# Trabajo Práctico Especial

## Estructura de Datos y Algoritmos (72.34)


**Juan Godfrid - 56609**
**Francisco Delgado - 57101**
**Lucas Emery - 57341**
**Ximena Zuberbuhler - 57287**
**Agustín Calatayud - 57325**

# Contenido

# Introducción: Un juego milenario

El Go es un antiguo juego de tablero estratégico, profundamente arraigado en la cultura oriental. Recientemente ha surgido una explosión de popularidad e interés en el mundo occidental por su complejidad de resolver computacionalmente y como caso de estudio en la teoría de juego. El Go tiene la particularidad de ser un juego con reglas sencillas de aprender y de ser fácil de comenzar a jugar, pero la brecha en nivel entre juego casual y profesional es colosal: se requiere miles de horas de estudio para comenzar a jugar profesionalmente. El tablero de Go profesional es de 19x19 espacios, mientras que los principiantes suelen jugar con tamaños más pequeños. A fines de practicidad, optamos por un tablero con dimensión de 13x13 espacios.

El objetivo de este trabajo práctico especial es implementar dicho juego en java y, más ambiciosamente, crear un sistema inteligente que lo pueda jugar. El sistema de inteligencia que juegue será basado en el algoritmo minimax.

En este informe se explicarán las decisiones que fueron tomadas en cuanto a la arquitectura del proyecto, las estructuras de datos, los algoritmos y la heurística. Finalmente se hará un análisis a modo de conclusión en cuanto a la eficacia de las decisiones tomadas y la utilización de minimax para una inteligencia que juegue al Go.

# Estructuras

## Dot Builder

Para la escritura del archivo dot se consideraron tres alternativas, la primera fue que se guardará el árbol generado por el minimax y que una vez elegido el mejor movimiento, se explorará ese árbol generando el archivo dot. Esta opción no era la más eficiente tanto porque se recorría el árbol nuevamente luego de haber terminado el minimax y porque se debían guardar todos los nodos de modo que no sería eficiente en memoria. La segunda opción consistía en generar un string a medida que el minimax buscaba el movimiento, con lo que no se exploraba nuevamente el árbol, pero se seguía guardando en memoria el string conteniendo la información para el dot que finalmente se exportaría al archivo. Finalmente la opción elegida fue la de generar el archivo dot a medida que se exploraba el minimax, de esta forma no se utiliza espacio en memoria para guardar el contenido del archivo ni se explora el árbol nuevamente para formar el archivo.

## Tabla de Zobrist

Para implementar Zobrist hashing se utilizó una tabla de 169x3 representando las 169 posibles posiciones y los 3 posibles estados de cada posición, ellos siendo vacío blanco y negro. Cada posición de la matriz tiene un bitstring de 63 bits generado de manera aleatoria cuando comienza el programa. el bitstring en la posición ( i , j ) se utilizará para indicar que en la posición i se encuentra la pieza j cuando se arme el hash.

Cada tabla mantiene un vector de 169 posiciones llamado *zobrist indices* el cual indica qué pieza (0, 1 o 2) se encuentra en la posición i en el tablero en ese estado particular. Esto se utilizará en el algoritmo de *zobrist hashing* junto a los bitstrings previamente mencionados.

el algoritmo que se basa en esta estructura se encuentra en la sección de algoritmos.

# Algoritmos

## Cálculo de territorio

La función de cálculo de territorio es necesaria tanto para la heurística como para el sistema de puntos, el funcionamiento del algoritmo consiste en recorrer todo el tablero aplicando un *Flood Fill* (o *algoritmo de relleno*) como sub algoritmo para calcular de quién es el territorio del punto en el cual está parado. Si la posición en la que está parado fue previamente inundada, no lo calcula nuevamente. El subalgoritmo “inunda” el terreno en el que se encuentra hasta haber recorrido todo el territorio contenido dentro de fichas de jugadores o extremos del tablero. Los territorios inundados y delimitantes son retornados y si son exclusivamente delimitados por piezas de un solo jugador, el territorio cuenta como perteneciente a dicho jugador. El pseudocódigo del sub algoritmo es el siguiente:

```
function floodFill()
        if coordinates out of board
                add new border
                return borders

        if space is visited
                if space is occupied
                        add new terrain
                return borders
        else
                mark space as visited

        if space is not empty
                add new border
                return borders
        else
                add new terrain

        floodfill(north)
        floodfill(east)
        floodfill(south)
        floodfill(west)
    return borders
```

La complejidad del algoritmo es $O(n^2)$ siendo n la altura del tablero.

