Análisis y Diseño de Algoritmos II

Trabajo Práctico Especial

Grupo 1

ATTILIO, Nicolás (*nicolasattilio@gmail.com*)

PIANCIOLA, Galo (*galopianciola*@*outlook.com*)

Facultad de Ciencias Exactas

Universidad Nacional del Centro de la Provincia de Buenos Aires

Cátedra Análisis y Diseño de Algoritmos II

Fecha de entrega: 30/10/17

El trabajo realizado se basa en la creación de una versión del juego llamado Amazonas. Este consiste en un tablero con 8 *reinas* (fichas) en él, y 2 jugadores que realizarán una jugada válida por turno. Luego de mover una ficha del color correspondiente al jugador, se arroja una flecha que inhabilita una celda del tablero.

El objetivo del trabajo es, por un lado, desarrollar el entorno del juego, de tal forma que se respeten todas sus reglas, movimientos válidos e imposiciones; y por otro lado, programar algoritmos con ingenio y criterio de juego para que uno de los jugadores sea la computadora.

Las tareas que involucró el desarrollo del mismo fueron la creación del entorno general del juego (su visualización por consola, cumplir todas sus reglas, los cambios de turno por jugador, etc.) , el desarrollo de las dos funciones de evaluación heurística, y el algoritmo de búsqueda de la jugada más conveniente entre todas las posibles.

Nuestra primera función de evaluación heurística (valor 1) se basa en analizar todo el tablero, evaluando la situación en el juego de cada reina enemiga mirando su entorno más cercano. El objetivo a atacar en cada jugada será la que este en situación más desfavorable.

La hipótesis que manejamos aquí fue que una eficiente manera de buscar perjudicar al jugador rival es intentar ir eliminando sus reinas de a una, siempre yendo en contra de la que peor esté en cada jugada, y de esta manera, llegar a encerrarlas o imposibilitarle algunos posibles caminos de juego.

A continuación, su implementación:

**for** (**int** i=0;i<=9;i++)

**for** (**int** j=0;j<=9;j++)

**if** (**this**->tablero[i][j]=='A'){

**if** (**this**->verificaEntorno(i,j,'A',ady)==true)

**if** (debil<ady)

debil=ady;

**return** debil;

La segunda heurística (valor 0) busca atacar siempre a la primera reina enemiga que se encuentre en el tablero, empezando su recorrido desde la posición (0,0) del tablero. Esta evaluación la hará cada vez que llegue su turno, y posiblemente su reina objetivo varíe (o no) tras cada jugada.

La hipótesis en la que nos basamos aquí fue la de hacer un ataque más imperativo e impositivo, sin analizar el entorno o la situación de la reina enemiga. De esta manera diferenciamos claramente una heurística de la otra para así poder evaluar diferentes inteligencias automáticas de juego y ver/decidir cuál resulta ser la más eficiente.

**for** (**int** i=0;i<=9;i++)

**for** (**int** j=0;j<=9;j++)

**if** (**this**->tablero[i][j]=='A')

**return** ady+1;

Aclaramos que en el programa, ambas heurísticas se encuentran en la misma función, con la posibilidad de que el usuario elija.

El algoritmo Negamax con poda Alpha-Beta que implementamos busca evaluar y poner en consideración todas las jugadas posibles del jugador en turno indicado, buscando siempre la más conveniente, “podando” las que no lo son. Nuestra implementación de este algoritmo es la siguiente:

**int** Tablero::negaMax(**int** depth, **int** alpha, **int** beta, char turno,**int** tipoHeuristica,estado & movimiento,**int** & tamanio) {

**int** valorJugador;

**if** ((!**this**->sigueJugando(turno)) **or** (depth==0)) {

**if** (turno=='A')

valorJugador=-1;

**else**

valorJugador=1;

**return** (this->heuristicas(tipoHeuristica)) \* valorJugador;

} **else** {

**int** maxx=INT\_MIN;

**list** <estado> movimientos;

**this**->generarMovimientosValidos(turno,movimientos);

**list** <estado>:: iterator it=movimientos.begin();

tamanio=movimientos.size();

estado contrario;

(sigue…)

(…)

**while**(it!=movimientos.end()) {

**this** ->movimientoValido(it->origenA,it->origenB,it->destinoA,it->destinoB,it->flechaA,it->flechaB,turno);

**if** (turno=='A')

turno='R';

**else**

turno='A';

**int** valor =-(negaMax(depth-1,-beta,-alpha,turno,tipoHeuristica,contrario,tamanio));

**if** (turno=='A')

turno='R';

