Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-5437 - Inteligencia Artificial I
Abril - Julio 2021

Proyecto 2: Árboles de Juego

Jesús De Aguiar 15-10360
Neil Villamizar 15-11523
Jesús Wahrman 15-11540

Problema

El objetivo del proyecto es aprender sobre el modelo de árboles de juego. Para esto, se estudiará una versión reducida del juego de Otello y distintos algoritmos para recorrer los árboles de juego aplicados a este juego:

- Negamax sin poda alpha-beta
- Negamax con poda alpha-beta
- Scout
- Negascout

Descripción de la Implementación

La implementación del proyecto se puede encontrar en el repositorio de GitHub jesuswr/proyecto-2-ci5437.

Representación de Othello

En primer lugar, se completó la implementación de la representación del juego Othello. Para ello, se completaron las dos funciones `outflank` y `move`. La primera revisa si un movimiento es válido y la segunda aplica el movimiento.

Para `outflank`, se necesitaba completar la implementación del chequeo de si existía un movimiento válido diagonal. Para completarla, se busca la posición en la que se desea poner la ficha, y se recorre cada diagonal en ambas direcciones mientras que no es encuentre una casilla vacía y mientras la casilla vista sea parte del tablero, esperando encontrar al menos una ficha del rival seguida por una ficha del jugador actual. Si esto ocurre, el movimiento es válido.

Para `move` era necesaria la implementación de las diagonales, por lo que la lógica del cambio implementado es parecida a la de `outflank` pero aquí se aplica el movimiento; es decir, se coloca la ficha y se reemplazan las del oponente que estén afectadas por esta.

Algoritmos de recorrido en árboles de juego

En la implementación de cada uno de los algoritmos mencionados anteriormente en la descripción del problema, se eliminaron los parámetros `depth` y `use_tt`, ya que ninguno fue utilizado en las implementaciones de estos: el primero porque no se quería detener la búsqueda a ninguna profundidad, y el segundo porque las tablas de transposición no fueron implementadas.

La implementación realizada es la aprendida en clases para cada uno de los algoritmos. La única adición relevante a estos en sus implementaciones es que, para este problema, se considera el caso de que un jugador no pueda colocar ninguna de sus fichas y deba pasar. Para ello, al momento de ejecutar los algoritmos, se debe revisar que haya algún movimiento válido, y en caso de no haberlo, el turno pasa al otro jugador usando el mismo estado.

Es importante notar que en la representación del juego obtenida en `othello_cut`, las fichas negras son representadas con el valor booleano *true*, y las fichas blancas con el valor *false*; mientras que en nuestra implementación de los algoritmos, el turno del jugador con fichas negras es representado con el entero 1 y el turno del jugador de fichas blancas, con -1, Por esta razon, se utilizó `color == 1` para pasar de la representación de los algoritmos a la del juego.

Experimentación y Resultados

Para poner a prueba los algoritmos, se ejecutó cada uno de estos de la manera descrita en el enunciado del proyecto: Evaluando el algoritmo en una instancia perteneciente a la variación principal del problema que se encuentra a una distancia cada vez más lejana de la meta del problema en cada iteración. El tiempo límite establecido por el grupo para la ejecución de cada uno de los algoritmos es de dos horas, donde se permitió que el algoritmo se ejecutara en la mayor cantidad de nodos iniciales posibles pertenecientes a la variación principal hasta que este tiempo se agotara.

Los resultados se exponen a continuación, en las distintas tablas y gráficas:

Algoritmo	Profundidad del nodo inicial
Negamax	17
Negamax con poda alpha-beta	11
Scout	10
Negascout	9

Table 1: Versión más compleja del problema resuelta por cada algoritmo en el tiempo establecido

Desde este momento podemos observar como el algoritmo de Negamax es bastante menos eficiente en comparación con su versión optimizada con poda alpha-beta, y al mismo tiempo este último es superado por poca diferencia por el algoritmo de Scout. El algoritmo de Negascout fue capaz de resolver un problema mayor al combinar estas optimizaciones.