## Negamax

Para recorrer el árbol de jugadas utilizamos *Negamax*. Es una variante de *Minimax,* la cual se basa en que Go es un juego de suma cero, es decir, que lo que es favorable para un jugador es igualmente desfavorable para el otro. Por lo tanto, el valor heurístico de un movimiento para un jugador es la negación del valor heurístico para el otro.

Como -min(-a,-b) = max(a, b), negamax siempre maximiza y simplemente invierte el signo del valor heurístico obtenido en el siguiente nivel del árbol de jugadas. Esto es más sencillo de observar en el pseudocódigo a continuación:

```
function negamax(node, depth, player)
    if depth == 0 or node is a terminal node
        return the heuristic value of node from player's perspective

    childNodes = generateMoves(node)
    bestValue = −∞
    foreach child in childNodes
        v = −negamax(child, depth − 1, otherPlayer)
        bestValue = max( bestValue, v )
    return bestValue
```

Sea b el branching factor del árbol de jugadas y d la profundidad hasta la que se quiere buscar, la complejidad del algoritmo es $O(b^d)$.

## Poda Alpha-Beta

Para mejorar el rendimiento del negamax se le agregó Poda Alpha-Beta. Este algoritmo elimina las ramas del árbol de jugadas que no aportan al resultado final. Se basa en que si se encontró una prueba de que el movimiento de esa rama es peor que uno hallado entonces este no será elegido por el algoritmo y por ende no afectará al resultado final. Por esto los movimientos de la rama no necesitan ser evaluados pues el valor final del nodo sería peor que el que fue encontrado previamente.

```
function negamax(node, depth, α, β, player)
    if depth == 0 or node is a terminal node
        return the heuristic value of node from player's perspective

    childNodes = generateMoves(node)
```

```
bestValue = −∞
foreach child in childNodes
   v = −negamax(child, depth − 1, −β, −α, otherPlayer)
   bestValue = max( bestValue, v )
   α = max( α, v )
   if α ≥ β
      break
return bestValue
```

Sea b el branching factor del árbol de jugadas y d la profundidad hasta la que se quiere buscar, si los movimientos no están ordenados la complejidad del algoritmo es aproximadamente $O(b^{3d/4})$.

## History Heuristic

Si se ordenan los movimientos antes de hacer el minimax, la poda alpha-beta resulta mucho más eficiente. Para esto se utilizó lo que se llama *History Heuristic*, que consiste en llevar un historial de los movimientos que produjeron podas en otras ramas y se ordenaron los hijos de cada nodo según este criterio. Se utiliza *depth* al cuadrado (La variable *depth* es cero cuando se llega a una hoja) para dar menos peso a los cortes en profundidades mayores. El pseudocódigo del negamax con alpha-beta + history heuristic es el siguiente:

```
function negamax(node, depth, α, β, player)
   if depth == 0 or node is a terminal node
      return the heuristic value of node from player's perspective

   childNodes = generateMoves(node)
   childNodes = orderMoves(childNodes)
   bestValue = −∞
   foreach child in childNodes
      v = −negamax(child, depth − 1, −β, −α, otherPlayer)
      bestValue = max( bestValue, v )
      α = max( α, v )
      if α ≥ β
          historyMap[player-1][child.y][child.x] += depth*depth;
         break
   return bestValue
```

Sea b el branching factor del árbol de jugadas y d la profundidad hasta la que se quiere buscar, si el ordenamiento es óptimo la complejidad del algoritmo es de $O(b^{d/2})$.

## Scout Layer

Hasta ahora, cuando se ordenan los hijos del nodo raíz el history heuristic todavía no tiene valores, por lo que el ordenamiento es aleatorio. Para poder ordenar los hijos del nodo raíz se los evalúa con la función heurística y se los ordena. Esto equivale a hacer una búsqueda de profundidad 1 antes de hacer la búsqueda completa. Scout layer solo se activa cuando se realizan búsquedas de profundidad mayor a 3, ya que con profundidades menores tiene un impacto negativo. El pseudocódigo es el siguiente:

```
function negamax(node, depth, α, β, player)
    if depth == 0 or node is a terminal node
        return the heuristic value of node from player's perspective

    childNodes = generateMoves(node)
    if scoutLayer == true
        scoutLayer = false
        foreach child in childNodes
            child.weight = the heuristic value of child from player's perspective
    childNodes = orderMoves(childNodes)
    bestValue = −∞
    foreach child in childNodes
        v = −negamax(child, depth − 1, −β, −α, otherPlayer)
        bestValue = max( bestValue, v )
        α = max( α, v )
        if α ≥ β
            historyMap[player-1][child.y][child.x] += depth*depth;
            break
    return bestValue
```

La variable *scoutLayer* es un atributo de AI y se setea antes de llamar al negamax.

