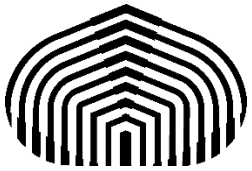


Universidad Simón Bolívar

CI-5437 Inteligencia Artificial

Prof. Carlos Infante

# INFORME DEL PROYECTO 2

Simon Puyosa 18-10717

Ana Shek 19-10096

Juan Cuevas 19-10056

Caracas, Marzo del 2024.

# Tabla de Contenido

# Introducción

La Inteligencia Artificial (IA) ha emergido como una disciplina transformadora que redefine la manera en que abordamos problemas complejos y tomamos decisiones en diversos campos. Su capacidad para imitar procesos cognitivos humanos y aprender patrones complejos la convierte en una herramienta invaluable en la resolución de problemas.

Dentro de los temas de estudio de la Inteligencia Artificial, se encuentra la Búsqueda Bidireccional, la cual es usada principalmente para resolver juego de 2 o más jugadores (En este caso estudiamos juegos de exactamente 2 jugadores), en donde cada jugador busca ganar la partida con acciones que perjudiquen al otro (y viceversa). La búsqueda bidireccional se encarga de buscar la mejor solución para el jugador de forma que obtenga el mayor puntaje y gane la partida.

Entre los algoritmos usados para la Búsqueda Bidireccional, se encuentran Negamax y Scout (cada uno con sus variaciones para aumentar su eficiencia). Los mismos, se encargan de maximizar o minimizar (un jugador busca el máximo y el otro el mínimo) los valores de los nodos del árbol de estados del juego, donde la raíz es el estado inicial. Esto hace que, al buscar las soluciones del árbol de juego; si todas las hojas tienen un valor mayor al inicial, entonces es una estrategia ganadora; en caso contrario es débilmente ganadora.

A continuación, el presente trabajo se enfoca en resolver árboles de juego con búsqueda bidireccional, el análisis de los algoritmos, como Scout y Negamax, aplicados a un problema concreto como el Othello. En este estudio se explorará la eficacia y la aplicabilidad de estos algoritmos en la resolución de problemas complejos, ofreciendo así una comprensión más profunda de su funcionamiento y sus posibles aplicaciones prácticas.

Los experimentos fueron realizados en la siguiente plataforma:

- Una computadora con procesador Intel Core i7-8700, disco SSD, 16GB de memoria RAM y Linux Mint 21.1 x86_64
- Un laptop con procesador AMD Ryzen 3 3200U, disco SSD, 20GB de memoria RAM y WSL 2 Ubuntu.
- Un laptop con procesador AMD Ryzen 5 5500U, disco SSD, 8GB de memoria RAM y WSL 2 Ubuntu.

# Representación del juego

En el archivo othello_cut.h se encuentra una implementación casi completa de la representación del juego Othello (también conocido como Reversi) en un tablero 6x6 con fichas blancas y negras. En el código se definen varios arreglos: rows, cols, dia1, dia2, que almacenan las posiciones de las fichas en filas, columnas y diagonales, respectivamente. Estos arreglos se utilizan para verificar los movimientos válidos y para voltear las fichas cuando se realiza un movimiento. Con esta definición dada de los cuatro arreglos dichos anteriormente, se puede inferir que el diseño del tablero es de la siguiente manera:

```
/* El tablero del juego es:
4   5   6   7   8   9
10  11  12  13  14  15
16  17  0   1   18  19
20  21  2   3   22  23
24  25  26  27  28  29
30  31  32  33  34  35
*/
```

Cada número de la imagen de arriba representa una posición del tablero.

Seguidamente, tenemos a la clase state_t que representa el estado del juego, el cual provee una representación del tablero con 72 bits en total, donde:

1. t_: Es una variable de tipo unsigned char para almacenar 8 bits, donde los 4 bits menos significativos representan las posiciones centrales del tablero (0, 1, 2 y 3). Si el bit es 0, significa que esa posición central está ocupada por una ficha blanca, en caso de que esté ocupada por una ficha negra, el bit es 1. Para crear una instancia de un estado, siempre se inicializa con t_ = 6, el cual 6 en número binario con 8 bits es 00000110. Esto se debe a que el juego siempre comienza con 4 fichas situadas en las cuatro casillas centrales del tablero, de forma que cada pareja de fichas iguales forman una diagonal entre sí.