Podemos compara un poco más a profundidad el rendimiento de cada una de estas ejecuciones utilizando la salida de la ejecución de los algoritmos:

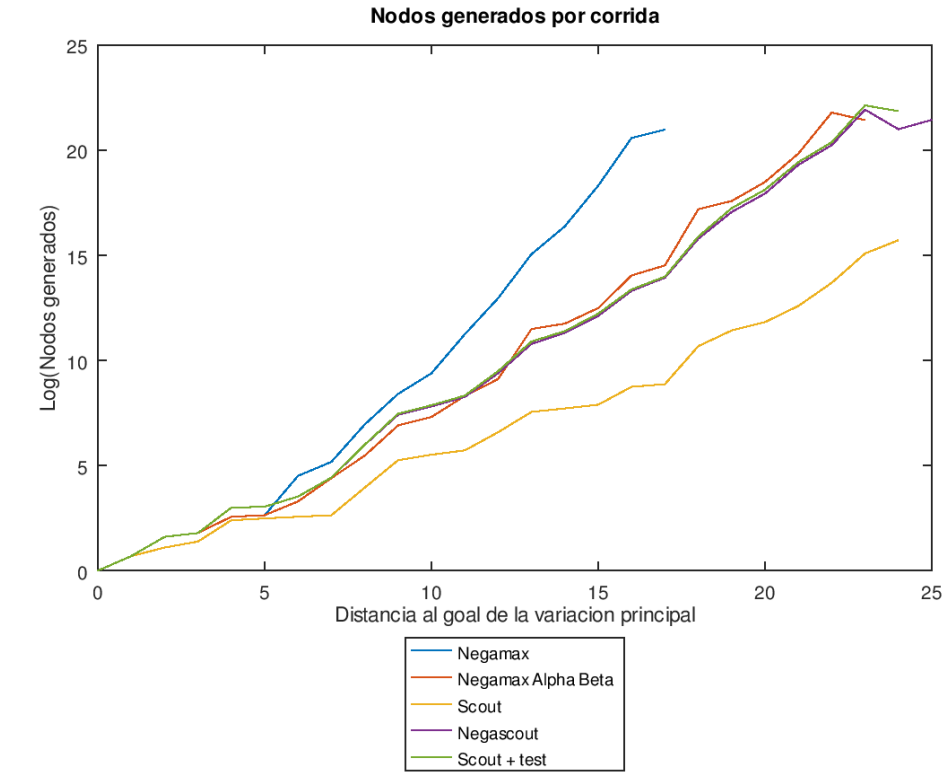
Algoritmo	Nodos Generados	Nodos Expandidos	Tiempo de Ejecución (s)	Nodos Generados por Segundo
Negamax	1305006091	3999381161	1686.39	773847
Negamax con poda alpha-beta	2031924	1549785	0.718762	2826980
Scout	7155	5637	0.429171	16671.7
Scout + Test	1189883	911296	0.415795	2861710
Negascout	1134797	869928	0.431562	2629510

Table 2: Resultados obtenidos por los algoritmos tras recorrer el árbol del juego desde la profundidad 17

Algoritmo	Nodos Generados	Nodos Expandidos	Tiempo de Ejecución (s)	Nodos Generados por Segundo
Negamax con poda alpha-beta	2931981147	2230058150	1080.84	2712680
Scout	888835	693495	306.539	2899.59
Scout + Test	713350259	545805549	260.821	2735020
Negascout	623019805	477000927	272.957	2282480

Table 3: Resultados obtenidos por los algoritmos tras recorrer el árbol del juego desde la profundidad 12

Figure 1: Nodos generados por corrida. La primera gráfica muestra los resultados en notación logarítmica para apreciar de mejor manera la diferencia entre los distintos algoritmos



