple, se hace una búsqueda de nuevo para obtener el nuevo valor de MAX o MIN.
- Introduce el concepto de búsqueda en ventana nula.
- Tiene como idea que los nodos ahorrados en el test compensarán con creces la repetición de las búsquedas cuando se utiliza en árboles razonablemente bien ordenados.

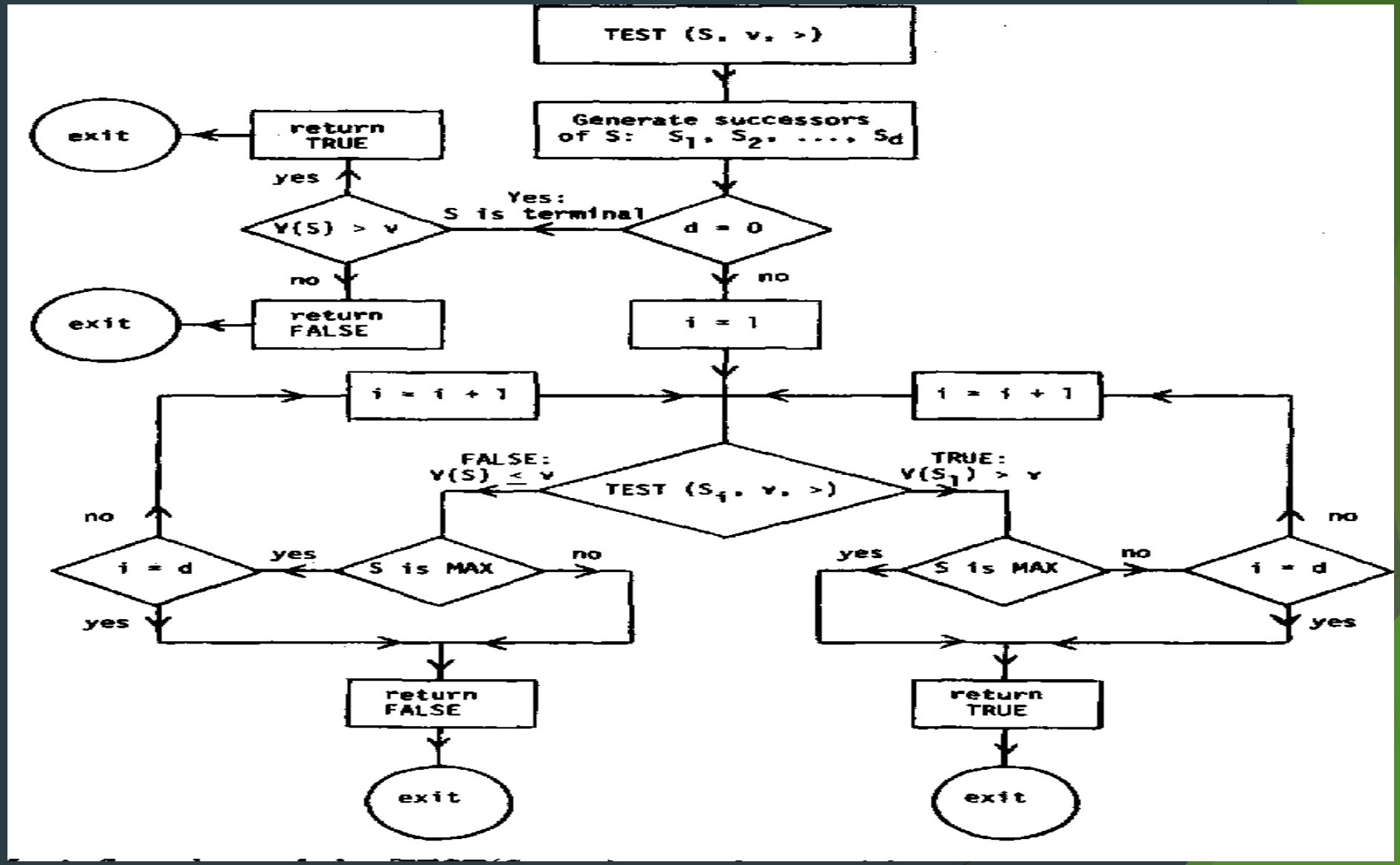
Scout

- Scout utiliza dos procedimientos recursivos llamados EVAL y TEST.
- El procedimiento principal EVAL(S) devuelve V(S), el valor de minimax para la posición S.
- La función de TEST(S, v, >) se utiliza para validar o refutar la verdad de la desigualdad V(S) > v, donde v es un valor de referencia dado.
- Ventajas:
  - La mayoría de los tests resultan en el descarte del nodo probado (y sus descendientes) de cualquier evaluación posterior.
  - El procedimiento de test es relativamente rápido y permite realizar muchos cortes que no necesariamente son permitidos por EVAL o cualquier otro esquema de evaluación. Tan pronto como un sucesor de un nodo MAX cumple el criterio V(S) > v, todos los otros sucesores pueden ser ignorados.

Scout

- Procedimiento TEST(S, v, >).
  - Para verificar si S satisface la desigualdad V(S) > v, comenzar aplicando el mismo test (llamándose a sí mismo) a los sucesores de izquierda a derecha.
  - Si S es MAX:
    - Devolver VERDAD tan pronto como un sucesor sea mayor que v.
    - Devolver FALSO si todos los sucesores son menor o iguales a v.
  - Si S es MIN:
    - Devolver FALSO tan pronto como un sucesor sea menor o igual a v.
    - Devolver VERDAD si todos los sucesores son mayores que v.