**else**

turno='A';

**this**->deshacerUltimoMovimiento(it->origenA,it->origenB,it->destinoA,it->destinoB,it->flechaA,it->flechaB,turno);

**if** (valor > maxx)

maxx = valor;

**if** (valor > alpha) {

alpha = valor;

movimiento=\*it;

}

**if** (alpha >= beta)

return alpha;

it++;

}

**return** alpha;

}

}

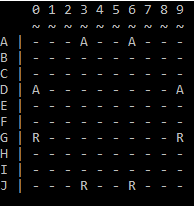
}

Con respecto a las decisiones en horas de implementación, las que consideramos revelantes para ser destacadas son, por ejemplo, la de utilizar registros (*struct*) para guardar cada acción de movimiento en el tablero, guardando origen y destino de la reina, y lo mismo de la flecha a lanzar. Al principio no habíamos hecho esto, por lo cual a la hora de armar el algoritmo Negamax, tuvimos problemas para construir la lista de posibles movimientos a efectuar, y tuvimos que retroceder en la implementación.

Otra de las decisiones, fue la que necesitamos tomar a la hora de buscarle un fin a la ejecución del juego: analizamos el área más cercana (cuadrado de 3x3) de cada reina en el tablero, teniendo en cuenta los límites del mismo. Si todas las reinas de un equipo están totalmente cerradas, este jugador pierde la partida.

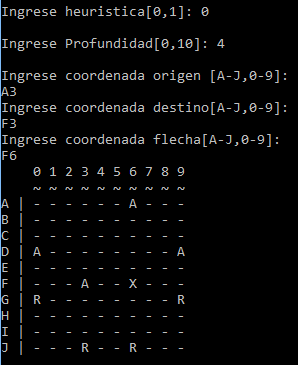
Tablero y ejemplos de movimiento:

En la imagen 1, se puede observar el tablero inicial del juego implementado:



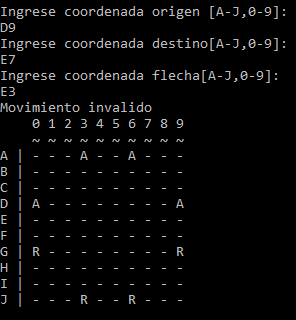
*(imagen 1)*

Luego, como se ve en la *imagen 2*, se elige la heurística, la profundidad de la exploración a utilizar en la función Negamax, y las coordenadas de movimiento de las reinas y la flecha, con el formato .



*(imagen 2)*

En la *imagen 3*, veremos un ejemplo de movimiento de ficha (reina en este caso) inválido, por no ser un movimiento permitido (horizontal, vertical u oblicuo).

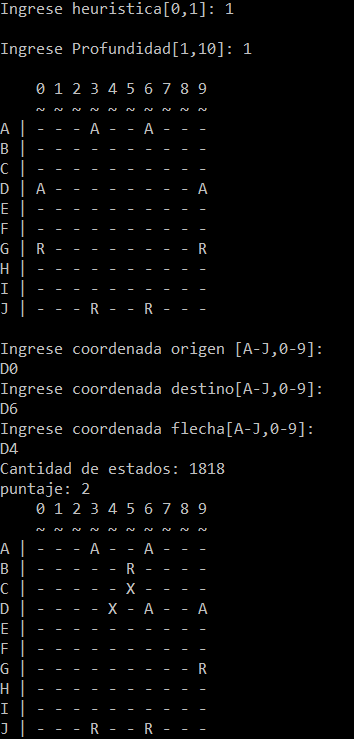


*(imagen 3)*

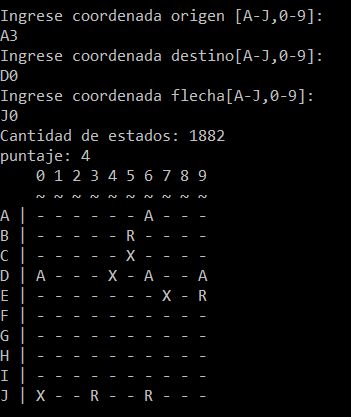
Análisis y presentación de resultados:

En las imágenes 4-6, podemos apreciar cómo actúa el algoritmo en torno a los movimientos del jugador, y siguiendo la estrategia planteada en la heurística elegida.