En profundidad 4 el algoritmo es un 50% más rápido que sin Scout Layer.

## Algoritmo de Hashing para las tablas

las tablas se *hashean* acorde al *zobrist hasing*, como vimos en la sección de estructuras, se provee los bitstrings de los estados posibles de un casillero junto a qué piezas se encuentran en cada lugar.

Para construir el hash inicial, o si se desea hashear toda la tabla, se recorren las 169 posiciones del tablero, para cada posición se selecciona el bit string que representa la posición y el estado de la misma y se hace un bitwise xor con el hash previo(0 si es el primero), así con todas las posiciones hasta obtener un hash de 63 bits.

Si se desea *rehashear* un tablero y se conoce el estado previo, para cada posición del tablero que cambie:

- se remueve el estado del hash haciendo un xor con el bit string que representa el estado de ese casillero, obtenido de la tabla de zobrist.
- se inserta el estado nuevo al hash haciendo un xor con el bit string que representa el estado nuevo en ese casillero, nuevamente obtenido de la tabla de zobrist.

Aquí se pone en evidencia por qué se mantiene el vector de zobrist índices: este nos permite rápidamente acceder a el estado del tablero para updatear los hashes.

Se aclara que este algoritmo de hashing no es de nuestra autoría sino que fue inventado por Albert Lindsey Zobrist. En un principio se pensó basarse en hashes provistos por java. Pero luego de investigar sobre cómo se plantean los hashes en proyectos de Go minimax tales como GNU GO se encontró zobrist hashing como una opción muy eficiente debido a la simplicidad del *hasheo* en donde como máximo se realizan 169 xors cuando se inicializa, además la performance de *rehashear* conociendo el estado previo es muy buena al tratarse de simples xor a nivel bit, esto es lo que se hace en gran parte ya que nos basamos en el estado previo para generar los posibles nuevos estados.

## Algoritmo de Bouzy

Para estimar el territorio potencial y la influencia de cada jugador se utiliza el algoritmo de Bouzy. Este consiste en dos operaciones simples: Dilatación y Erosión. Para obtener buenos resultados se utilizan 5 dilataciones y 21 erosiones, se pueden obtener otras combinaciones a partir de la siguiente fórmula:

$$E = 1 + D \times (D - 1)$$

Donde:

E es la cantidad de Erosiones

D es la cantidad de Dilataciones.

El pseudocódigo de cada operación se detalla a continuación:

```
function dilation()
   foreach intersection in bouzyMap
      if intersection >= 0 and no neighbors < 0
         intersection += number of neighbors > 0
      if intersection <= 0 and no neighbors > 0
         intersection −= number of neighbors < 0

function erosion()
   foreach intersection in bouzyMap
      if intersection > 0
         intersection −= number of neighbors <= 0
         if intersection < 0
            intersection = 0
      else if intersection < 0
         intersection += number of neighbors >= 0
         if intersection > 0
            intersection = 0
```

La matriz *bouzyMap* tiene las misma dimensiones que el tablero y se inicializa en cero donde no hay piedras, en un número positivo grande (500 por ejemplo) donde hay piedras del jugador que evalúa y en un número negativo grande (-500 por ejemplo) donde hay piedras del otro jugador. Las piedras son ignoradas cuando se calcula el territorio potencial y la influencia.

# Heurística

La función heurística se utiliza para determinar cuán favorable es un tablero para el jugador. Para esto se utilizan distintos criterios ponderados explicados a continuación:

- **Territorio:** En esta etapa se calculan los territorios de cada jugador y se hace la diferencia entre ambos, ya que lo que determina el ganador es quien tiene más territorio.

- **Capturas:** Las capturas valen lo mismo que un espacio de territorio y aportan de la misma manera a la puntuación. Por esta razón, el peso de una captura en el valor heurístico final es el mismo que el de un espacio de territorio. De esta manera, el valor de capturar una piedra es el doble que el de simplemente capturar un territorio porque ocurren ambos eventos simultáneamente.

- **Territorio Potencial:** El territorio potencial es una estimación de los espacios que podrían llegar a ser tomados por el jugador en jugadas futuras. Para obtenerlo se utilizó el algoritmo de Bouzy 5/21 (5 dilataciones y 21 erosiones). El peso del territorio potencial sobre el valor heurístico final es varias veces menor que el del territorio y las capturas. De no ser así, la computadora tiene buen control sobre el tablero pero no siempre concreta el territorio, lo que resultaba en un estilo de juego muy pasivo.