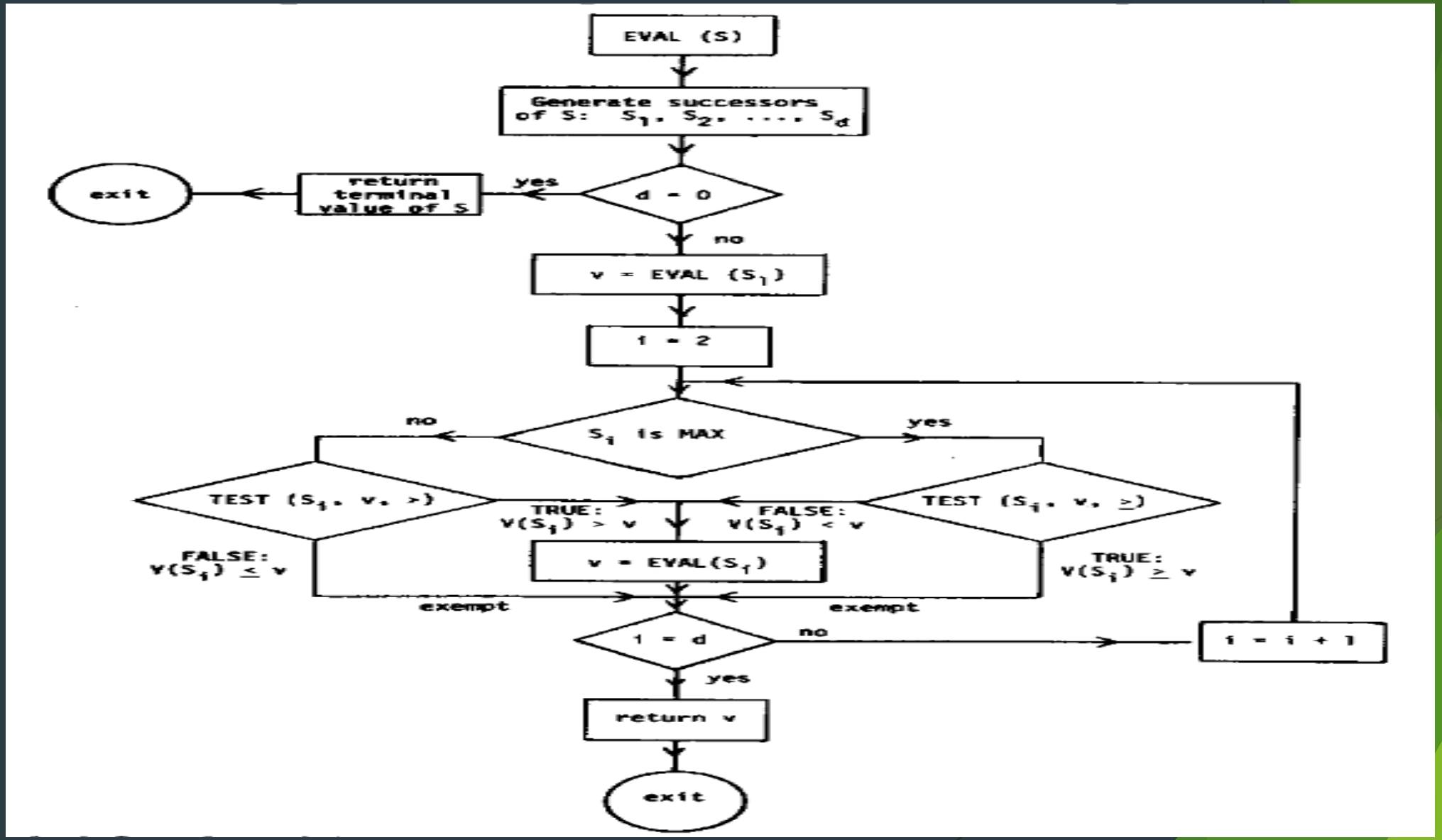
Scout


TEST (S, v, >)
Generate successors of S: $S_1, S_2, \ldots, S_d$
d = 0
Yes: S is terminal
V(S) > v
yes
return TRUE
exit
no
return FALSE
exit
no
i = 1
i = i + 1
i = i + 1
FALSE: V(S) ≤ v
TEST (S_i, v, >)
TRUE: $V(S_1)$ > v
no
i = d
yes
S is MAX
no
yes
yes
S is MAX
no
i = d
no
yes
return FALSE
exit
return TRUE
exit

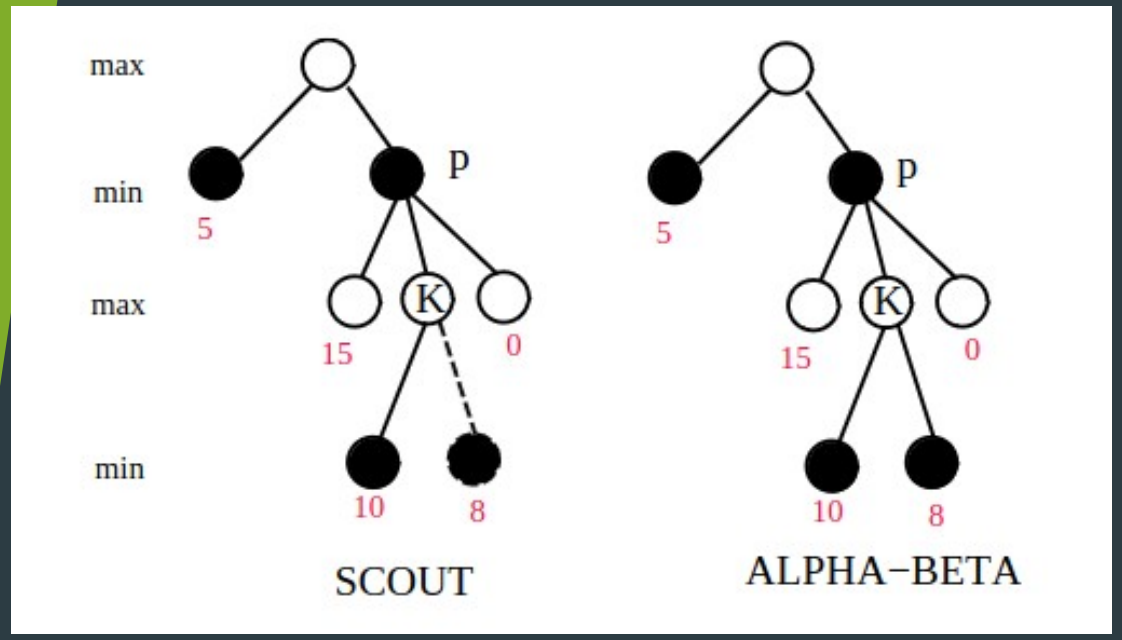
Scout

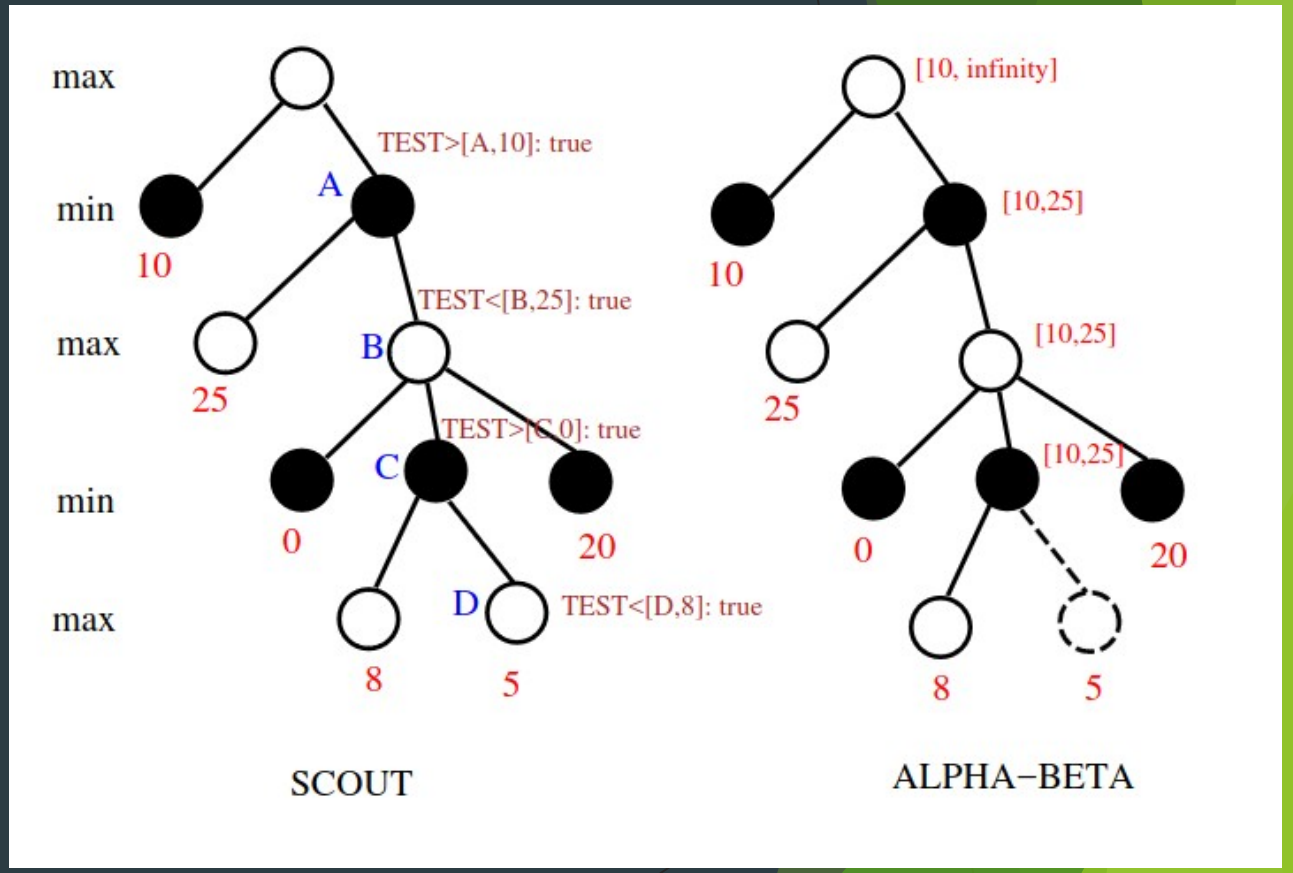
- Procedimiento EVAL(S)
  - EVAL evalúa una posición MAX S evaluando (llamándose a sí mismo) su sucesor más a la izquierda $S_{1.}$
  - Realiza scouting con el resto de sucesores llamando a TEST para determinar si $V(S_k) > V(s_1)$.
    - Si la desigualdad se cumple para Sk, el nodo se evalúa de manera exacta con EVAL(Sk) y su valor V(Sk) se utiliza para los tests de scouting subsiguientes.
    - Si no se cumple, S_k queda exento de evaluación y S_{k+1} pasa a testearse.
    - Cuando todos los sucesores han sido evaluados o testados, el último valor obtenido es asignado a V(S).
  - Un procedimiento idéntico se realiza para evaluar la posición MIN S, con la diferencia de que $V(S_k) \geq V(S_1)$ constituye la prueba para excluir a S_k de la evaluación.