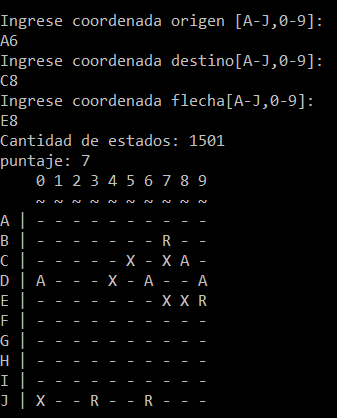
Aclaración: Las reinas del jugador son A’s, y las del contrario son R’s.



*(imagen 4)*



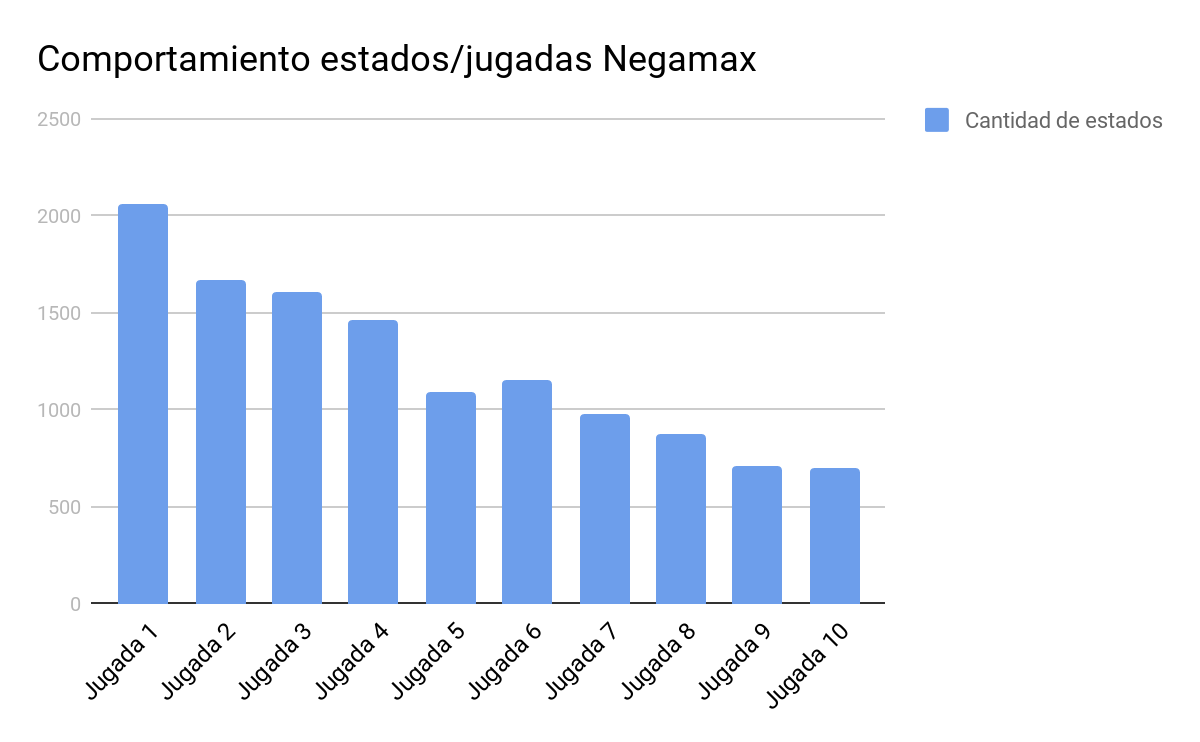
*(imagen 5)*



*(imagen 6)*

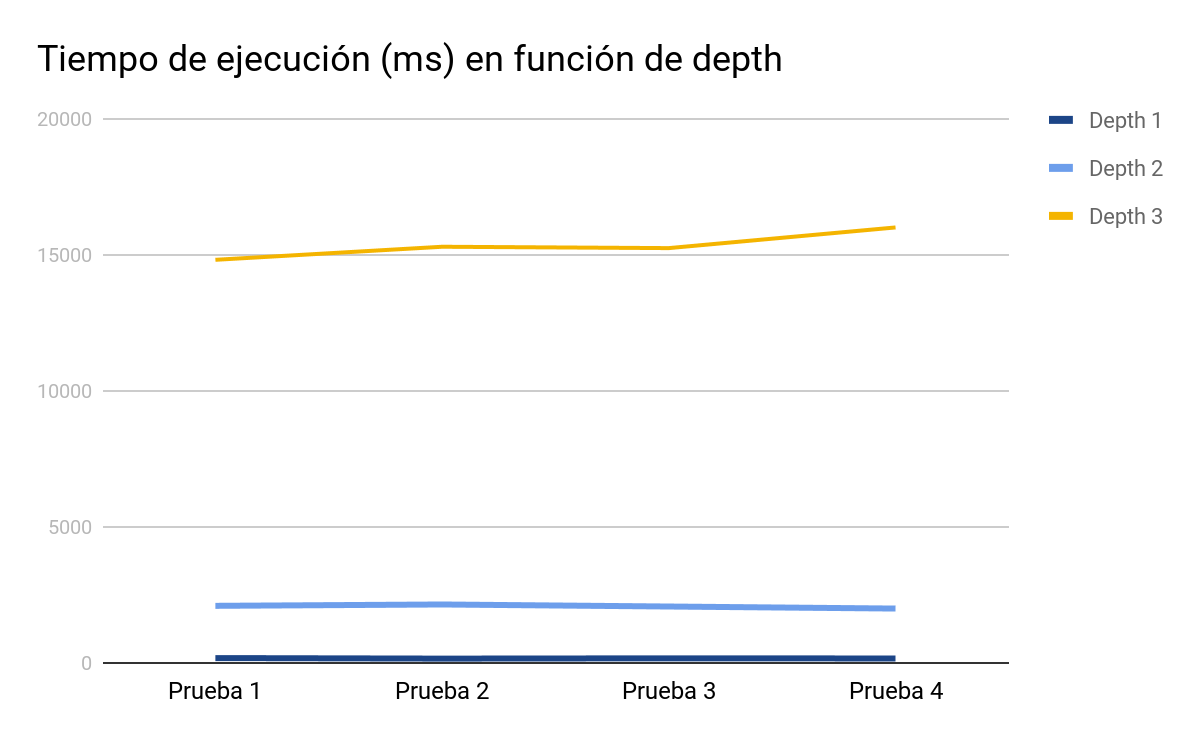
En cuanto la comparación e interpretación de los resultados presentados, la calidad de solución que encontramos fue positiva, ya que los algoritmos de juego automático programados no sólo cumplen con las reglas impuestas, sino que actúan de manera inteligente, y buscan incomodar al jugador usuario.

Haciendo diferentes pruebas sobre el programa final, pudimos realizar un análisis estadístico del comportamiento del algoritmo que arrojó los siguientes resultados:



En referencia al espacio de búsqueda total, pudimos apreciar que a medida que avanza la partida, es decir, cuanto más casillas ocupadas tiene el tablero, las posibilidades planteadas por Negamax se reducen a partir de la poda. Por lo tanto, la lista que contiene todos los movimientos posibles, tendrá cada vez menos elementos. Es necesario aclarar que esto puede no cumplirse todas las veces rigurosamente, sino que se pueden presentar pequeñas variaciones (como se apreció en el gráfico de análisis presentado anteriormente).

Con respecto al tiempo de ejecución en función a la profundidad ingresada por el usuario para la partida, vemos que el tiempo crece junto con el parámetro *depth*. A continuación, los datos obtenidos para demostrar la hipótesis planteada:



Otra deducción que surge a partir de estos análisis, es que el tiempo de ejecución no se ve afectado por la cantidad de estados generados por Negamax, ya que en la primer jugada, se tienen entre unos 2000-2200 posibles estados aproximadamente. Este rango va variando dependiendo de la poda Alpha-Beta, generando disminución en el número de estados obtenidos.

Conclusiones:

La conclusión acerca del trabajo propuesto, fue descubrir la posibilidad de presentar problemas y resolverlos desde otro punto de análisis. El hecho de buscar idear y programar la forma de actuar en base a estados que toma, en este caso, el tablero del juego Amazonas, es un buen ejemplo de lo que se plantea: darle a la computadora una forma de pensar y actuar en base a hechos concretos.

En nuestro trabajo, comenzamos enfocándonos en la creación del tablero con sus fichas, y los debidos movimientos válidos e inválidos del juego. Luego, cambiando el foco a las heurísticas, ideamos diferentes formas de analizar la situación del juego y actuar en base a ello. Por último, intentamos (y pudimos) resolver el algoritmo Negamax, en el cual tuvimos una complejidad elevada a la hora de programarlo.

Para cerrar el tema, coincidimos en que las sensaciones que nos deja el balance del trabajo en su totalidad son buenas, ya que aprendimos nuevos enfoques de la programación y resolución de problemas, aunque con dudas y mala toma de decisiones para la implementación de ciertos algoritmos puntuales.