2. free_: Para los siguientes 32 bits, cada una de estas bits representa si una posición está libre o no. Si está libre, el bit es 0, en caso contrario, se prende el bit.

3. pos_: Para los últimos 32 bits, cada una de ellas representa cada posición en el tablero, sin contar con las 4 posiciones centrales (0, 1, 2 y 3). Si el bit es 0, significa que esa posición está ocupada por una ficha blanca o también que por defecto significa que está

libre esa posición. Y si el bit es 1, significa que esa posición está ocupada por una ficha negra.

Además, la clase state_t proporciona varios métodos para consultar y manipular un estado del juego, incluyendo el hashing de un estado (para almacenar estados visitados en una tabla de transposición), la verificación de si una posición está ocupada por una ficha negra o blanca o está libre, verificar si un movimiento es válido o no, verificar si el estado es terminal, generar un movimiento válido aleatorio para un color de una ficha, y para imprimir el tablero.

La función outflank es particularmente importante, ya que verifica si un movimiento es válido comprobando si hay fichas opuestas rebasadas (es decir, rodeadas por las fichas del jugador) en las filas, columnas o diagonales. Si hay fichas opuestas rebasadas, el movimiento se considera válido y las fichas rebasadas serán volteadas cuando se realice el movimiento.

Para completar esta representación, se implementó la verificación de la diagonal principal y secundaria en las funciones outflank y move. Además, se implementó una función adicional get_valid_moves(bool color) en state_t para retornar un vector de estados para movimientos válidos para una ficha de color dado, esta función nos servirá para utilizar los 4 algoritmos para árboles de juego que se pide en este proyecto.

## Algoritmo para árboles de juego

Se implementó los siguientes algoritmos para árboles de juego:

1. Negamax sin poda alpha-beta
2. Negamax con poda alpha-beta
3. Scout
4. Negascout = negamax con poda alpha-beta + scout

Se realizó una evaluación a lo largo de la variación principal del juego Othello. La variación principal es una secuencia de mejores movimientos para ambos jugadores en cada profundidad que conduce a un estado terminal con un valor de juego conocido, que para este caso, es -4.

El enfoque utilizado consiste en hacer backtracking, comenzar desde el tablero terminal en la variación principal y ascender gradualmente hacia la raíz del árbol. En cada nodo de la variación principal, se ejecuta el algoritmo de solución correspondiente a partir de ese nodo, ya

sea hasta que termine la búsqueda completa o hasta que se alcance un límite de tiempo establecido (se estableció 2 horas para el límite de tiempo para cada algoritmo).

Por lo tanto, el mejor algoritmo será aquel que pueda ascender más lejos (más cerca de la raíz del árbol) en la variación principal, manteniendo la evaluación correcta de los nodos.

En el siguiente enlace al spreadsheet se encuentran los resultados al emplear estos 4 algoritmos para el juego Othello 6x6: https://docs.google.com/spreadsheets/d/1MZN4lNsOqBHmlPDvr0VJg6FMYBbMlQlHTQTRLmtsdXU/edit?usp=sharing