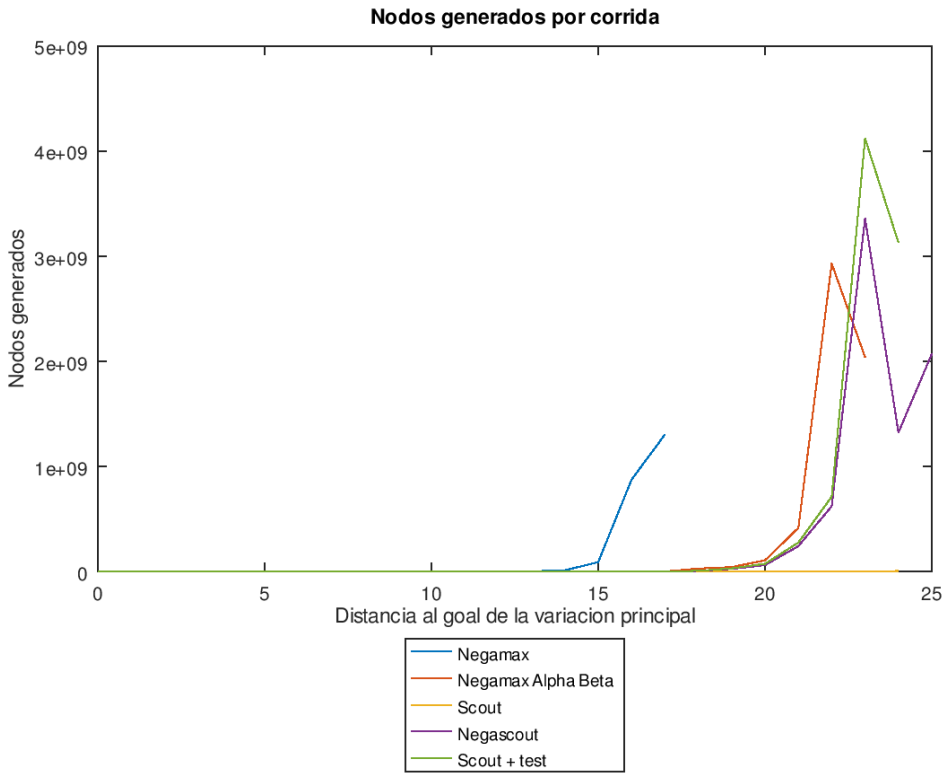
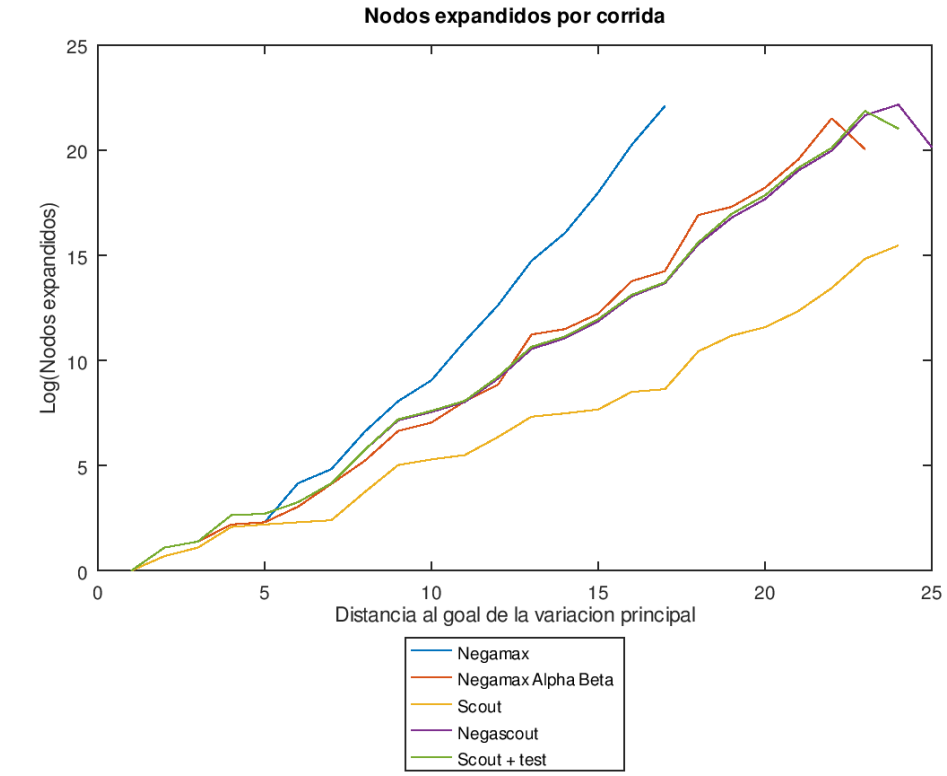


Figure 2: Nodos expandidos por corrida. La primera gráfica muestra los resultados en notación logarítmica para apreciar de mejor manera la diferencia entre los distintos algoritmos



