Inteligencia Artificial
Práctica 4

Martín Selgas, Blanca (blanca.martins@estudiante.uam.es)
Villar Gómez, Fernando (fernando.villarg@estudiante.uam.es)
Grupo 2302 - Pareja 3

03/05/2018

1. Introducción

En el presente documento se detalla la realización de las tareas correspondientes a la cuarta práctica de Inteligencia Artificial. Entre ellas se incluyen la implementación en LISP de funciones de evaluación (heurísticas) para generar jugadores automáticos de una variante del juego Mancala y la contestación a una serie de preguntas teóricas.

El desarrollo de la práctica se ha realizado en un entorno GNU/Linux a través de los lenguajes de programación LISP y C (se explican los detalles más adelante en esta memoria).

2. Funciones de evaluación

Durante el desarrollo de las heurísticas han surgido unos cuantos puntos en donde se han tenido que realizar ciertas decisiones de diseño, que describimos en los siguientes apartados:

2.1. Lenguaje de programación

En un principio, el código proporcionado para la realización de la práctica está escrito en LISP, por lo que para la realización de cualquier tipo de pruebas fiables hubiera sido adecuado usar este lenguaje, dado que la única tarea pendiente era la propia implementación de las funciones de evaluación. Sin embargo, la ejecución de cada partida en LISP es demasiado lenta para obtener resultados satisfactorios, por lo que finalmente se decidió replicar todo el funcionamiento de esta versión de Mancala en C, de manera que las posibles pruebas fueran significativamente más veloces.

A lo largo de este segundo apartado se van a detallar los diferentes módulos que se han creado para la realización de todas las tareas necesarias de replicación de Mancala. En un principio, hay que implementar el desarrollo de una partida, para lo cual se utilizarion tres módulos:

- **Módulo Mancala**, implementado en los archivos `mancala.c` y `mancala.h`, donde se incluyen las funciones para realizar movimientos, comprobar el fin de la partida, jugadas especiales como la captura o la repetición de turno, etc.

  La unidad principal de información es una estructura que representa el estado de una partida, y que fundamentalmente contiene un array de 14 números enteros (los 14 hoyos donde puede haber semillas), donde las posiciones 0-6 son los 6 hoyos del jugador uno y su kalaha, y las posiciones 7-13 son los 6 hoyos del jugador dos y su kalaha. Además, en esta estructura se guardan también el jugador que tiene el turno y las estrategias de ambos; estas últimas están formadas por su función de evaluación (heurística) y la profundidad del árbol de decisión que se genera para realizar la elección de movimiento.

  Se incluye también una función para generar el árbol de decisión a partir de un estado de la partida. Estos árboles de decisión se gestionan desde el módulo siguiente, a excepción de la función mencionada ahora mismo.

- **Módulo de árboles**, implementado en los archivos `hextree.c` y `hextree.h`, donde se incluyen las funciones para generar y rellenar los árboles con factor de ramificación 6 correspondientes a los posibles movimientos que se pueden realizar desde cada estado, además de la implementación del algoritmo **negamax**.

  Estos árboles se implementan a traves de una estructura de nodos en las que se incluyen los seis posibles nodos hijos, el valor del nodo (que se irá actualizando con el algoritmo negamax a partir de los valores de las hojas de los árboles, obtenidos a su vez a través de las funciones de evaluación aplicadas sobre los estados a los que se corresponden) y un índice que se corresponde al primer movimiento de la secuencia de movimientos que se han de realizar para llegar al estado del nodo actual.

- **Módulo de heurísticas**, implementado en los archivos `heuristic.c` y `heuristic.h`, en el que se define lo que es una heurística y un generador de heurísticas y se implementan las funciones correspondientes.

Ampliamos la explicación sobre este último módulo en el siguiente apartado.

2.2. Heurísticas

Al ser un proyecto desarrollado en C, se han creado dos tipos de datos asociados a punteros a funciones, el tipo `heuristic` que se corresponde a funciones de evaluación, y el tipo `generateWH` que se explica más adelante.