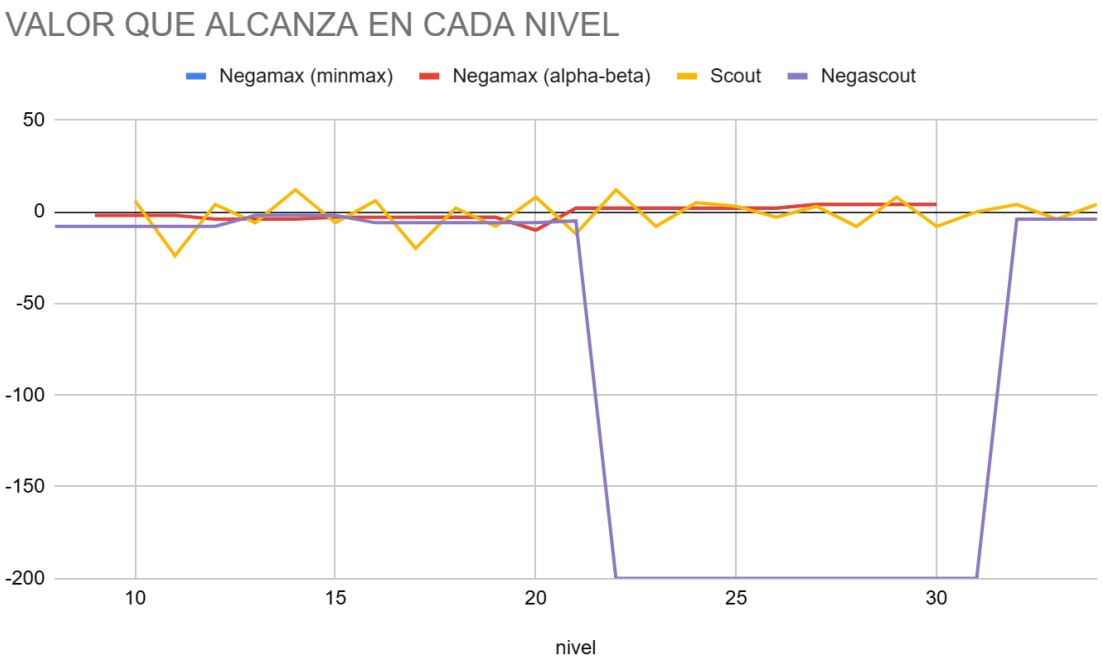
## Análisis de los resultados

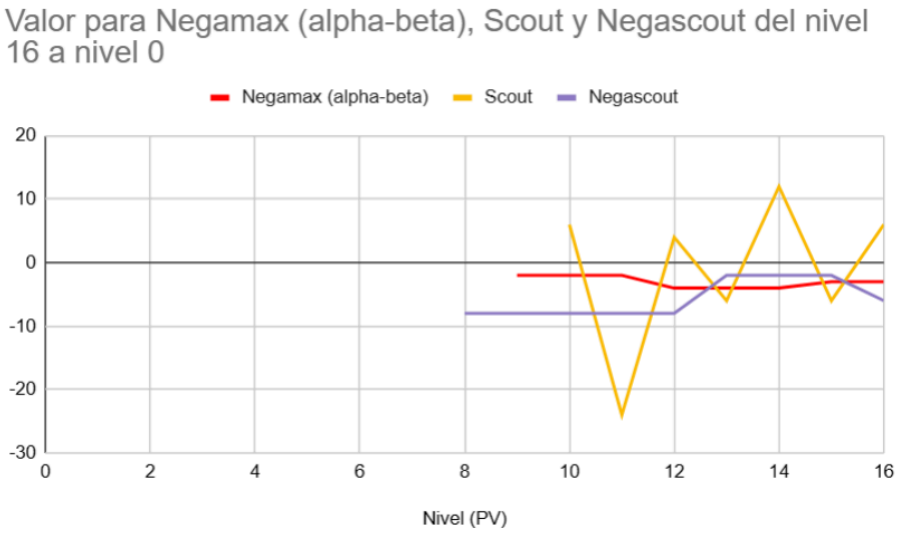
Tenemos que cada algoritmo empieza desde el PV o nivel 34 y los niveles alcanzados (más cerca de la raíz del árbol) en cada algoritmo son:

1. Negamax sin poda alpha-beta: PV = 17 con un valor -3.
2. Negamax con poda alpha-beta: PV = 9 con un valor -2.
3. Scout: PV = 10 con un valor +6.
4. Negascout = negamax con poda alpha-beta + scout: PV = 8 con un valor -8.