Scout


EVAL (S)
Generate successors of S: $S_1, S_2, \ldots, S_d$
d = 0
yes
return terminal value of S
exit
no
v = EVAL (S_1)
i = 2
S_i is MAX
no
yes
TEST (S_i, v, >)
TRUE: $V(S_i) > v$
FALSE: $V(S_i) < v$
TEST (S_i, v, ≥)
FALSE: $V(S_i) \leq v$
v = EVAL(S_i)
TRUE: $V(S_i) \geq v$
exempt
exempt
i = d
no
i = i + 1
yes
return v
exit

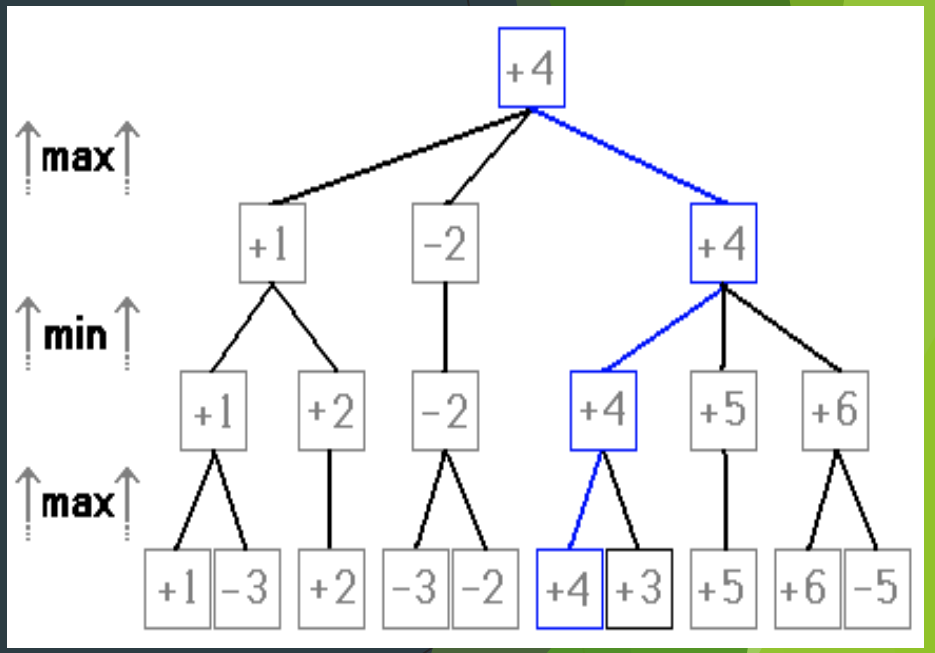
Scout


max
min
max
min
p
K
5
15
0
10
8
SCOUT
p
K
5
15
0
10
8
ALPHA−BETA


max
min
max
min
max
A
B
C
D
10
25
0
20
8
5
TEST>[A,10]: true
TEST<[B,25]: true
TEST>[C,0]: true
TEST<[D,8]: true
SCOUT
[10, infinity]
[10,25]
[10,25]
[10,25]
10
25
0
20
8
5
ALPHA−BETA

Variante principal (PV)

- Variante particular que es la más ventajosa para el jugador actual.
- Secuencia donde ambos bandos juegan el movimiento más fuerte.
- Se trata de la mejor o más correcta línea de juego.
- Todos los nodos de la PV tienen el mismo valor que la raíz.
- Ningún jugador puede mejorar los movimientos de la PV.
- Pueden haber diferentes PV si hay movimientos igualmente buenos.
- El término PV se usa típicamente para la primera secuencia descubierta. El resto son cortados mediante pruning.



Principal Variation Search (PVS) y Negascout

- Mejora de la poda alfa-beta para calcular el valor minimax de un nodo.
- Utiliza el concepto de variante principal (PV) para realizar los cortes.
- La idea del algoritmo es que solo se necesita la puntuación exacta en la variante principal y una función de verificación rápida para descartar los nodos subóptimos.
- Requiere de una ordenación precisa para aprovechar sus capacidades al máximo.
- La ordenación de los movimientos se suele determinar por búsquedas previas a menor profundidad.
- Asume que el primer nodo es la variante principal y verifica el resto con una ventana nula, lo que es más rápido que una ventana alfa-beta normal.
- Si el test falla, el primer nodo no era la variante principal y la búsqueda continua con el algoritmo normal alfa-beta.
- Con una ordenación aleatoria de los nodos es más lento que el algoritmo alfa-beta, al tener que revisitar algunos nodos.

Principal Variation Search (PVS) y Negascout

- La profundidad es suficiente o está en una posición terminal, devolver el valor calculado por la función de evaluación de la posición.
- Para el primer hijo, realizar una búsqueda normal con ventana alfa-beta.
- Para el resto de hijos, realizar una búsqueda en ventana nula para testear si se deben hacer cortes.
- Realiza rebúsquedas sobre los nodos cuando el valor de un subárbol sea mayor que el mayor valor actual calculado.
- Cuando se revisita un subárbol, es mejor usar información de la búsqueda previa para acelerar la búsqueda actual, como empezar por la posición donde se devolvió la estimación de valor más alta.

Principal Variation Search (PVS) y Negascout

```
function pvs(node, depth, α, β, color) is
    if depth = 0 or node is a terminal node then
        return color × the heuristic value of node
    for each child of node do
        if child is first child then
            score := -pvs(child, depth - 1, -β, -α, -color)
        else
            score := -pvs(child, depth - 1, -α - 1, -α, -color) (* search with a null window *)
            if α < score < β then
                score := -pvs(child, depth - 1, -β, -score, -color) (* if it failed high, do a full re-search
*)
        α := max(α, score)
        if α ≥ β then
            break (* beta cut-off *)
    return α
```