En cuanto a funciones de evaluación, se han implementado las dos que usa el código original de la práctica (la estrategia del jugador regular, llamada aquí `heuristicIARegular` y la estrategia del jugador bueno, llamada aquí `heuristicIABuena`), y a partir de ahí lo que resta es decidir qué tipo de heurísticas se van a crear para las pruebas, cómo van a ser generadas y qué tipo de pruebas se van a hacer sobre ellas.

2.2.1. Tipo de heurísticas

Dado que la información de la que disponemos en un estado dado es, sencillamente, las semillas contenidas en cada hoyo, la opción más natural es hacer una especie de clasificación de la importancia de la cantidad de semillas de cada hoyo. Para representar esta "importancia", lo más intuitivo es la creación de un sistema que asigne diferentes pesos a la cantidad de semillas de cada hoyo, sumando finalmente todos estos valores para dar lugar al valor final de la heurística. Siendo p_i el peso asociado al hoyo i, y siendo h_i la cantidad de semillas contenidas en el hoyo i, se representa con la fórmula:

$$\sum_{i=0}^{14} p_i \cdot h_i$$

Donde, para conseguir una representación lo más correcta posible de la importancia del estado de cada hoyo, idealmente los pesos deben cumplir $-p < p_i < p$ para todo hoyo i con $p > 0$; de esta forma, un peso negativo representa tener interés en que no haya semillas en un hoyo, un peso cercano a 0 indica indiferencia sobre la cantidad de semillas de un hoyo y un peso positivo representa tener interés en que haya semillas en un hoyo. El valor de este p no es importante; dados p_1 y p_2 reales no negativos diferentes, un mismo estado de la partida de Mancala y dos conjuntos de pesos p_{i_1}, p_{i_2} que cumplen $-p_1 < p_{i_1} < p_1$, $-p_2 < p_{i_2} < p_2$ y $\exists k : k = \frac{p_{i_1}}{p_{i_2}}$ para todo hoyo i:

$$\sum_{i=0}^{14} p_{i_1} \cdot h_i = \sum_{i=0}^{14} k p_{i_2} \cdot h_i = k \sum_{i=0}^{14} p_{i_2} \cdot h_i$$

Es decir, los valores devueltos por las dos funciones de evaluación correspondientes son proporcionales; y por tanto a la hora de tomar una decisión dados varios estados diferentes de una partida de Mancala, si hay que escoger el máximo la elección va a ser la misma para ambas funciones, y lo mismo ocurre si hay que escoger el mínimo. Por tanto, con ambos paradigmas (y al ser los pesos números reales) se van a obtener siempre las mismas heurísticas, que se diferenciarán entre ellas simplemente por el factor de proporcionalidad k.

Para tener una buena claridad y entender de forma muy intuitiva el papel que juegan los pesos en el cálculo de la heurística y su relación con la importancia de cada hoyo, en nuestro caso asignamos siempre el valor $p = 10{,}0$ para los cálculos.

Se podrían seguir otros muchos esquemas para conseguir el mismo objetivo de la representación de la importancia de los hoyos. Algunos no serían buenos (por ejemplo, cualquier función que manipule los pesos p_i sencillamente equivaldría a establecer un valor $p_{new} = f(p)$ con una distribución de probabilidad no uniforme para los posibles valores de los $p_{i_{new}}$), y otros son más complejos pero posiblemente mejores (aplicar funciones sobre los h_i, ya sean polinómicas o de cualquier tipo).

Para simplificar y acelerar el proceso de pruebas, y basándonos en la premisa de que la diferencia entre las elecciones de estos distintos tipos de heurísticas no es la suficiente como para compensar el tiempo de ejecución (lo cual no tiene por qué ser cierto), vamos a estandarizar esta combinación lineal de pesos y semillas para trabajar de ahora en adelante.

Una vez tomada esta decisión, es conveniente estudiar de qué formas se van a generar las heurísticas de este tipo. A partir de ahora, las heurísticas basadas en la combinación lineal de pesos y semillas recibirán el nombre de *weighted heuristics* (`heuristicWeight` en C), de forma que todo componente que incluya las siglas **WH** trabajará con este tipo de heurísticas.