En este sentido, ninguno de estos algoritmos logró alcanzar el nivel 0 o la raíz dentro del límite de tiempo establecido. Sin embargo, podemos inferir que negascout tiene mejor desempeño en comparación con otros algoritmos, ya que es el que alcanzó el menor nivel posible y por lo tanto es el que asciende más lejos. Mientras que negamax sin poda alpha-beta es el algoritmo más exhaustivo de los evaluados.

1. **Gráfico de valor para cada algoritmo en  cada nivel o PV**




Se puede observar que efectivamente la línea que está más cerca del 0 en el eje “Y” es negascout (línea morada), seguido de negamax con poda alpha-beta (línea roja), y a continuación scout (línea amarilla).

También, cabe resaltar que en el nivel 31 que le toca el turno a las fichas negras (el jugador MIN), el valor es -infinito para los algoritmos negamax sin y con poda alpha-beta

Por consiguiente, en este nivel, el juego favorece al jugador MIN, igualmente para los otros dos algoritmos, el cual tenemos que disminuye su valor al llegar del nivel 32 al nivel 31.

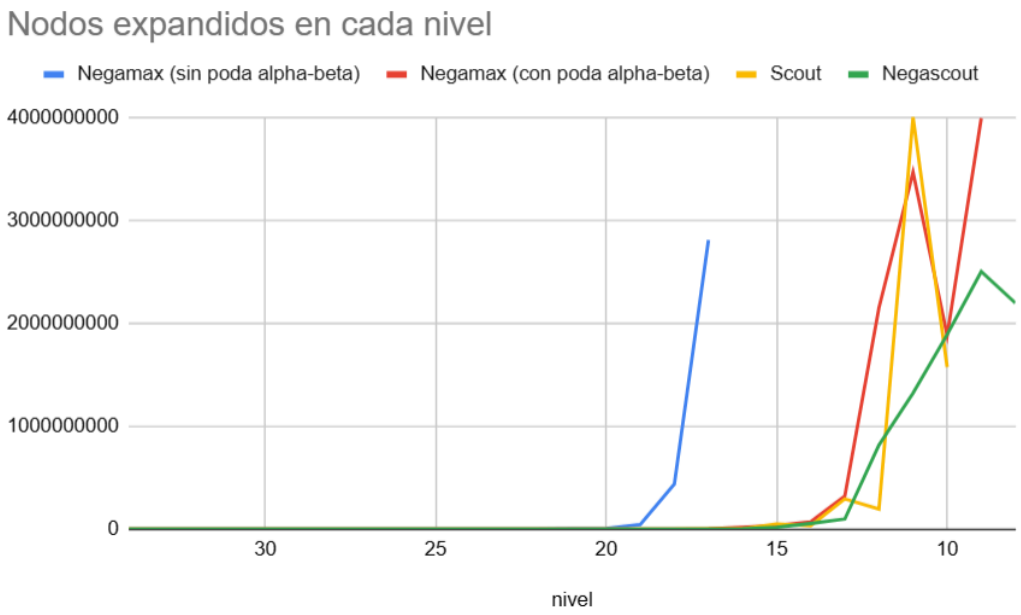
| nivel | Color | Negamax (minmax) | Negamax (alpha-beta) | Scout | Negascout |
|---|---|---|---|---|---|
| 34 | White | -4 | -4 | 4 | -4 |
| 33 | Black | -4 | -4 | -4 | -4 |
| 32 | White | -4 | -4 | 4 | -4 |
| 31 | Black | -INF | -INF | 0 | -200 |
| 30 | White | 4 | 4 | -8 | -200 |
| 29 | Black | 4 | 4 | 8 | -200 |
| 28 | White | 4 | 4 | -8 | -200 |
| 27 | Black | 4 | 4 | 3 | -200 |
| 26 | White | 2 | 2 | -3 | -200 |
| 25 | Black | 2 | 2 | 3 | -200 |
| 24 | White | 2 | 2 | 5 | -200 |
| 23 | Black | 2 | 2 | -8 | -200 |
| 22 | White | 2 | 2 | 12 | -200 |
| 21 | Black | 2 | 2 | -12 | -5 |
| 20 | White | -10 | -10 | 8 | -6 |
| 19 | Black | -3 | -3 | -8 | -6 |
| 18 | White | -3 | -3 | 2 | -6 |
| 17 | Black | -3 | -3 | -20 | -6 |
| 16 | White | NULL | -3 | 6 | -6 |
| 15 | Black | NULL | -3 | -6 | -2 |