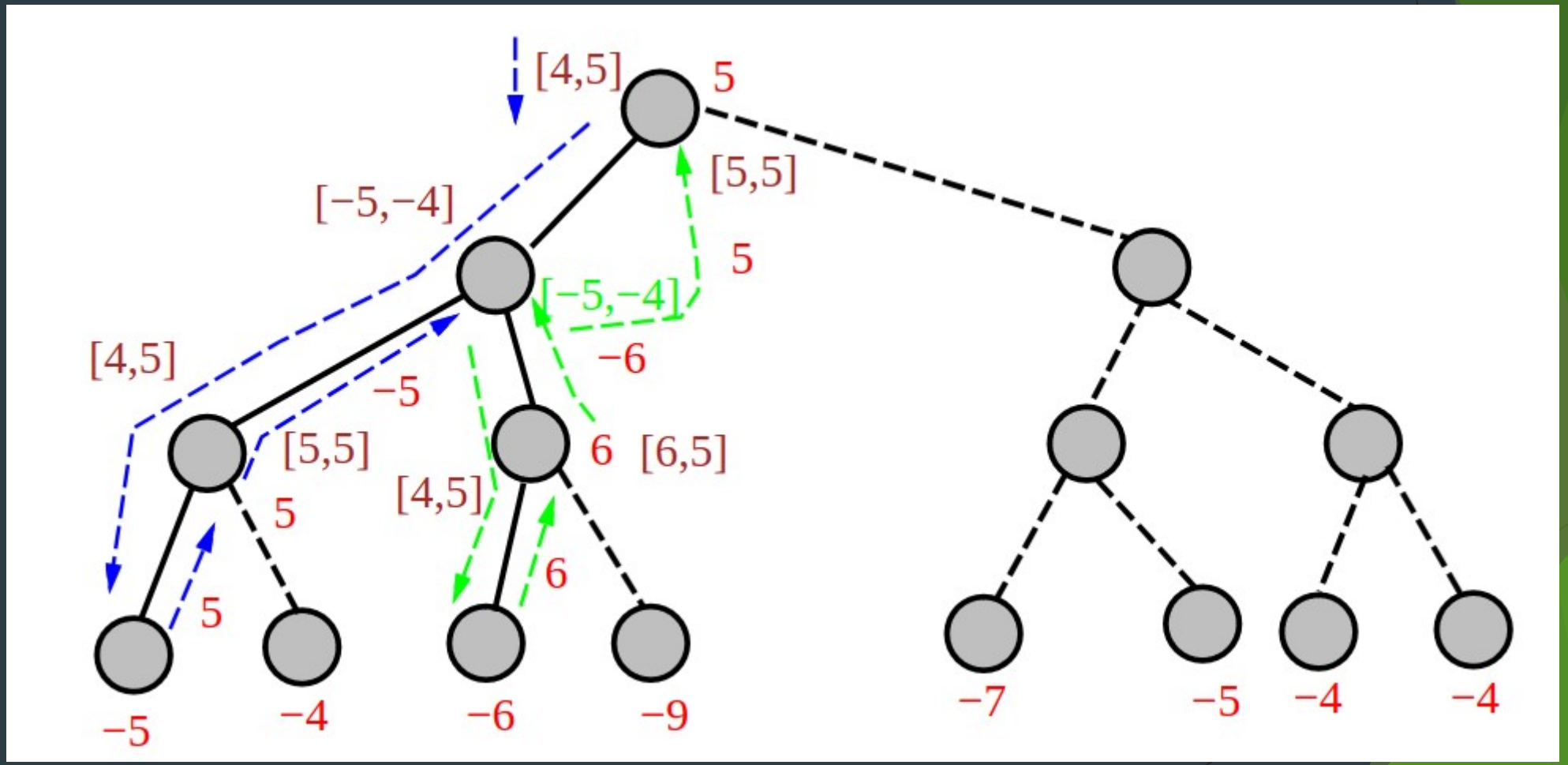
Principal Variation Search (PVS) y Negascout

```
int NegaScout ( position p; int alpha, beta )
{                        /* compute minimax value of position p */
   int a, b, t, i;
   if ( d == maxdepth )
      return Evaluate(p);                          /* leaf node */
   determine successors p_1,...,p_w of p;
   a = alpha;
   b = beta;
   for ( i = 1; i <= w; i++ ) {
      t = -NegaScout ( p_i, -b, -a );
      if ( (t > a) && (t < beta) && (i > 1) && (d < maxdepth-1) )
         a = -NegaScout ( p_i, -beta, -t );     /* re-search */
      a = max( a, t );
      if ( a >= beta )
         return a;                                  /* cut-off */
      b = a + 1;                          /* set new null window */
   }
   return a;
}
```

Principal Variation Search (PVS) y Negascout

```
int NegaScout ( position p; int alpha, beta )
{                        /* compute minimax value of position p */
   int b, t, i;
   if ( d == maxdepth )
      return quiesce(p, alpha, beta);            /* leaf node */
   determine successors p_1,...,p_w of p;
   b = beta;
   for ( i = 1; i <= w; i++ ) {
      t = -NegaScout ( p_i, -b, -alpha );
      if ( (t > a) && (t < beta) && (i > 1) )
         t = -NegaScout ( p_i, -beta, -alpha ); /* re-search */
      alpha = max( alpha, t );
      if ( alpha >= beta )
         return alpha;                                /* cut-off */
      b = alpha + 1;                      /* set new null window */
   }
   return alpha;
}
```

Principal Variation Search (PVS) y Negascout



MTD(f)

- Algoritmo de búsqueda de mejor primero.
- Memory-enhanced Test Driver (MTD(f)) fue presentado como una mejora del algoritmo Alfa-Beta.
- Usa la ventana nula del algoritmo Negascout de manera eficiente.
- Utiliza dos variables más para determinar el límite superior e inferior.
- El algoritmo puede realizar varias búsquedas en cada llamada Alfa-Beta y usar los límites obtenidos para favorecer la convergencia usando los límites inferior y superior para realizar cortes más rápidos en el árbol.
- Utiliza una tabla de transposición para almacenar y obtener información sobre partes del árbol para agilizar el revisitado de los nodos del árbol.
- La tabla de transposición requiere mayor espacio de memoria.

MTD(f)

- F. Primera estimación del valor. 0 en la primera llamada.
- D. Profundidad a la que buscar.
- AlphaBetaWithMemory. Versión de la búsqueda alfa-beta que almacena resultados previos.
- MTD(f) realiza las búsquedas de ventana nula desde la raíz del árbol y necesita tablas de transposición para ser eficiente.

```
function MTDF(root, f, d){
        g := f
        upperBound := +∞
        lowerBound := -∞
        while lowerBound < upperBound{
                if g = lowerBound then
                        β := g+1
                else
                        β := g
                g := AlphaBetaWithMemory (root,  β-1,
                                                 β, d)
                if g < β then
                        upperBound := g
                else
                        lowerBound := g
        }
        return g
}
```

MTD(f)

```
function AlphaBetaWithMemory(n : node_type; alpha , beta , d : integer) : integer;
    if retrieve(n) == OK then /* Transposition table lookup */
    if n.lowerbound >= beta then return n.lowerbound;
    if n.upperbound <= alpha then return n.upperbound;

    alpha := max(alpha, n.lowerbound);
    beta := min(beta,n.upperbound);

    if d == 0 then g := evaluate(n); /* leaf node */
    else if n == MAXNODE then
        g := -INFINITY;
        a := alpha; /* save original alpha value */
        c := firstchild(n);

        while (g < beta) and (c != NOCHILD) do
            g := max(g, AlphaBetaWithMemory(c, a, beta, d - 1));
            a := max(a, g);
            c := nextbrother(c);

    else /* n is a MINNODE */
        g := +INFINITY;
        b := beta; /* save original beta value */
        c := firstchild(n);

        while (g > alpha) and (c != NOCHILD) do
            g := min(g, AlphaBetaWithMemory(c, alpha, b, d - 1));
            b := min(b, g);
            c := nextbrother(c);

        /* Traditional transposition table storing of bounds */  /* Fail low result implies anupper bound */
        if g <= alpha then n.upperbound := g; store n.upperbound;   /* Found an accurate minimax value - will not occur if called with zero window*/
        if g > alpha and g < beta then n.lowerbound := g; n.upperbound := g;
        store n.lowerbound, n.upperbound;   /* Fail high result implies a lower bound*/
        if g >= beta then n.lowerbound := g;
            store n.lowerbound;
        return g;
```

B*

- Algoritmo de búsqueda de mejor primero.
- Encuentra el camino de menor coste desde el nodo a cualquier nodo objetivo de entre uno o varios objetivos posibles.
- Utiliza tres valores: la evaluación estática y una cota inferior y superior del valor real.
- Las ideas principales del algoritmo están basadas en:
  - Parar cuando un camino es mejor que el resto.
  - Centrarse en la exploración de caminos que puedan llevar a parar el algoritmo.
- Expande la búsqueda basándose en una estrategia de probar el mejor y refutar el resto.
  - En la estrategia de probar el mejor, el algoritmo selecciona el nodo con la mayor cota superior porque tiene mayor probabilidad de aumentar su cota inferior por encima de la cota superior del resto de nodos.
  - En la estrategia de refutación, se selecciona el nodo con la segunda cota superior más alta porque tiene mayor probabilidad de reducir su cota superior por debajo de la cota inferior del mejor nodo.

B*

```
(1) DEPTH ← 0; CURNODE ← 0; OPTIM[0] ← −∞; PESSIM[0] ← ∞;
(2) if CURNODE has not been expanded yet then generate and evaluate
    successors, giving each a name and a pointer to CURNODE.
(3) BESTNODE ← name of successor of CURNODE with best OPTIM value;
    ALTERN ← name of successor with second best OPTIM value;
    MAXOPTIM ← OPTIM[BESTNODE];
    MAXPESS ← Value of the best PESSIM value of all successors;
(4) if (MAXOPTIM <' PESSIM[CURNODE])
      or (MAXPESS >' OPTIM[CURNODE]) then
    begin                                        ! Back-up (values and search)
        PESSIM[CURNODE] ← MAXOPTIM;
        OPTIM[CURNODE] ← MAXPESS;
        if DEPTH > 0 then
        begin
          CURNODE ← PARENT[CURNODE];
          DEPTH ← DEPTH−1;
          go to 3;
        end
        else if DEPTH = 0 then              ! Check for termination
              if PESSIM[BESTNODE] ≥ OPTIM[ALTERN] then
                exit with ANSWER = BESTNODE;
    end;
```

```
(5) if DEPTH = 0 then
    begin
      decide STRATEGY;
      if STRATEGY = DISPROVEREST then CURNODE ← ALTERN
      else if STRATEGY = PROVEBEST then CURNODE ← BESTNODE;
    end;
(6) if (DEPTH ≠ 0) then CURNODE ← BESTNODE;
(7) DEPTH ← DEPTH+1; go to 2.                ! Go forward
```

B*

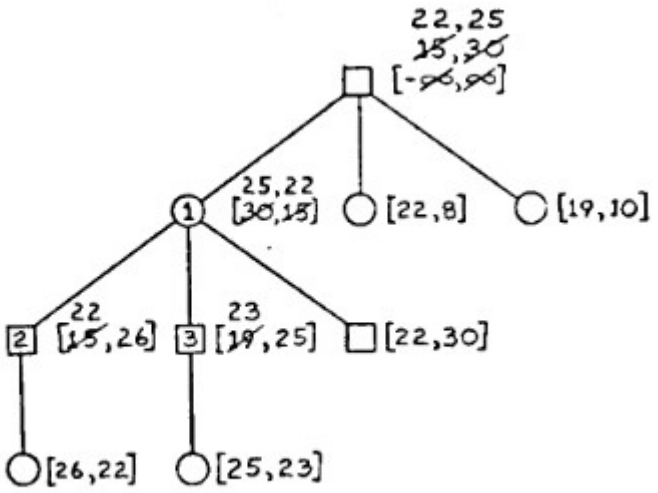


FIG. 4. The PROVEBEST strategy.

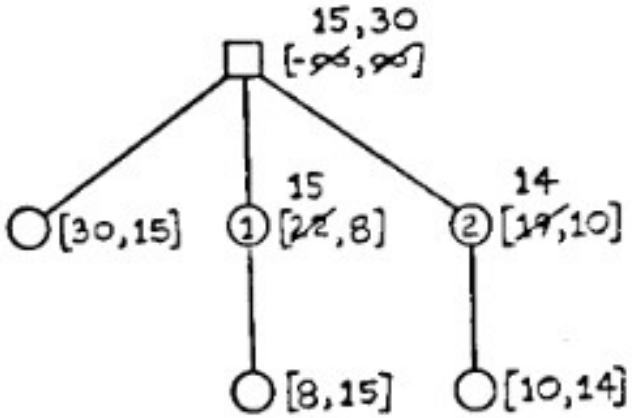
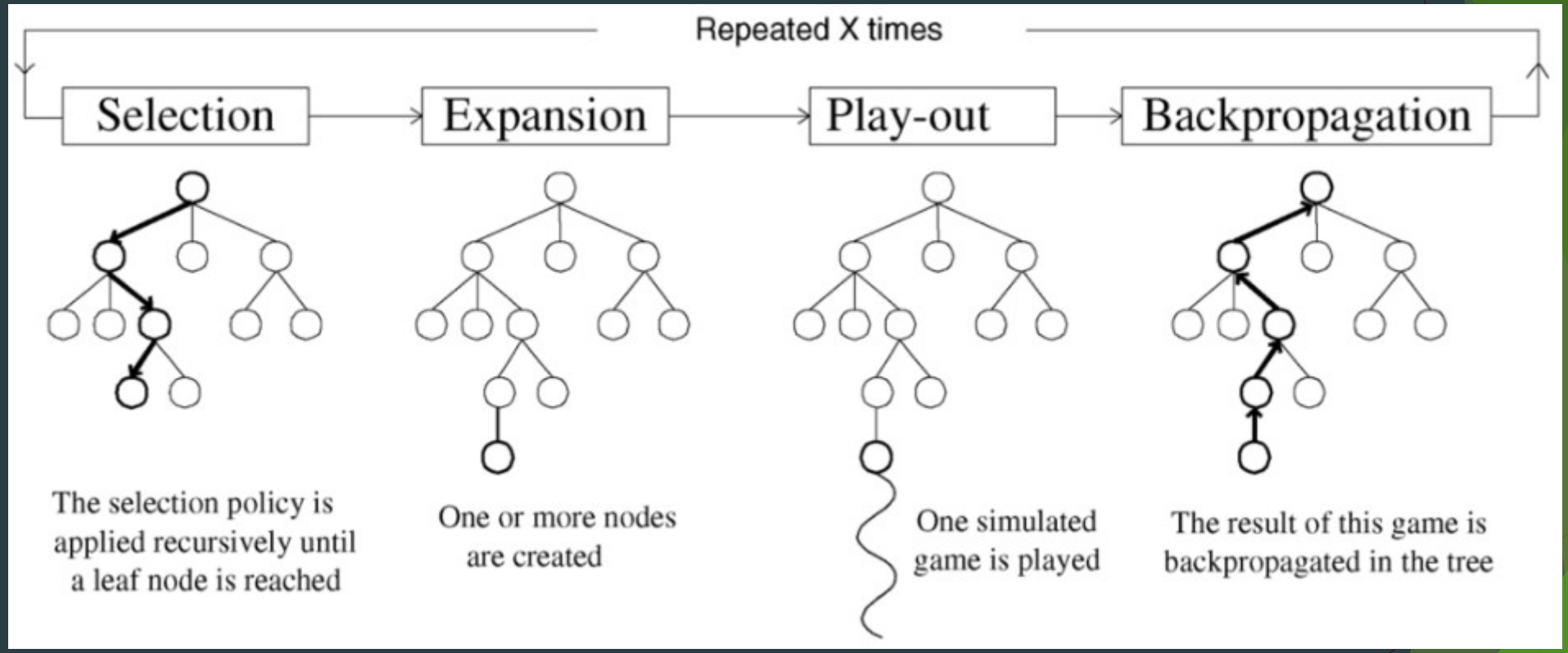


FIG. 5. The DISPROVEREST strategy.