2.2.2. Generación de heurísticas

Aquí es donde entra en juego el tipo de datos mencionado anteriormente, `generateWH`. Como su nombre indica, se tratan de punteros a funciones que generan *weighted heuristics*; en términos de representación de las heurísticas en C, como es evidente son sencillamente un array de 14 números de tipo `float`.

En este sentido, y tras numerosas pruebas sobre cómo era conveniente generar este tipo de heurísticas y si existían una serie de patrones para crear heurísticas buenas desde su propia generación (pruebas que serán detalladas más adelante), se ha llegado a la conclusión de que la importancia de unos hoyos viene determinada por la importancia que se le da a otros (por ejemplo, asignarle un peso muy positivo al kalaha propio suele venir asociado a asignarle un peso muy negativo al último hoyo antes de dicho kalaha), y localizar todos estos patrones es demasiado costoso computacionalmente; de hecho, lo más conveniente hubiera sido utilizar una red neuronal para descubrirlos, pero se ha considerado un método demasiado avanzado y complicado para conseguir el objetivo final de la práctica.

En consecuencia, se ha optado por la solución más sencilla: generar los pesos aleatoriamente, y luego realizar las pruebas convenientes con la heurística resultante. También se ha añadido una función que genera una mutación a partir de una heurística dada cambiando mínimamente sus pesos, pero para las pruebas finales no se ha tenido en cuenta como mecanismo de generación de heurísticas (pero veremos que el concepto de mutación sí que aparecerá en algún momento implementado de forma diferente).

2.2.3. Almacenamiento de heurísticas

Para realizar pruebas sobre heurísticas debemos contar con un mecanismo de almacenamiento de las mismas. Para ello se implementa el **módulo WHDB** (ficheros `whdb.c` y `whdb.h`), un módulo que gestiona bases de datos de heurísticas a través de ficheros de una forma muy sencilla: guardando en dicho fichero el número de heurísticas que contiene y, a continuación, además de alguna información de control, las heurísticas con sus valores de 14 en 14.

Este módulo también incluye mecanismos de actualización de la base de datos asociados a las diferentes pruebas que se realizan sobre ellos. Dichas bases de datos, una vez cargadas desde sus ficheros correspondientes, se guardan en una estructura en la que se dispone de diversa información de control y de un array de arrays que contiene todas las heurísticas.

2.3. Pruebas y obtención de heurísticas buenas

Disponiendo ya de los mecanismos para generar y almacenar heurísticas, tenemos tres posibles métodos con los cuales realizar las pruebas, que de forma ideal deben simular algún tipo de aprendizaje automático, de manera que a medida que se hagan más pruebas obtengamos heurísticas mejores.

Sin embargo, hay que tener en cuenta que Mancala es un juego demasiado simple desde el punto de vista de las funciones de evaluación como para poder realizar análisis muy profundos de cada estado a partir de las semillas contenidas en cada hoyo. El primero de los tres métodos es, como ya se ha mencionado anteriormente, la construcción de una red neuronal que detecte patrones en las partidas para determinar las decisiones, pero tiene un problema: en nuestra práctica se nos pide que calculemos una serie de heurísticas, no que aportemos una función que directamente decida el movimiento teniendo un estado delante.

Para que la red neuronal funcione, deberíamos aportarle una base de datos de partidas (o sencillamente dejar que las genere solas), de manera que se creara un conjunto de entrenamiento cuyas entradas sean diferentes estados y cuyas salidas fueran los movimientos óptimos dado cada estado; pero como necesitamos una función de evaluación que genere un número asociado a cada estado para la aplicación del algoritmo minimax/negamax, ahora las salidas del conjunto de entrenamiento deberían ser las salidas de la función de evaluación aplicada al estado a una cierta profundidad del árbol de decisiones generado a partir del estado inicial: en resumen, una locura totalmente inviable. Por tanto, hay dos métodos restantes con los cuales ir testeando las diferentes heurísticas, que son los siguientes:

- **Aleatoriedad completa con actualización**. Método más simple: se genera una base de datos con una cantidad concreta de heurísticas, y a partir de ahí se genera una heurística nueva que se

prueba sobre toda la base de datos; si obtiene un número de victorias por encima de un umbral, se actualiza la base de datos eliminando la heurística más antigua por la más nueva.