- **Influencia:** La influencia es una medida del “poder” sobre el tablero, es el impacto que tiene una piedra sobre los espacios a su alrededor. En nuestro modelo se comporta como una luz, su intensidad disminuye con la distancia hasta que es imperceptible. Determinamos que la distancia máxima de influencia de una piedra es de 5 espacios. Para calcularla de manera rápida utilizamos un resultado parcial del algoritmo de Bouzy, luego de realizadas las 5 dilataciones se analiza el mapa de Bouzy, se estima la influencia y se procede con las erosiones para el cálculo de territorio potencial. El peso de cada espacio sobre el que se tiene influencia en el valor heurístico final es mucho menor que el de un espacio de territorio potencial. Esto es porque una piedra puede tener más de 20 espacios de influencia y el control que tiene dicha piedra sobre esos espacios es bajo.
  La estimación de influencia le permite a la computadora tener una visión más amplia del tablero para no quedarse jugando en una esquina mientras cede el resto del tablero al contrincante.

La ecuación del valor heurístico final es:

$$V = 200 \times \Delta T + 200 \times \Delta C + 10 \times \Delta TP + 1 \times \Delta I$$

Donde:

V es el valor heurístico final
ΔT es la diferencia de territorio
ΔC es la diferencia de capturas
ΔTP es la diferencia de territorio potencial
ΔI es la diferencia de influencia

Si el jugador gana en el tablero evaluado, es decir, si ambos pasaron y tiene más puntos, el valor heurístico es de ese tablero es MAX-1 (El -1 está para que si o si reemplace el movimiento dummy con el que arranca el minimax).
Si el jugador empata o pierde, el valor heurístico de ese tablero es MIN+1 (Razón del +1 idem -1).

# Benchmarks

*promedios de tiempo de respuesta de la inteligencia artificial de 20 turnos*

| **Profundidades** | **Con *prune (ms)*** | **Sin *prune (ms)*** |
|---|---|---|
| **1** | 12.16 | 12.16 |
| **2** | 166.1 | 1631.63 |
| **3** | 4658 | 249509.25 |
| **4** | 26145.33 | *>300000* |

Es destacable notar que la profundidad 3 sin prune tarda más que la profundidad 4 con prune.

# Conclusión

En la toma de decisiones del proyecto, llegamos a la realización que el comportamiento de la inteligencia artificial (dada la restricción de tiempo para poder jugar en vivo) estaba sujeto a un balance entre la complejidad de la función heurística y la profundidad del árbol de jugadas: Si desarrollábamos una heurística sofisticada, la función requeriría más tiempo de cómputo y el árbol no podría llegar a profundidades muy grandes. En cambio si desarrollábamos una heurística simple, el árbol podría llegar considerablemente más profundo, pero con un menor pensamiento estratégico. Concluímos que era más productivo y educativo desarrollar una heurística más sofisticada y aceptar las consecuencias del mayor tiempo de cálculo.

En el estado actual del proyecto, realísticamente, no se puede mantener un ritmo de juego ágil con profundidades mayores a 3. Esto podría ser mejorado (y lo fué) con ingeniosos cambios que incrementen la eficiencia del sistema, pero el mayor factor limitante fue la elección algoritmo de la inteligencia artificial. El minimax es un algoritmo determinista que se basa en completar un juego y evaluarlo estáticamente. El Go comienza con un *branching factor* de $13^2$, es decir, cada nodo va a tener 169 hijos inicialmente. La naturaleza exponencial del árbol de jugadas impide un análisis presciente del juego ya que a mayores profundidades llega a un número inalcanzable de cálculos muy rápidamente. A futuro, se podría evaluar el uso de otros algoritmos para la inteligencia artificial, como el *Monte Carlo Tree Search*.

En cuanto al desempeño del algoritmo, logra vencer a algunos principiantes que no conocen su funcionamiento, pero luego de ser informados acerca de las restricciones de profundidad resulta muy simple desarrollar una táctica que toma en consideración la inhabilidad del sistema de ver más de dos jugadas a futuro. Tomando en cuenta las restricciones intrínsecas del sistema y la ambición del emprendimiento, consideramos satisfactorio el desarrollo del proyecto en su totalidad, tanto en valor pedagógico como en funcionamiento.

# Referencias

1. https://es.wikipedia.org/wiki/Go
2. http://www.delorie.com/gnu/docs/gnugo/gnugo_201.html
3. https://chessprogramming.wikispaces.com/History+Heuristic
4. https://en.wikipedia.org/wiki/Negamax
5. https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning