Técnicas de Inteligencia Artificial

Tema 3.3

Optimizaciones en búsqueda adversaria

Búsqueda en juegos en la práctica

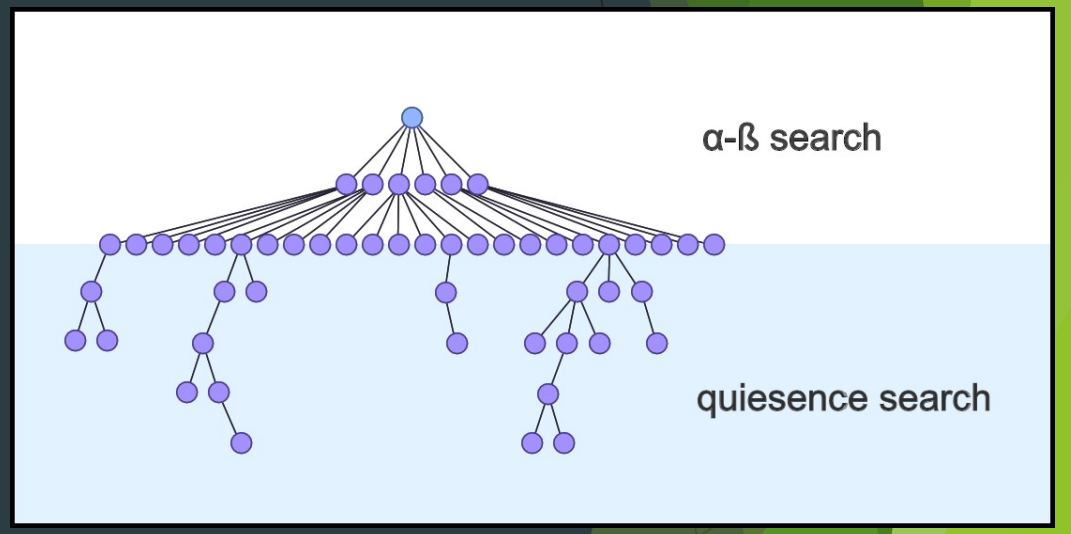
- En aplicaciones prácticas con limitaciones de tiempo, se requiere tener ciertas consideraciones:
  - Uso de heurísticas para valorar la posición actual del juego sin ser nodo terminal.
  - Reordenación de nodos del mejor al peor.
  - Descartar de manera temprana ciertas variantes según funciones de evaluación o estadísticas (forward pruning).
  - Usar movimientos que se consideraron como el mejor en situaciones anteriores.
  - Evitar estados repetidos. Tablas hash, conocidas como tablas de transposición.
  - Ajustar de manera dinámica los esquemas de ordenamiento de movimientos.
  - Libros de aperturas con las primeras jugadas, y bases de datos de partidas previas jugadas por profesionales.
  - Usar la búsqueda profundidad iterativa (IDS): permite cortar la búsqueda por requerimientos de tiempo y proporcionar pistas de ordenación de movimientos para iteraciones futuras.
  - Expandir movimientos forzados que cambien drásticamente la evaluación de la posición.

Efecto horizonte

- El efecto horizonte está causado por la limitación de profundidad de un algoritmo de búsqueda.
- Se pone de manifiesto cuando un evento negativo es inevitable pero postergable.
- Debido a que solo se analiza una parte del árbol de juego, puede parecer que un evento puede ser evitado cuando no es el caso, y su efecto aparecerá a un nivel de profundidad que no se ha llegado a poder analizar.
- Se considera que la exposición al efecto horizonte es uno de los factores limitantes más importantes para los módulos de juego, impidiendo evaluar de manera correcta una posición.
- Uno de los remedios para este efecto es realizar la evaluación de la posición solamente en “posiciones tranquilas”, haciendo uso de la búsqueda de quiescencia.

Búsqueda de quiescencia

- Se basa en la idea de la búsqueda en profundidad variable.
- El algoritmo sigue la búsqueda en profundidad fija en la mayoría de ramas.
- Sin embargo, en algunas ramas el algoritmo aumenta la profundidad de búsqueda hasta alcanzar posiciones tranquilas, en las que la evaluación estática de la posición no cambie drásticamente en turnos consecutivos.
- Su objetivo es evitar el efecto horizonte teniendo en cuenta las jugadas que pueden afectar de manera significativa a la evaluación de la posición.
- Se utiliza una evaluación estática (standing pat) para realizar descartes tempranos de nodos en las que la desventaja de un bando es notoria.