La ventaja de este método es, además de la simpleza, su rapidez (se juegan pocas partidas por lo general, aunque depende del tamaño de la base de datos) y su capacidad de aprendizaje (dado que siempre se actualiza la base de datos con heurísticas mejores); sin embargo, la desventaja principal es que esta capacidad de aprendizaje es discutible, dado que puede ocurrir que la base de datos original no cubra todos los posibles estilos de juego, y por tanto se vayan generando constantemente heurísticas que ganan frente a unos estilos de juego concretos y no lo hacen frente a otros.

Por tanto, este método depende de la calidad y amplitud de la base de datos original (lo cual no siempre se puede controlar) y de la pura "suerte" de que se genere aleatoriamente una heurística buena. Es más: puede ser que se generen aleatoriamente heurísticas buenísimas que, sin embargo, pierdan contra otras que por lo general son malas pero que en este caso concreto ganen contra la buena. La poca complejidad de Mancala en este sentido provoca un componente muy aleatorio sobre la calidad de cada heurística.

- **Aplicación de un algoritmo genético**. Con este método, disponemos de una base de datos de heurísticas sobre la que vamos a aplicar el algoritmo genético y otra base de datos que va a servir para determinar la "adaptación" de cada heurística de la primera base de datos; es decir, en cada proceso evolutivo las heurísticas de la primera base de datos que tengan más porcentaje de victorias contra las heurísticas de la segunda base de datos van a tener más probabilidad de sobrevivir.

  Como en todo algoritmo genético, partiendo de una base de datos de heurísticas y de sus porcentajes de victorias, se le aplica un algoritmo de selección (según el porcentaje de victorias, cuanto más alto sea más probabilidades de ser seleccionado para la siguiente generación), un algoritmo de cruce (de las heurísticas que no son seleccionadas, se agrupan en pares, y de cada par se obtienen otras dos heurísticas originadas a partir de elegir pesos de una o de otra de forma aleatoria) y un algoritmo de mutación (hay una cierta probabilidad de que cualquier heurística cambie de forma ligera sus pesos).

  Por tanto, se puede iterar esta creación de generaciones y su testeo contra una base de datos hasta que se obtiene una heurística con un porcentaje de victorias que supere un umbral determinado, consiguiendo así una heurística suficientemente buena como para constituir un buen jugador de Mancala.

  La ventaja de este método es que, supuestamente, hay un aprendizaje real: poco a poco nos vamos acercando más a la solución ideal dado que las mejores heurísticas suelen permanecer en la base de datos, y las demás se van cruzando hasta que son también buenas. Sin embargo, esto podría ser discutible, dado que, como hemos comentado anteriormente, una heurística muy buena en general puede ser muy mala en situaciones específicas.

  La principal desventaja es que exige una capacidad de computación muy alta, dado que en cada proceso iterativo, cada partida de la base de datos correspondiente a la población de heurísticas hay que testearla contra toda la base de datos de testeo, generando una cantidad muy grande de partidas; y como, además, las mutaciones y los cruces se van generando poco a poco, cuesta mucho llegar a una generación lo suficientemente avanzada como para satisfacer nuestras necesidades.

Además de los dos métodos generales de testeo que acabamos de comentar, pueden ser optimizados con otras pequeñas pruebas que, además de asegurarnos la validez de las heurísticas (es decir, que ganan a la estrategia "Regular" empezando con el turno o sin él), optimiza todos los procesos iterativos:

- Limitar la generación de heurísticas (y de bases de datos de heuríticas) a las que ganen a la estrategia "Regular" empezando con el turno o sin él.
- Realizar la misma limitación que en el apartado anterior, pero incluyendo también las victorias ante la estrategia "Buena".
- En el caso del método aleatorio con actualización, incrementar el umbral de porcentaje de victorias cada vez que se actualiza la base de datos, para que cada vez sea necesario ganar con más frecuencia.

Todos estos métodos y pequeñas pruebas están contenidos en los módulos **de testeo** y **genético**, implementados en `tests.c`, `tests.h`, `genetic.c` y `genetic.h`.

3. Jugador regular vs Jugador bueno

La segunda pregunta de las solicitadas en la práctica nos pide explicar por qué el jugador bueno pierde en ocasiones frente al jugador regular, cuando el jugador bueno debería ganar siempre puesto que expande el árbol de decisión una profundidad más en todo momento.

La razón por la que pasa esto es meramente por un detalle de la implementación del jugador bueno. Este jugador viene implementado de la misma manera que el regular, pero tomando el máximo de todos los valores tras aplicar la heurística a los posibles estados posteriores desde el que se encuentra (por ejemplo, si elegimos profundidad 2, toma el máximo de los valores tras aplicar las heurísticas a profundidad 3, y a partir de ahí realiza las mismas operaciones que el jugador regular).

Sin embargo, como para asegurar la existencia de dichos estados de posteriores necesitamos comprobar que la partida en el estado a profundidad 2 no ha terminado, se incluye la siguiente línea de código como comprobación:

```
(if (juego-terminado-p estado)
    -50                              ;; Condicion especial de juego terminado
```

Pero esta implementación es errónea; o quizá, más que erronea, deberíamos decir imprecisa. La consecuencia de esta comprobación es que cuando la partida se ha terminado en el estado a profundidad 2, siempre se devuelve un valor que va a ser más pequeño que cualquiera que se genere con la función de evaluación (dado que es la diferencia entre la suma de los valores de un lado del tablero menos el otro, como mínimo será -36); incluso cuando en esta situación el jugador bueno ha ganado la partida en dicho estado. Por ejemplo, en la partida entre ambos aparece la siguiente situación tras 10 jugadas:

```
                TABLERO:

          5   4   3   2   1   0
        ------------------
          7   0   0   8   0   0      18   Ac:   4
      3                         3
    *   1   3   1   0   5   5      18   Ac:   1
        ------------------
          0   1   2   3   4   5
```

Aquí podemos ver que tras la jugada `(R 2)` por parte del jugador bueno, que tiene el turno, significaría que captura las 8 semillas del hoyo 2 del enemigo, acumulando pues 12 en su marcador; y en la siguiente jugada el enemigo sólo puede realizar el movimiento `(R 5)`, con el cual se queda sin semillas en su lado del tablero y la partida termina a favor del jugador bueno. Sin embargo, esta situación, con la implementación actual, nunca va a tenerse en cuenta, puesto que la función de evaluación devolverá un valor de -50 a todos los estados que surjan desde la jugada `(R 2)` y en consecuencia nunca se va a elegir.

Una solución sería, por ejemplo, quitar esa comprobación y no tener en cuenta las jugadas imposibles, cambiando el código mostrado anteriormente por el siguiente:

```
(if (juego-terminado-p estado)
    (- (suma-fila (estado-tablero estado)
                  (estado-lado-sgte-jugador estado))
       (suma-fila (estado-tablero estado)
                  (lado-contrario (estado-lado-sgte-jugador estado)))))
```

También se podría implementar en ambos jugadores, tanto el regular como el bueno, una condición en la que si se encuentran con un estado de partida terminada, devolvieran 50 en caso de ganarla y -50 en caso de perderla; o cualquier otra variante que implicara no penalizar de forma incorrecta a una jugada ganadora.

Tras aplicar los cambios en el código propuestos, la partida pasa a ser ganada por el jugador bueno.

```
FIN DEL JUEGO por VICTORIA en 12 Jugadas
Marcador:  Bueno 32 - 4 Regular
```

NOTA: Por algún motivo, realizar el cambio directamente sobre `f-eval-Bueno` no es suficiente. Si en vez de usar el jugador bueno creado con la instrucción `defvar` se crea otro usando un `setf`, sí que se obtiene el resultado anterior de 32-4; usando el jugador de `defvar` no cambia nada, por razones desconocidas.