Monte-Carlo Tree Search (MCTS)

- Basado en una exploración aleatoria del árbol de búsqueda.
- Utiliza los resultados examinados anteriormente de los nodos.
- Cada vez que se ejecuta produce una mejor estimación de los valores.
- Consiste de cuatro fases, repetidas iterativamente mientras se disponga de tiempo:
  - Fase de selección. Comienza en el nodo raíz, atraviesa el árbol de juego seleccionando el movimiento más prometedor hasta alcanzar un nodo hoja.
  - Fase de expansión. Si el nombre de visitas alcanza un umbral, la hoja es expandida para construir un árbol más grande.
  - Fase de simulación. Calcula el valor de la hoja realizando una simulación de juego en ella.
  - Fase de retropropagación. Se retrotrae desde la hoja hasta la raíz para actualizar los valores modificados en la fase de simulación.

Monte-Carlo Tree Search (MCTS)



Monte-Carlo Tree Search (MCTS)

```
Data: root node
Result: best move
while (has time) do
    current node ← root node

    /* The tree is traversed
    while (current node ϵ T ) do
        last node ← current node
        current node ← Select(current node)
    end

    /* A node is added
    last node ← Expand(last node)

    /* A simulated game is played
    R ← P lay simulated game(last node)

    /* The result is backpropagated
    current node ← last node
    while (current node ϵ T ) do
        Backpropagation(current node, R)
        current node ← current node.parent
    end

end

return best move = argmaxN ϵ Nc (root node)
```

Monte-Carlo Tree Search (MCTS)

- Combina las estrategias tradicionales de los métodos de Montecarlo, muestreo aleatorio y evaluación estadística, con técnicas de búsqueda en árboles.
- Se especializa en muestrear y explorar solamente las zonas prometedoras del espacio de búsqueda.
- La idea central es construir un árbol de búsqueda de forma incremental mediante la simulación de más de una actuación aleatoria (conocida como "rollout" o "playout") desde la posición actual.
  - Estas simulaciones se llevan a cabo hasta que se alcanza un estado terminal o una profundidad predefinida.
  - Los resultados de estas simulaciones se retropropagan por el árbol, actualizando los registros de los nodos visitados en alguna fase de la jugada, lo que incluye los porcentajes de victoria.
- El algoritmo balancea de manera dinámica entre exploración y explotación de nodos (UCT).
  - Explotación de movimientos con alto porcentaje de victoria.
  - Exploración de movimientos poco visitados.

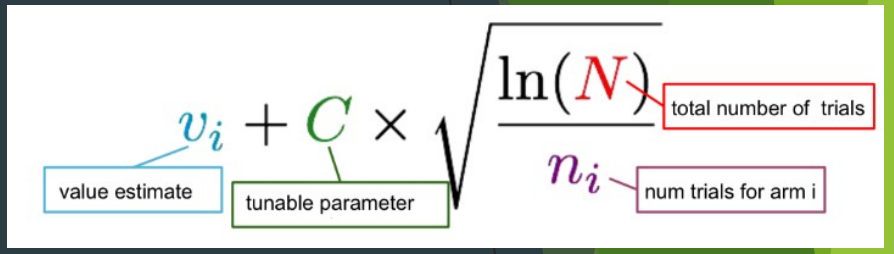
Monte-Carlo Tree Search (MCTS)

- Upper Confidence Bound (UCB)

  1) Primero, intentar cada nodo una vez.

  2) Entonces, en cada iteración:

     - Seleccionar el nodo i que maximiza UCB.



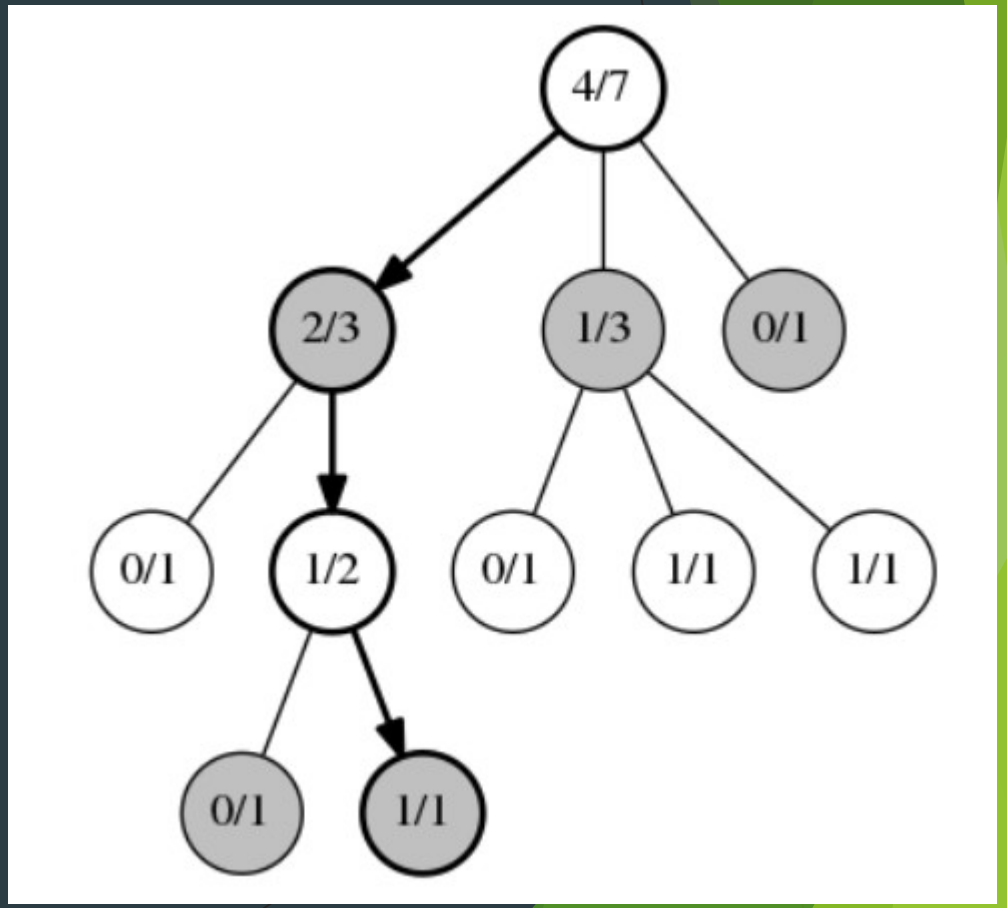
- Explotación: el valor v_i más alto es mejor.
  - Se espera que el valor verdadero esté en cierto intervalo de confianza con respecto a v_i.
- Exploración: el intervalo de confianza es grande cuando el número de pruebas n_i es pequeño.
  - Disminuir la incertidumbre aumentando el número de pruebas de un nodo cuando es mucho menor que el total.
- Cuando se aplica sobre árboles, simulando el resultado de los caminos, se conoce como Upper Confidence bounds applied to Trees (UCT).

Monte-Carlo Tree Search (MCTS)

- Selección de movimientos para UCT.
  - Escenario:
    - Ejecutar UCT tantas iteraciones como sea posible.
    - Ejecutar simulaciones e ir haciendo crecer el árbol.
  - Cuando se termina el tiempo, ¿qué movimiento seleccionar?.
    - Mayor porcentaje de victorias.
    - Mayor UCB.
    - Movimiento simulado más veces.
      - El algoritmo prueba los movimientos más interesantes con mayor frecuencia.

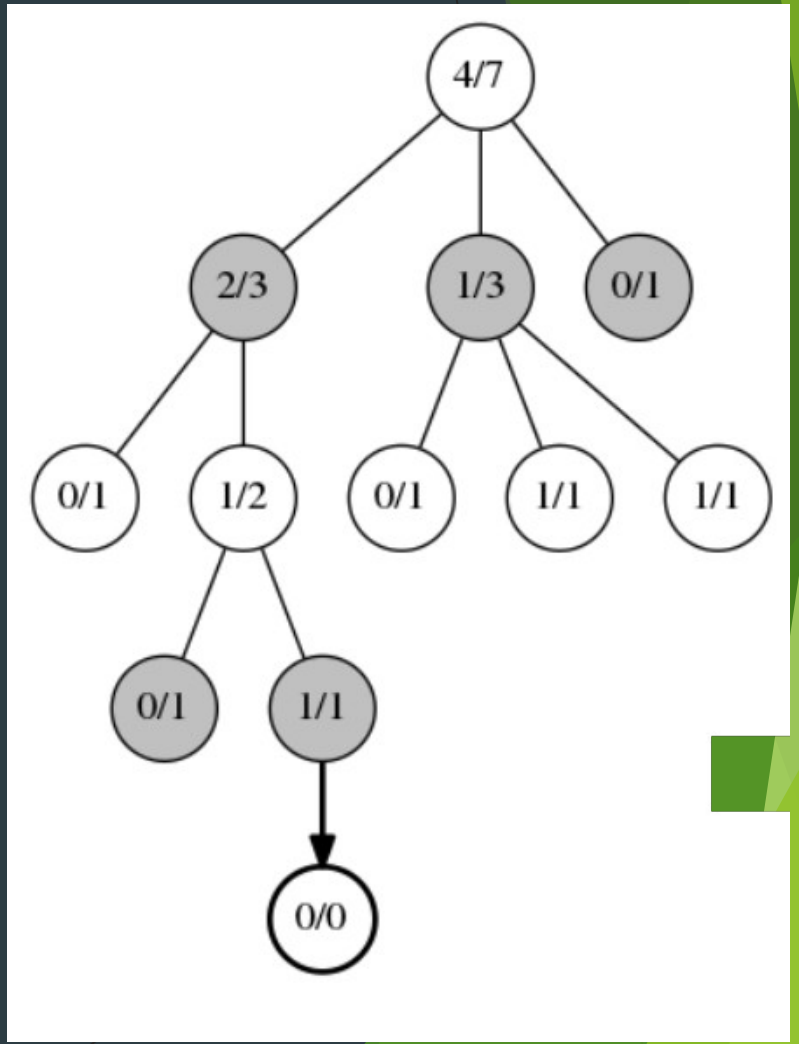
Monte-Carlo Tree Search (MCTS)

1) **Selección**. La política de selección se aplica de manera recursiva hasta que se encuentra un nodo hoja.



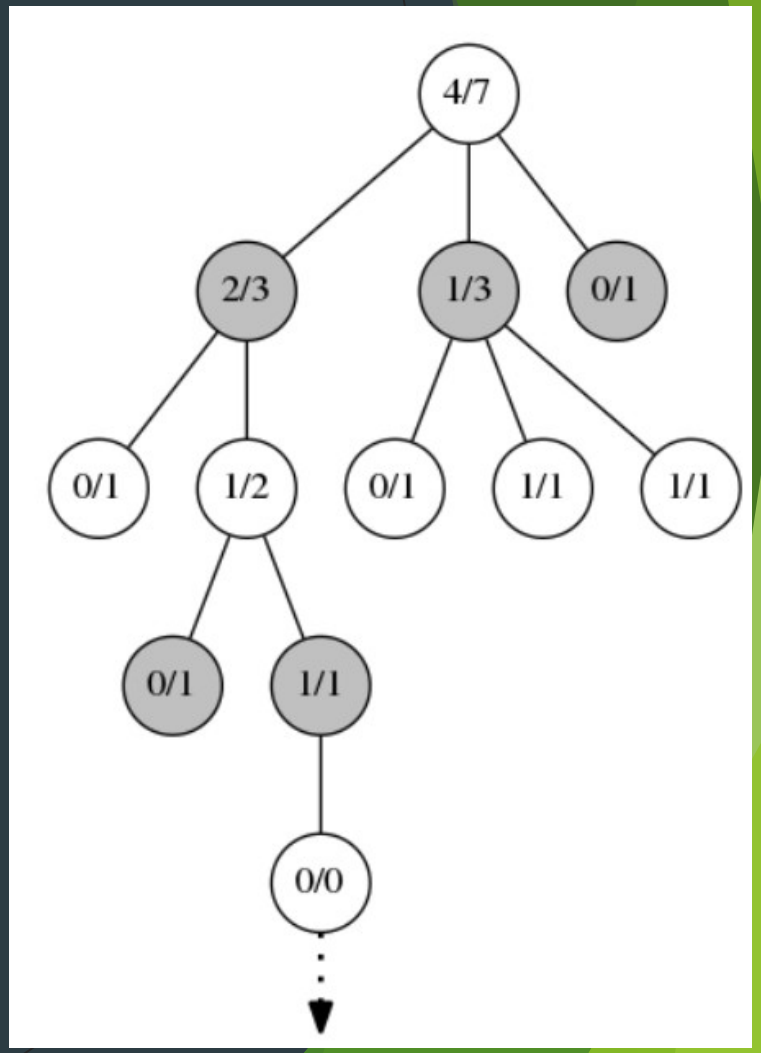
Monte-Carlo Tree Search (MCTS)

2) **Expansión**. Creación de nodos a partir del seleccionado.



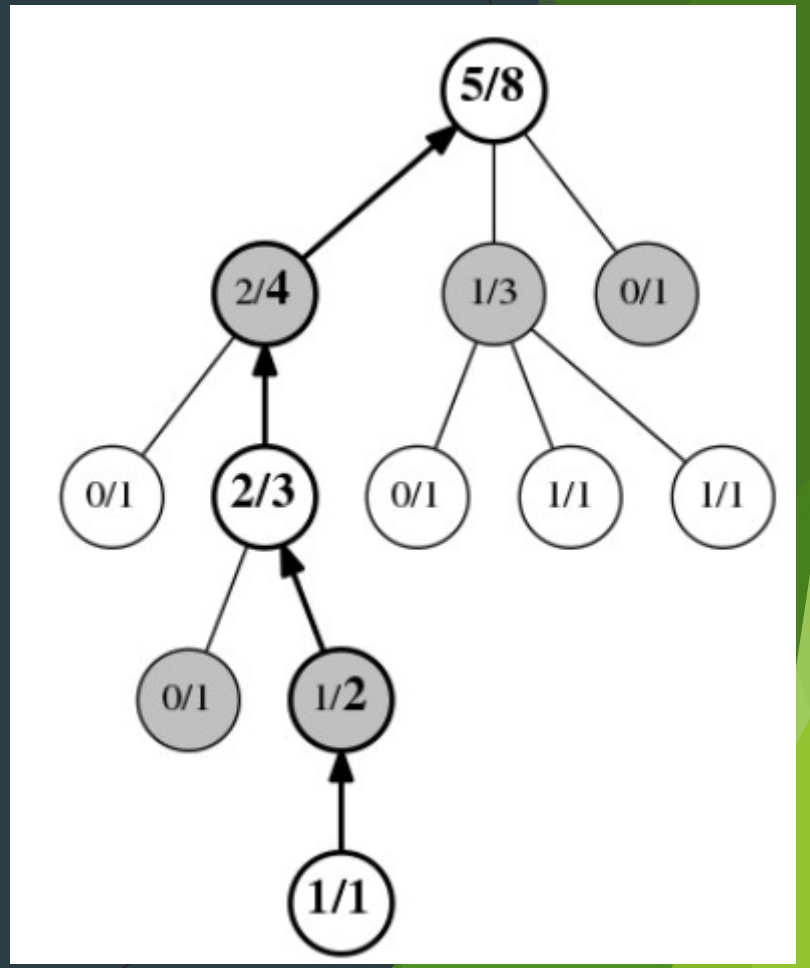
Monte-Carlo Tree Search (MCTS)

3) **Simulación**. Se simula un juego a partir del nodo expandido.