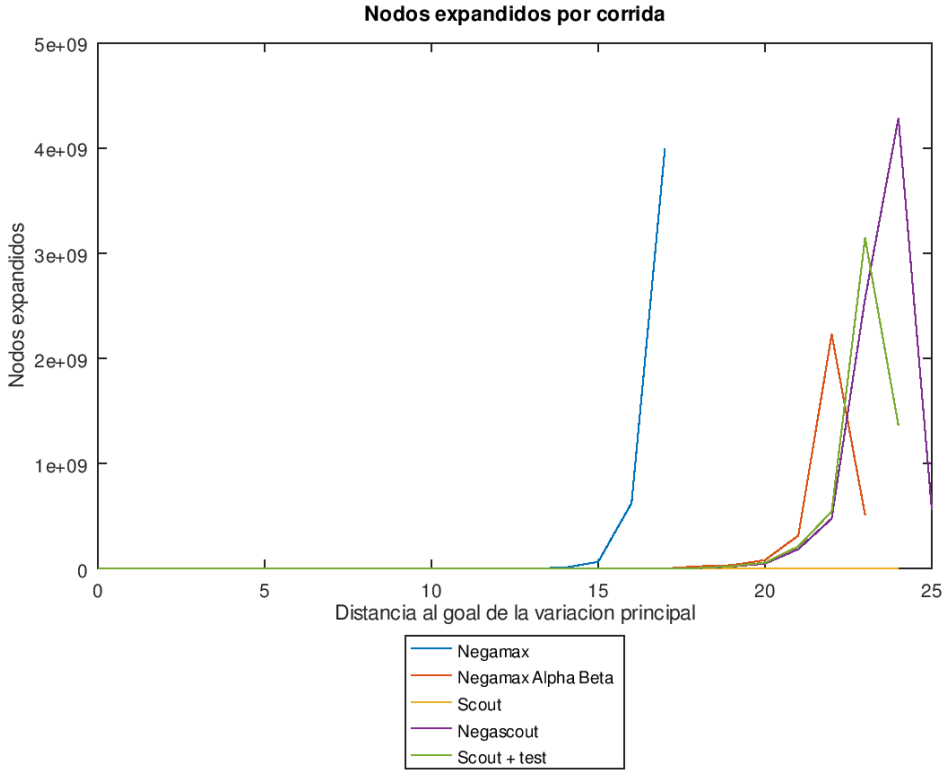
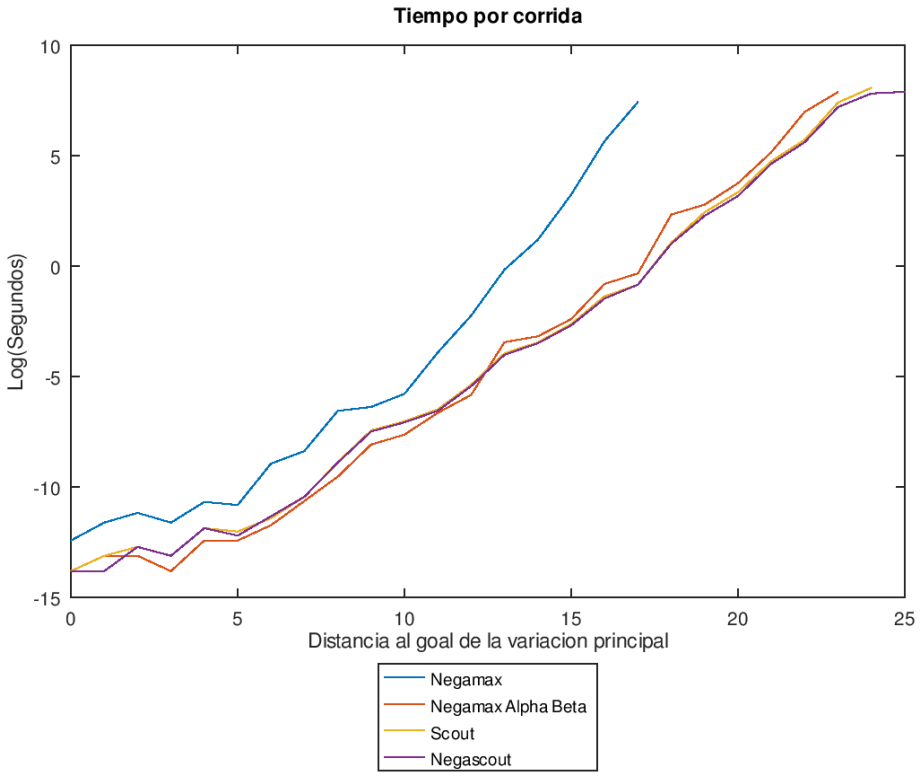
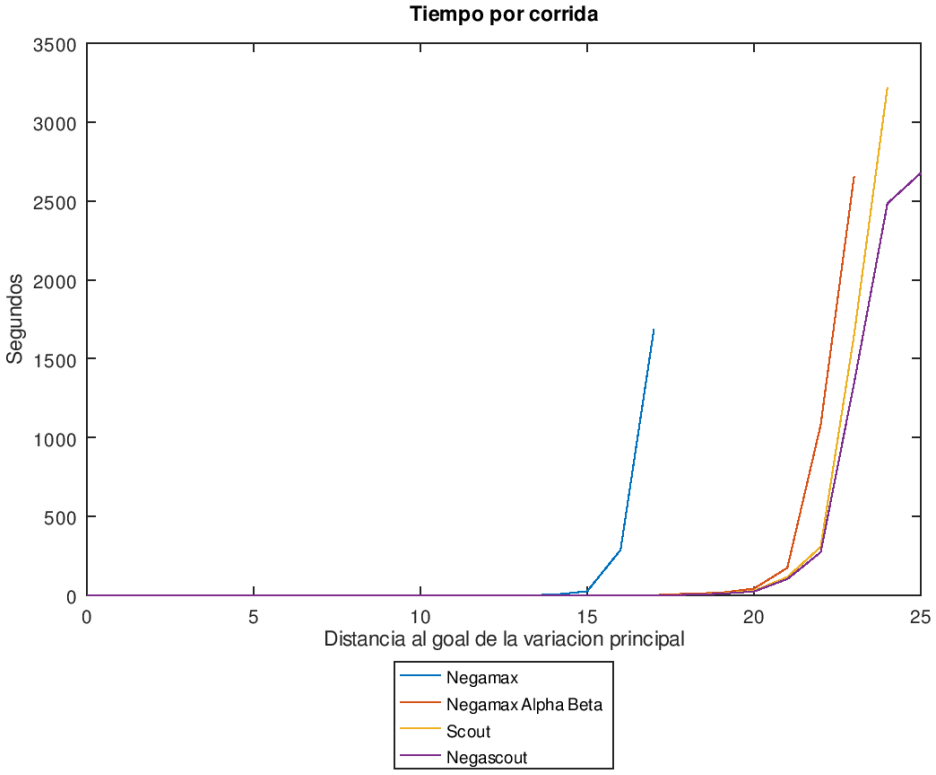