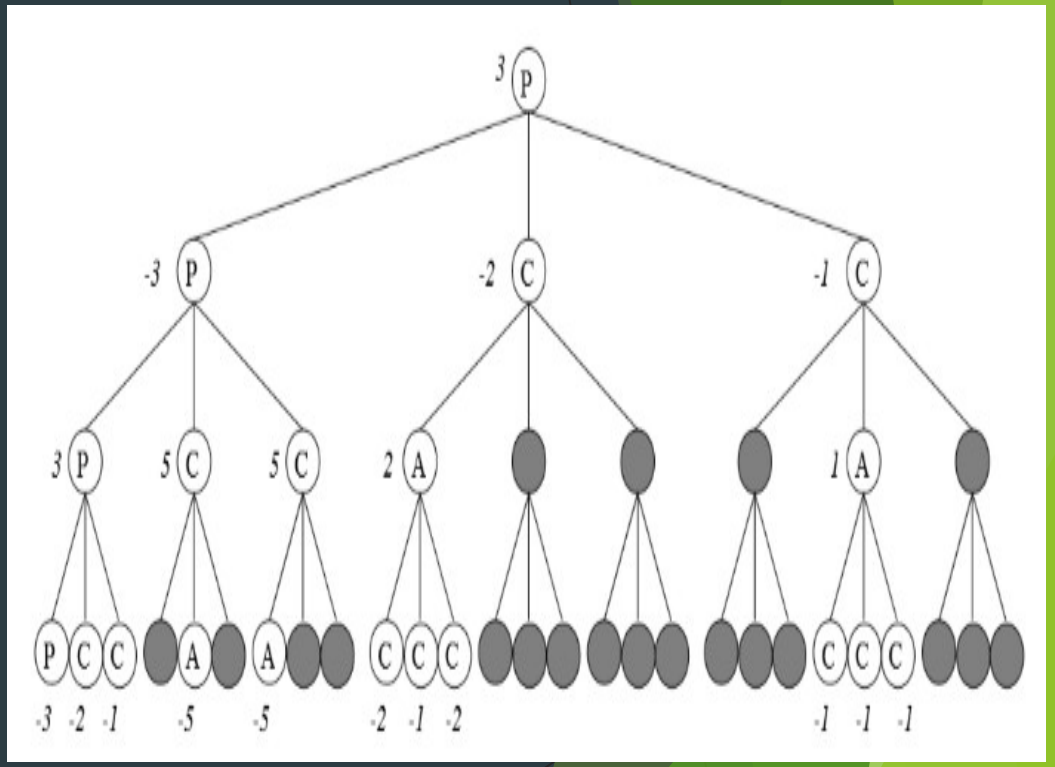


Búsqueda de quiescencia

```
int Quiesce( int alpha, int beta ) {
    int stand_pat = Evaluate();
    if( stand_pat >= beta )
        return beta;
    if( alpha < stand_pat )
        alpha = stand_pat;

    until( every_capture_has_been_examined )  {
        MakeCapture();
        score = -Quiesce( -beta, -alpha );
        TakeBackMove();

        if( score >= beta )
            return beta;
        if( score > alpha )
           alpha = score;
    }
    return alpha;
}
```

Forward pruning

- Técnica de optimización que consiste en descartar ciertos movimientos basándose en ciertos criterios heurísticos, sin hacer una búsqueda o con una búsqueda muy reducida.
- Los humanos solo consideran una pequeña porción de movimientos prometedores en cada posición.
- La idea consiste en realizar una evaluación previa de las jugadas candidatas y considerar solamente las más prometedoras.
- Se corre el riesgo de eliminar el mejor movimiento con esta técnica.



N-Best Selective Search

- Técnica de forward pruning.
- Considera solamente los N-mejores movimientos.
- Utiliza una función de evaluación estática para determinar la puntuación.
- La selección de la función de evaluación es muy importante.
- Se debe considerar un compromiso entre reducir el corte del mejor movimiento y el incremento de la profundidad de búsqueda.

ProbCut

- Técnica de forward pruning.
- Se basa en la idea de que las evaluaciones obtenidas en búsquedas a diferente profundidad están directamente correlacionadas.
  - El resultado $V_{D'}$ de una búsqueda superficial a una altura d' puede ser usada para predecir el resultado de V_D de una búsqueda más profunda a una altura d.
- A partir del valor devuelto se calcula la probabilidad de que una búsqueda más profunda caiga dentro de la ventana alfa-beta actual.
  - Si es alta, se ignora una búsqueda más profunda ya que es poco probable que afecte al resultado.
  - En caso contrario, se realiza una búsqueda más profunda para obtener el resultado preciso.

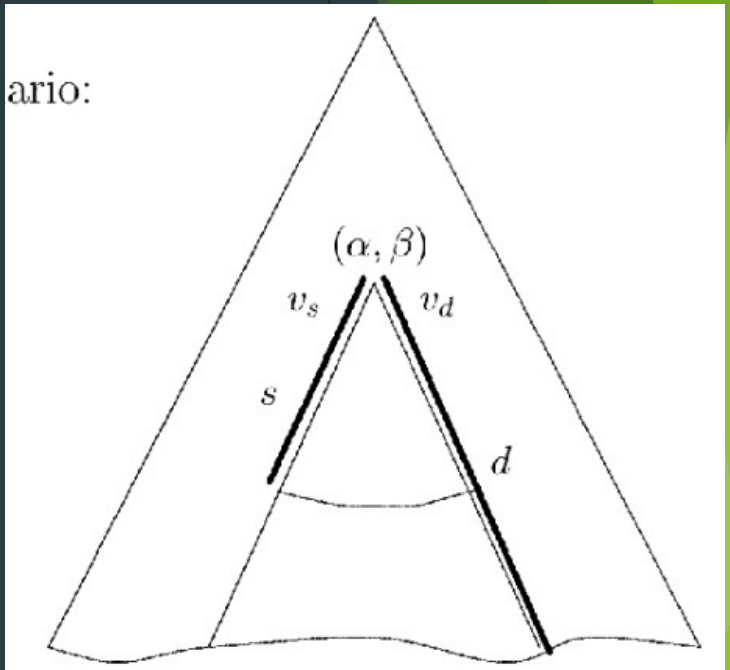
ProbCut

```
int alphaBetaProbCut(int α, int β, int depth) {
   const float T(1.5);
   const int DP(4);
   const int D(8);

   if ( depth == 0 ) return evaluate();

   if ( depth == D ) {
      int bound;

      /* v >= β with prob. of at least p? yes => cutoff */
      bound = round( ( T * σ + β - b) / a );
      if ( alphaBetaProbCut( bound-1, bound, DP) >= bound )
         return β;

      /* v <= α with prob. of at least p? yes => cutoff */
      bound = round( (-T * σ + α - b) / a );
      if ( alphaBetaProbCut( bound, bound+1, DP) <= bound )
         return α;
   }
   /* the remainder of alpha-beta goes here */
   ...
}
```

ProbCut

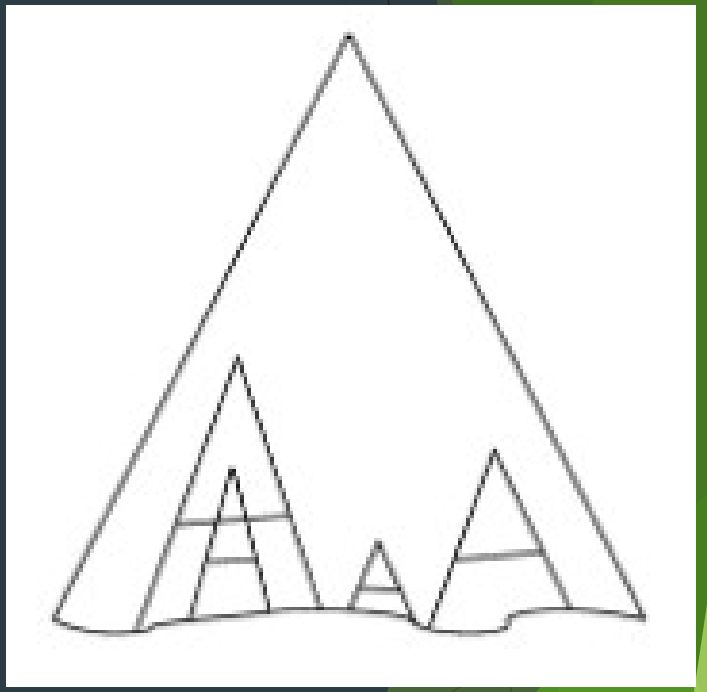
- $V_D = a * V_{s'} + b + e$ $a, b \in \mathcal{R}$ $e \sim \mathcal{N}(0, \sigma^2)$
  - Los valores a, b y e son parámetros que deben aprenderse y ajustarse.
- Cortar el árbol si no está dentro de los límites alfa-beta