También se observa que en el gráfico no aparece la línea azul (los valores de negamax sin poda alpha-beta en cada nivel) debido a que ellos coincide exactamente con los valores de negamax con poda alpha-beta en cada nivel, por lo tanto, la línea roja está superponiendo completamente la línea azul. Esto no es sorprendente, ya que Negamax con poda alfa-beta es una optimización del algoritmo Negamax básico, y si no se produce la poda, ambos algoritmos deberían obtener los mismos resultados en valor.

Y del gráfico, se observa que la línea amarilla (scout) parece más volátil, con picos más altos y bajos, mientras que las líneas de negamax con y sin poda alpha-beta son más estables.

Además, para la línea morada (negascout), se tiene que entre los niveles 22 y 31, el valor es -200, lo cual sugiere una evaluación favorable para el jugador MIN (las fichas negras) en esos niveles.

## 2. Gráfico de nodos expandidos en cada nível o PV

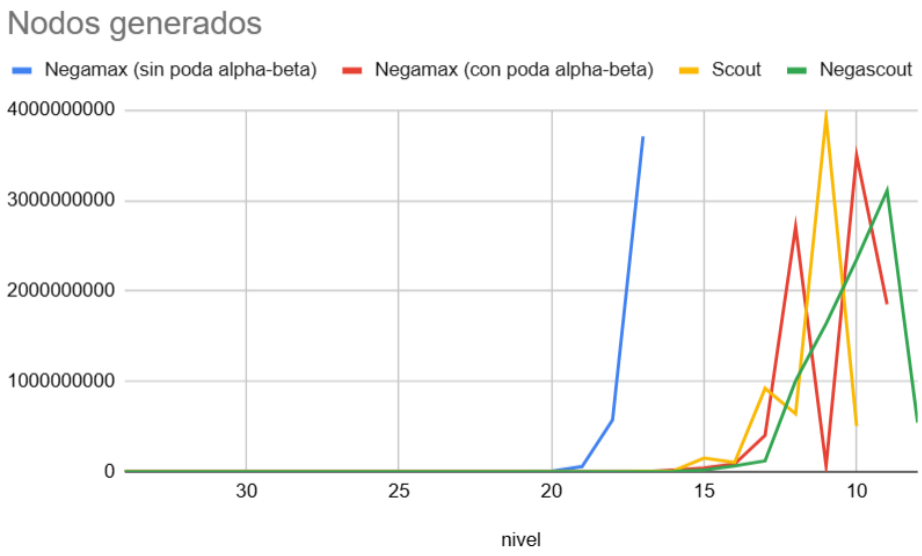



Se observa que la línea azul tiene un crecimiento exponencial en los niveles más bajos del árbol del juego, el cual sugiere que el negamax sin poda expande más nodos en cada nivel en comparación con los otros algoritmos. Y no se observa la continuación de la línea azul en los niveles 16 hacia atrás debido a que este algoritmo terminó en el nivel 17 sin completar la exploración, como se dijo anteriormente.

Para las líneas roja y amarilla (negamax con poda y scout), se observa una disminución de nodos explorados a partir del nivel 12 hasta llegar al nivel 10, donde la línea amarilla comienza a expandir más nodos que los otros (3.996.657.611 nodos expandidos) y se termina por el tiempo de límite establecido, luego la línea roja de negamax con poda vuelve a crecer hasta llegar al punto más alto (3.987.694.059 nodos expandidos).

Para negascout (línea verde), se observa que es la que explora menor cantidad de nodos a medida que alcance al nivel 0 (donde está la raíz del árbol). Y se muestra una disminución de nodos expandidos en el último nivel que alcanza este algoritmo.

## 3. Gráfico de nodos generados en cada nivel o PV



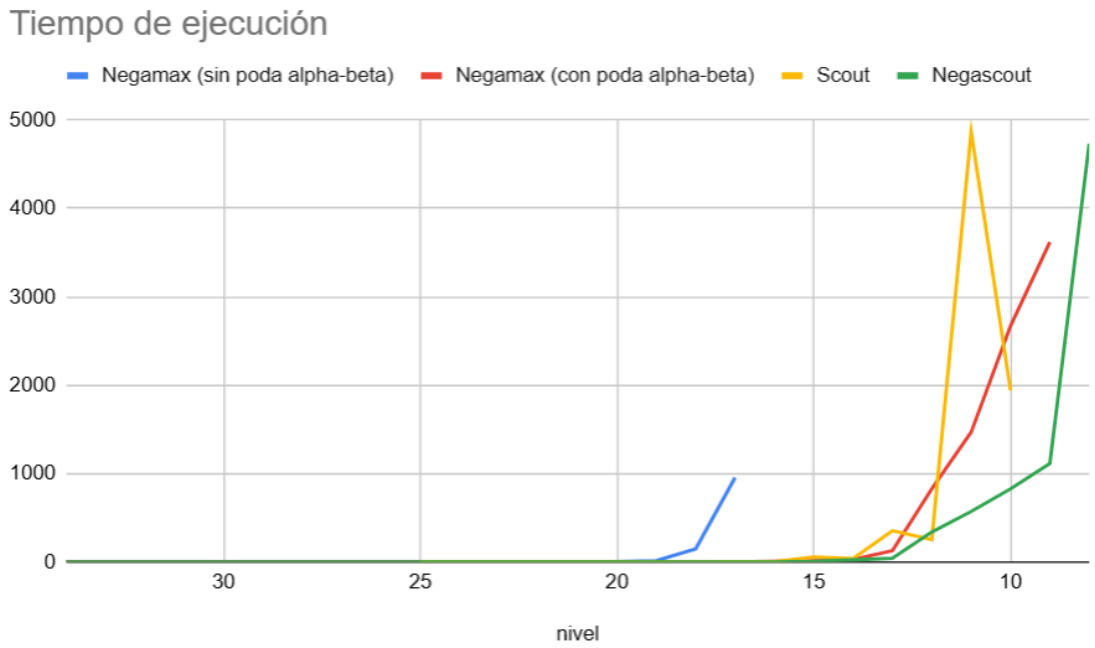

Para negamax sin poda (línea azul) se observa un crecimiento exponencial en el número de nodos generados a medida que se asciende en el árbol y en los niveles más bajos (más lejos de la raíz del árbol), lo cual es esperado debido a la naturaleza exhaustiva de este algoritmo sin podas. En su último nivel que alcanza (nivel 17) se genera una cantidad de 3.708.262.376 nodos.

Para negamax con poda (línea roja) se muestra una variación a lo largo de cada nivel, el cual, en el nivel 12 llega a generar 2.708.688.679 nodos, luego a partir del nivel 11 asciende con una reducción significativa en el número de nodos generados (67.948.849 nodos), pero en el nivel 10 esta línea vuelve a crecer hasta llegar a 3.497.406.762 nodos generados y vuelve a bajar para su último nivel alcanzado con 1.853.737.262 nodos generados.

Para scout (línea amarilla) se puede observar que en los niveles más bajos, este algoritmo genera menos cantidad de nodos en comparación con negamax con y sin poda, pero aumenta drásticamente desde el nivel 13 hacia atrás (hacia el nivel 0). La mayor cantidad de nodos que genera es 3.919.889.743.

Y para negascout (línea verde), éste genera menos nodos en cada nivel en comparación con los anteriores 3 algoritmos, exceptuando el nivel 11, el cual en este nivel el que genera menos nodos es negamax con poda. La máxima cantidad de nodos generados por el algoritmo negascout es 3.109.932.267 en el nivel 9.

## 4. Gráfico de tiempo de ejecución en cada nivel o PV

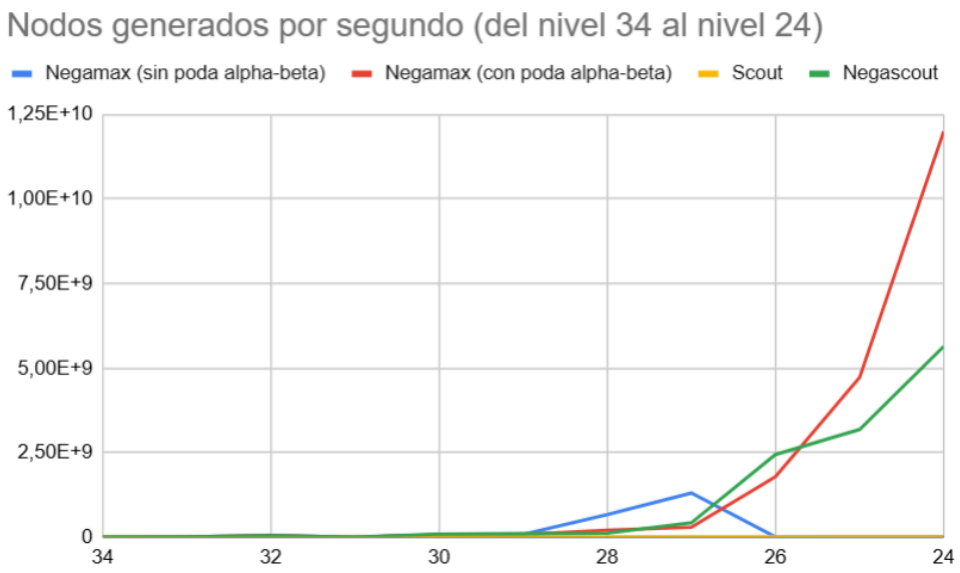


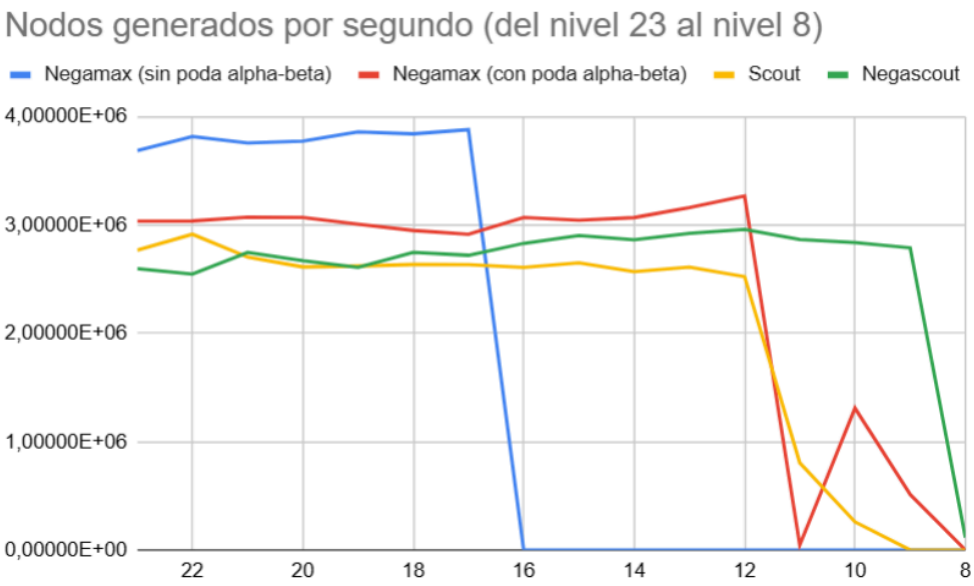

Este gráfico muestra que en los niveles más profundos negamax sin poda tarda más en ejecutarse en comparación con otros algoritmos.

Para el algoritmo scout (línea amarilla), se observa que tarda más tiempo en comparación con los algoritmos negamax con poda y negascout, particularmente, en el nivel 11 se tiene que scout tardó 4.866,47 segundos que es, aproximadamente, 81 minutos, en terminar su exploración para este nivel.

Para negascout (línea verde) se puede observar que siempre está debajo de las otras líneas (excepto en el nivel 12, donde el algoritmo scout es el que tarda menos). Esto sugiere que negascout es el algoritmo más eficiente en términos de tiempo de ejecución para casi la mayoría de los niveles explorados en este caso.

## 5. Gráfico de nodos generados por segundo en cada nivel o PV




Tenemos que en los niveles más profundos del árbol (del nivel 34 al nivel 24) se genera una cantidad significativa de nodos por segundo para los algoritmos de árboles de juego, exceptuando el algoritmo scout.

Además, tenemos que del nivel 34 al nivel 27, el que genera mayor cantidad de nodos por segundo es negamax sin poda (línea azul), y del nivel 27 al nivel 26, sería negascout (línea verde), y luego la línea roja que representa negamax con poda sobrepasa la línea verde a partir del nivel 26 al nivel 24.

Luego, en el nivel 23, negamax sin poda vuelve a ser el que genera mayor cantidad de nodos por segundo hasta terminarse por límite de tiempo excedido. Posteriormente, la línea roja (negamax con poda) se encuentra encima de la amarilla y de la verde hasta llegar al nivel 12.

Finalmente, del nivel 12 al nivel 8, la línea verde de negascout se coloca encima de las otras líneas. Cabe resaltar también que la línea amarilla permanece siempre debajo de las otras tres líneas, lo cual indica que el algoritmo scout es el que genera menor cantidad de nodos por segundo en comparación con los anteriores 3 algoritmos mencionados.

# Conclusión

En resumen, para los hallazgos de los resultados obtenidos sobre los algoritmos de búsqueda bidireccional aplicados al juego Othello, sugieren que la poda es una buena estrategia para mejorar la eficiencia de los algoritmos de búsqueda ya mencionados y disminuir la cantidad de nodos generados. Además, a medida que aumenta la profundidad, el número de estados generados de los cuatro algoritmos crece exponencialmente, lo que puede llevar a un crecimiento no viable en términos de tiempo y recursos computacionales.

Cabe destacar que los algoritmos basados en Scout tienden a ser más eficientes que los algoritmos basados en Negamax según los resultados. De la misma manera, los algoritmos con algún tipo de poda, tienden a ser más eficientes que los algoritmos sin poda. Esto se evidencia al comparar Negamax con poda y Negascout, con Negamax sin poda y Scout; los dos primeros tiene un tiempo de ejecución por nivel más bajo en la mayoría de los casos, ya que, al tener poda, los mismos evitan generar nodo ya visitados.

Sin embargo, el algoritmo que dió mejores resultados en cuanto al valor fue el algoritmo Scout, ya que fue el que daba resultados más coherentes en relación a el valor buscado en cada nivel. Esto debido a que las piezas blancas buscaban valores mayores de -4 y las piezas negras menores al mismo. Así mismo, Scout generaba menos nodos por segundo en comparación con los otros 3 algoritmos.

En última instancia, esta investigación subraya la importancia de considerar las características específicas del problema y elegir el enfoque algorítmico más adecuado dentro del marco de la Inteligencia Artificial. En el caso del Othello, podemos ver que la poda nos ayuda a buscar un resultado esperado en la mayoría de los casos, ya que evita estados ya visitados, quitando así posibles jugadas. Además, usar Negamax sin poda es poco eficiente ya que es básicamente usar fuerza bruta, lo cual hace que tarde más tiempo que usar los otros tres algoritmos. Por lo tanto, particularmente para el Othello, el mejor algoritmo de búsqueda bidireccional sería el algoritmo Negascout, pues fue el que ascendió más en el árbol de juego, y en la mayoría de los casos exploraba y generaba menos nodos en cada nivel que ascendía, lo cual, tenía tiempos menores en comparación con los otros algoritmos. Cabe destacar que los demás algoritmos pueden ser útiles en otros casos, como Negamax en el Tic Tac Toe por ejemplo.