Figure 3: Tiemplo empleado en cada corrida. La primera gráfica muestra los resultados en notación logarítmica para apreciar de mejor manera la diferencia entre los distintos algoritmos





Conclusiones

Como conclusión general podemos darnos cuenta que Negascout fue el algoritmo más eficiente en realizar el recorrido a través del árbol de juego. Entrando un poco en detalle, se puede observar que la poda alpha-beta fue capaz de mejorar de gran manera el rendimiento de negamax, y permitirlo llegar mucho más lejos que su versión básica. Por otro lado, si comparamos scout y negascout con la poda alpha-beta, esta permitió mejorar su rendimiento aunque, en principio, de una manera menos determinante. En la tabla 2 se puede observar como la capacidad de evitar generar nodos es el factor determinante entre los resultados mencionados anteriormente.

Las distintas gráficas apoyan estas observaciones. En la figura 2, se observa cómo rápidamente a lo largo de las corridas la cantidad de nodos de generados por negamax aumenta de gran manera en una ejecución más temprana (de igual manera, en la figura 3 con los nodos expandidos). Estas gráficas también demuestran que el comportamiento de negascout y scout es similar, por lo que la poda alpha-beta no marca una diferencia importante entre estos dos.

Otra observación que se puede realizar a partir de las tablas es la capacidad que tiene la función `test` para recortar estados en la ejecución de scout. El algoritmo principal de Scout genera y expande una cantidad mucho menor de nodos que son descartados por la función `test`.

Podemos observar también que en el último par de corridas hay una discrepancia entre los nodos generados y expandidos. La gráfica nos muestra que el número de nodos expandidos es mayor al número de nodos generados en la ejecución de negascout en la versión mas grande del problema. Esto se debe a que, por la representación de este número en el código y la gran cantidad de estados generados, el valor verdadero excedió el límite de representación (overflow). Aún así, sabemos que el algoritmo fue capaz de resolver el problema hasta ese punto.

Para resolver el problema de manera general se recomienda el uso de negascout puesto que fue el que logró resolver el problema en una instancia menos profunda (más compleja). Scout también podría ser una buena opción a utilizar.