$$a \cdot v_s \notin (\alpha - b - T \cdot \sigma \, , \, \beta - b + T \cdot \sigma)$$



Multi-ProbCut

- Técnica de forward pruning.
- Optimización de ProbCut.
  - Permite cortar subárboles irrelevantes a diferentes alturas en vez de a una profundidad específica.
  - Al realizar diferentes búsquedas de comprobación de profundidad creciente, puede detectar movimientos muy malos con búsquedas superficiales de profundidades muy bajas.
  - Optimiza los umbrales de corte separadamente para cada fase de la partida en vez de usar los mismos para todo el juego.



Multi-ProbCut

```
struct Param {
   int d;          /* shallow search depth */
   float t;        /* cut threshold */
   float a, b, σ; /* slope, offset, standard deviation */
} param[MAX_STAGE+1][MAX_HEIGHT+1][NUM_TRY];
```

```
int alphaBetaMPC(int α, int β, int depth)
{
   if ( depth == 0 ) return evaluate();

   if ( depth <= MAX_D ) {
      for (int i=0; i < NUM_TRY; i++) {
         int bound;
         const Param &pa = param[stage][depth][i];

         if (pa.d < 0 )
            break; /* no more parameters available */

         /* v >= β with prob. of at least p? yes => cutoff */
         bound = round( ( pa.t * pa.σ + β - pa.b) / pa.a );
         if ( alphaBetaMPC( bound-1, bound, pa.d) >= bound )
            return β;

         /* v <= α with prob. of at least p? yes => cutoff */
         bound = round( (-pa.t * pa.σ + α - pa.b) / pa.a );
         if ( alphaBetaMPC( bound, bound+1, pa.d) <= bound )
            return α;
      }
   }
   /* the remainder of alpha-beta goes here */
   ...
}
```

Función de evaluación heurística

- Aunque lo deseable es alcanzar un estado terminal, donde la valoración de la posición (victoria, derrota o empate) es objetiva, muchas veces no es posible alcanzar ese estado en tiempo finito.
- Para guiar los algoritmos de búsqueda se requiere de una función de evaluación que nos indique qué bando tiene ventaja en una determinada posición.
- Históricamente, los términos de la función de evaluación se han construido manualmente utilizando el conocimiento de expertos en un determinado juego.
- El éxito de la función de evaluación consiste en ajustar el peso de cada uno de los términos para aproximarse lo más posible a la evaluación objetiva de la posición.
- En los últimos años, la función de evaluación se aprende mediante aprendizaje automático utilizando redes neuronales y técnicas de aprendizaje por refuerzo o aprendizaje supervisado.

Función de evaluación heurística

- Una buena función de evaluación debe tener las siguientes propiedades:
  - Evaluar los nodos terminales de forma perfecta.
  - Ejecutarse muy rápidamente.
  - Ser precisa en las probabilidades de victoria para nodos no terminales.
- Una aproximación común consiste en una combinación lineal de varios términos ponderados según su influencia en la valoración de la posición: $w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$
  - Sin embargo, los pesos pueden variar durante la partida.
  - La combinación puede ser no lineal.
- Los pesos provienen de dos fuentes principales:
  - Humanos expertos.
  - Aprendizaje automático (a menudo refinados con miles de partidas de máquinas consigo mismas).