Monte-Carlo Tree Search (MCTS)

4) **Backpropagation**. El resultado de la simulación se retropropaga hacia el nodo raíz.



Monte-Carlo Tree Search (MCTS)

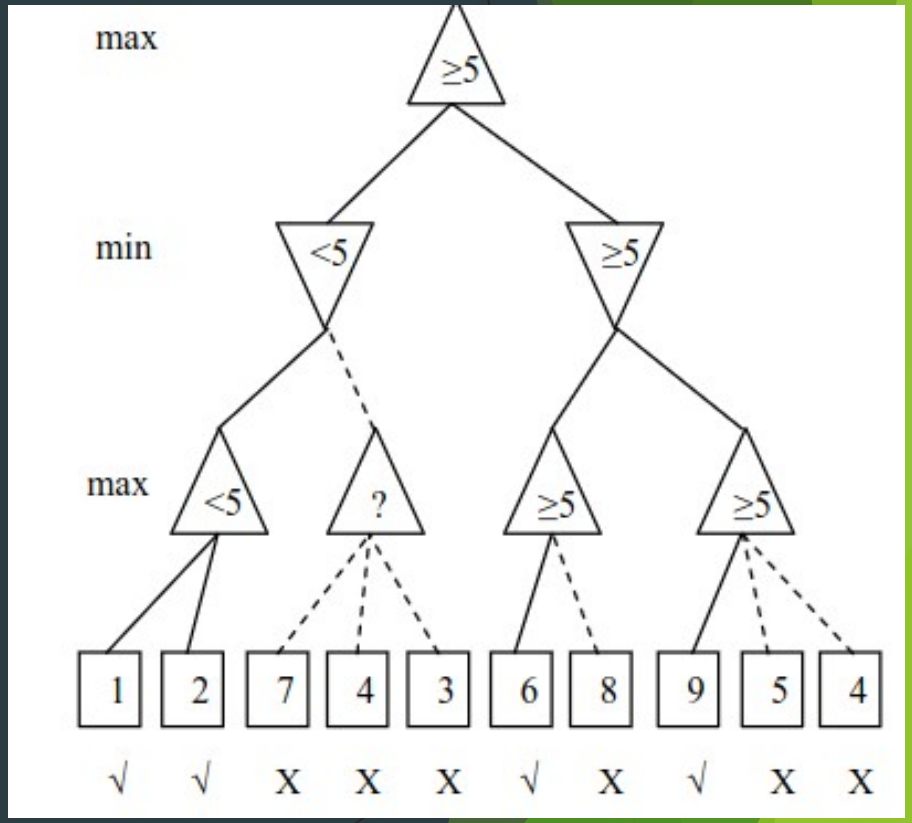
- Ventajas:
  - **Independiente del dominio**. Aplicable a gran variedad de juegos.
    - Sólo requiere conocer las reglas para generar movimientos y para finalizar la partida.
  - UCB y UCT tienen garantías de convergencias bajo condiciones relativamente laxas.
  - Puede obtener gran ventaja del conocimiento específico del juego cuando está presente.
    - Mayor eficiencia al realizar las exploraciones y las simulaciones cuando se pueden ordenar de mejor a peor los movimientos a priori.
    - Redes neuronales entrenadas sobre partidas de expertos para generar respuestas en la simulación y reforzadas con aprendizaje por refuerzo con partidas contra sí mismos (AlphaGo y AlphaZero).
  - Respuesta instantánea.
    - Se puede parar el algoritmo en cualquier momento y obtener una respuesta, aunque mejora la calidad de la respuesta con más tiempo.

Monte-Carlo Tree Search (MCTS)

- Desventajas:
  - Requiere gran cantidad de memoria.
  - Se requiere gran cantidad de iteraciones para obtener una buena estimación.
  - Necesita encontrar un buen balance entre exploración y explotación para ser eficiente y efectivo.
  - En aplicaciones reales requiere de una buena heurística para guiar la búsqueda y poder obtener buenos resultados de manera eficiente.
  - Overfitting. En algunas situaciones puede sobreestimar ciertos patrones presentes en simulaciones iniciales y generar decisiones subóptimas.

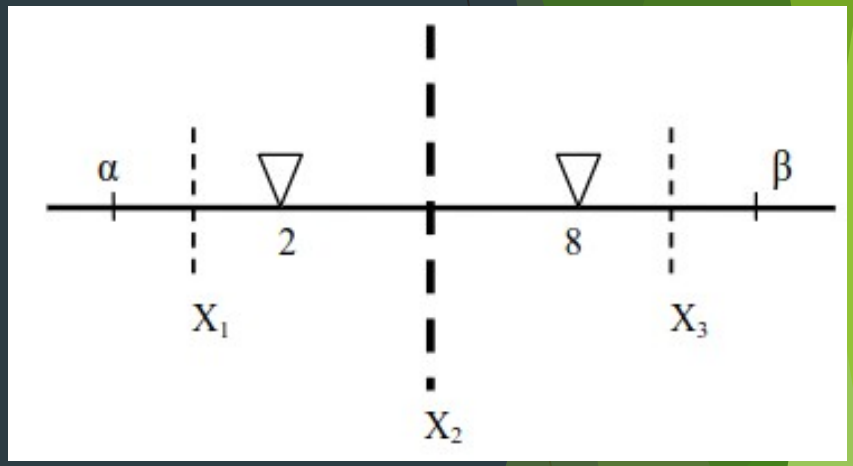
Búsqueda de mejor nodo (BNS)

- Intento de implementar un modo de pensamiento humano basado en una evaluación aproximada de las posiciones.
- No interesa la evaluación exacta, sino el nodo que garantiza un mejor resultado.
- Se comprueba si un subárbol es mejor o peor que un cierto valor X.
- Se realizan diferentes cortes:
  - En el nivel MAX, si la evaluación es mayor o igual que el valor de búsqueda.
  - En el nivel MIN, si la evaluación es menor que el valor de búsqueda.



Búsqueda de mejor nodo (BNS)

- La idea es encontrar un valor de test que permita dividir las ramas de forma que solo una de ellas tenga un valor superior.
- En el ejemplo, si se selecciona el valor X_2 se consigue la separación. En caso de seleccionar X_1 o X_3, se debe ajustar la ventana de alfa-beta con el valor X_1 o X_3 y seguir iterando.
- En el caso de haber tres o más subárboles, el objetivo es el mismo: conseguir que solo una rama tenga una estimación mayor que el valor seleccionado.
- No se requiere saber el valor exacto de minimax para el subárbol para seleccionar un movimiento.



Búsqueda de mejor nodo (BNS)

```
function nextGuess(α, β, subtreeCount) is
    return α + (β − α) × (subtreeCount − 1) / subtreeCount


function bns(node, α, β) is
    subtreeCount := number of children of node

    do
        test := nextGuess(α, β, subtreeCount)
        betterCount := 0
        for each child of node do
            bestVal := −alphabeta(child, −test, −(test − 1))
            if bestVal ≥ test then
                betterCount := betterCount + 1
                bestNode := child
        (update number of sub-trees that exceeds separation test value)
        (update alpha-beta range)
    while not (β − α < 2 or betterCount = 1)

    return bestNode
```