Heurística histórica

- Técnica para mejorar el ordenamiento de movimientos sin conocimiento específico del juego.
- Otorga una puntuación adicional a movimientos que provocaron cortes en una búsqueda para ser priorizados en búsquedas sucesivas.
- Se basa en la observación de que un movimiento fuerte en una posición particular debe ser de fuerza similar en posiciones parecidas del árbol de juego.
- En aplicaciones prácticas, los módulos de juego suelen almacenar dos movimientos de este tipo por cada profundidad del árbol de juego, y comprueban inicialmente si estos movimientos provocan un corte antes de generar y considerar el resto de alternativas.
- En su implementación original se implementa como una tabla indexada por alguna característica del movimiento, como el origen y el destino de una pieza. Cuando se produce un corte, la casilla correspondiente es incrementada con un factor dependiente de la profundidad de la búsqueda.

Heurística histórica

```
AlphaBeta( p : position; α, β, depth : integer ) : integer;
var
    bestmove, score, width, m, result : integer;
    moves, rating : array[ 1..MAX_WIDTH ] of integer;

begin
    if depth = 0 then          { At a leaf node }
        return( Evaluate( p ) );

    width := GenerateMoves( moves );
    if width = 0 then          { No moves in this position }
        return( Evaluate( p ) );

    { Assign history heuristic score to each move }
    for m := 1 to width do
*       rating[ m ] := HistoryTable[ moves[ m ] ];
    Sort( moves, rating );     { Put highest rated moves first }

    score := -∞;
    for m := 1 to width do
       begin
         { Recurse using nega-max variant of αβ }
         result := -AlphaBeta( p.moves[m], -β, -α, depth-1 );
         if result > score then
             score := result;

         { Check for cut-off }
         if score >= β then
           begin
             { Cut-off: no further sub-trees need be examined }
             bestmove := moves[ m ];
             goto done;
           end;

         α := MAX( α, score );
       end;

  done:
    { Update history score for best move }
*   HistoryTable[ bestmove ] := HistoryTable[ bestmove ] + 2^depth;
    return( score );
end.
```

Heurística de movimiento nulo

- El movimiento nulo (null-move) es otra forma de llamar a “pasar turno”.
- Significa que un bando pasa y permite al oponente realizar dos movimientos consecutivos.
- Se basa en la observación del movimiento nulo (null move observation) que dice que, en casi cualquier posición, el bando que mueve tiene un movimiento mejor que permanecer quieto.
- Si el oponente puede mejorar su posición después de un movimiento nulo, se requiere realizar una búsqueda ordinaria, ya que la línea no es suficiente mala para él.
- Si a pesar de poder realizar dos movimientos consecutivos no consigue mejorar su posición significativamente, se puede podar el subárbol completo ya que es muy probable de que se trate de una variante muy mala.
- El movimiento nulo también es muy usado para la detección de amenazas del contrario y poder mejorar la ordenación de movimientos del árbol de búsqueda.
- Uno de sus principales problemas es el zugswang, posición donde la obligación de jugar hace perder al bando en movimiento, que esta técnica no permite detectar.

Heurística de movimiento nulo

```
/* the depth reduction factor */
#define R  2
int search (alpha, beta, depth) {
   if (depth <= 0)
      return evaluate(); /* in practice, quiescence() is called here */
   /* conduct a null-move search if it is legal and desired */
   if (!in_check() && null_ok()){
      make_null_move();
      /* null-move search with minimal window around beta */
      value = -search(-beta, -beta + 1, depth - R - 1);
      if (value >= beta) /* cutoff in case of fail-high */
         return value;
   }
   /* continue regular NegaScout/PVS search */
   ...
}
```

R: depth reduction factor

Tablas de transposición

- La tabla de transposición es una caché de posiciones anteriormente vistas con su evaluación asociada y la de sus subárboles.
- Permite evitar búsquedas adicionales sobre la misma posición, sobre todo en juegos en los que se puede llegar a trasponer mediante secuencias diferentes de movimientos.
- Se implementan típicamente usando tablas hash, codificando la posición del tablero como él índice hash.
- La consulta de una posición en la tabla de transposición es muy rápida, en tiempo constante.
- Al haber una gran cantidad de posiciones posibles, las tablas de transposición pueden llegar a consumir una enorme cantidad de memoria.
- Cuando se superan las restricciones de memoria del sistema, se borran las posiciones menos usadas para dejar espacio a las nuevas.

Zobrist hashing

- Función de hash usada para implementar tablas de transposición.
- Se compone de arrays de bits generados aleatoriamente para cada posible elemento del tablero de juego.
- Por ejemplo, para representar una posición de ajedrez, se generará un array de bits para cada combinación de una pieza y una posición.
  - Cada configuración del tablero se puede dividir en componentes independientes pieza-posición y mapearse en uno de los arrays de bits generados aleatoriamente.
- El hash de Zobrist resultante se calcula mediante la combinación de los arrays de bits usando una operación XOR.
- Si los arrays son de un tamaño suficientemente grande, diferentes posiciones tendrán valores distintos de hash sin colisiones.
- Muchos módulos solo almacenan el valor de hash en la tabla de transposición, asumiendo que las colisiones no ocurrirán o no tendrán una gran influencia en los resultados.

Zobrist hashing

```
constant indices
    white_pawn := 1
    white_rook := 2
    # etc.
    black_king := 12

function init_zobrist():
    # fill a table of random numbers/bitstrings
    table := a 2-d array of size 64×12
    for i from 1 to 64:  # loop over the board, represented as a linear array
        for j from 1 to 12:      # loop over the pieces
            table[i][j] := random_bitstring()
    table.black_to_move = random_bitstring()

function hash(board):
    h := 0
    if is_black_turn(board):
        h := h XOR table.black_to_move
    for i from 1 to 64:      # loop over the board positions
        if board[i] ≠ empty:
            j := the piece at board[i], as listed in the constant indices, above
            h := h XOR table[i][j]
    return h
```

Zobrist hashing

- En vez de calcular el valor de hash para todo el tablero, el valor se puede actualizar incrementalmente aplicando la función XOR sobre los array de bits de las posiciones que han cambiado.
- Por ejemplo, si en ajedrez un peón ha sido reemplazado por una torre procedente de otra casilla, se generaría la secuencia de operaciones XOR del ejemplo.
- Esta característica hace la función de hashing de Zobrist muy eficiente para la búsqueda sobre árboles de juego.

```
'pawn at this square'     (XORing out the pawn at this square)
'rook at this square'     (XORing in the rook at this square)
'rook at source square'   (XORing out the rook at the source square)
```

Libros de aperturas

- En la mayoría de juegos la cantidad de movimientos iniciales es muy grande.
- Resulta casi imposible analizar su evaluación ya que el nivel de profundidad requerido es enorme y la cantidad de movimientos crece de forma exponencial.
- El libro de aperturas es una base de datos que contiene los mejores movimientos de los instantes iniciales de la partida.
- Originalmente, esta base de datos se construía basándose en un análisis estadístico de los resultados de partidas entre maestros y los análisis hechos por expertos.
- Actualmente, los libros de aperturas se refinan con las partidas jugadas entre máquinas y los análisis profundos realizados por computadores dedicadas durante días y semanas a analizar determinadas posiciones.
- Una vez uno de los jugadores realiza un movimiento fuera del libro, se activan los algoritmos de búsqueda normales.
- Habitualmente, se suelen implementar utilizando valores de hash similares a los de las tablas de transposición, junto con información estadística adicional.

Tablas de finales

- Base de datos que contiene un análisis exhaustivo precalculado de una determinada posición en la que el material de juego es muy limitado y la partida se encuentra en su fase final.
- Contiene el valor teórico (victoria, derrota o empate) de cada posición posible, así como la cantidad de movimientos para alcanzar el resultado con juego perfecto. La base de datos actúa como un oráculo que siempre da como respuesta los movimientos óptimos.
- Estas bases de datos se generan mediante un análisis retrógrado, partiendo desde una posición terminal de la partida, y retrotrayéndose hacia atrás en el árbol de variantes.
- Requieren de una gran capacidad de memoria para almacenar los resultados.
- Las soluciones encontradas han permitido avanzar en la teoría de finales de algunos juegos cambiando la valoración de ciertas posiciones, previamente evaluadas por humanos de manera errónea.
- Aunque los avances más significativos se han alcanzado en el ajedrez, otros módulos de análisis de juegos como las damas o el juego del molino también lo utilizan.