Inteligencia Artificial. Memoria práctica 3

Juegos con adversario: Mancala

Antonio Coín Castro. 3º DGIIM

1. Introducción

El objetivo de esta práctica es desarrollar un bot que juegue al juego de Mancala. Para ello debemos programar un comportamiento inteligente en el mismo, para que en cada turno elija el movimiento más prometedor.

Una aproximación inicial sería implementar el algoritmo minimax con poda alpha-beta visto en clase. Sin embargo, optamos por el algoritmo MTD-f [1], que se trata de una variación de la poda alpha-beta con tamaño de ventana cero.

2. Diseño del algoritmo

Representación de un estado de la búsqueda

Para representar un estado en el espacio de búsqueda, empleamos una estructura de datos que agrupa los datos esenciales, a saber:

```
Node {
  h;              // Evaluation function chosen for nodes
  board;          // Current board state
  prev_move;      // Movement performed to get to this node
  h_value;        // Heuristic value of node
  is_maximizing;  // Whether this is a MAX or a MIN node
};
```

Como vemos, en cada nodo guardamos el estado del tablero, el movimiento que nos permitió llegar hasta él, su valor heurístico (dado por una función de evaluación concreta), y si se trata de un nodo MAX ó MIN.

Además, se añade un método para generar los hijos de un nodo, que devuelve una lista de 6 nodos. Cada uno de estos hijos es el resultado de seleccionar uno de los posibles casilleros, obtenidos gracias a la función `simulateMove`, y teniendo en cuenta que si el jugador asociado es el mismo que el del padre debemos mantener el mismo tipo (MAX ó MIN) que el padre, mientras que si el jugador es distinto, cambiamos el tipo.

La implementación de los métodos se encuentra en el archivo `AlumnoBot.cpp`.

Descripción general del algoritmo MTD-f

La idea del algoritmo es realizar una serie de búsquedas apha-beta con tamaño de ventana 0, usando una "buena" cota. El pseudocódigo del algoritmo es el siguiente:

funct $mtdf(root, f, d)$
  $g := f;$
  $upperbound := INFINITY$
  $lowerbound := -INFINITY$
  while $lowerbound < upperbound$ do
    $beta := max(g, lowerbound + 1)$
    $g := alphaBetaWithMemory(root, beta - 1, beta, d)$
    if $g < beta$
      then $upperbound := g$
      else $lowerbound := g$
    fi
  od
  return g

En este caso, la llamada a la función de búsqueda no devuelve el valor minimax, sino una cota del mismo, y para converger al valor exacto debemos realizar una serie de iteraciones actualizando las cotas inferiores y superiores según sea necesario.

Se puede comprobar que con llamadas de ventana 0, el algoritmo de poda alpha-beta consigue cortar ramas antes. Además, se necesita una primera aproximación del valor minimax (variable `first_guess`), que en una primera llamada puede ser 0.

Detalles de implementación

Una vez programado el algoritmo MTD-f, realizamos una **búsqueda iterativa en profundidad** hasta un máximo fijado (`MAX_DEPTH`). En cada iteración realizamos una llamada a `mtdf`, actualizando el `first_guess`, y saliendo del bucle si vemos que dicha función ha devuelto un código especial (`TIMEOUT`) que indica que estamos cerca consumir los 2 segundos permitidos.

Programamos una función `alphaBetaWithMemory`, que devuelve una pareja `<Bound,Move>` representando el mejor valor esperado (cota) y el movimiento asociado al mismo. Esta función implementa básicamente el algoritmo de poda alpha-beta visto en clase, con la salvedad de que emplea una **tabla *hash*** para ir guardando los nodos entre iteraciones.

Esta tabla hash se implementa como un `unordered_map`, con una función *hash* ingenua construída a partir del estado de un tablero, pues consideramos que dos nodos son iguales si sus tableros asociados lo son. La utilidad que tiene es que si encontramos un nodo en la tabla, tenemos información sobre una búsqueda anterior y las correspondientes cotas.

Creamos una estructura de datos adicional, `NodeInfo`, que representa una entrada de la tabla, indexada por un objeto de tipo `Node`. En ella guardamos la profundidad a la que nos encontramos el nodo en la búsqueda, la "edad" de la información, la cota asociada al nodo (superior o inferior) y el mejor movimiento que sabemos hasta ahora que puede realizar a partir de él.

También consideramos una **ordenación de movimientos**, pues es bien sabido que la poda alpha-beta es óptima si se exploran primero los mejores nodos. En nuestro caso, ordenamos los hijos de un nodo al crearlos en base a su valor heurístico. Además, si consultamos un nodo en nuestra tabla, probamos primero el movimiento que tiene asociado, con la esperanza de que sea el mejor y produzca un corte.

Por último, cabe destacar que la medición de tiempos se realiza utilizando la librería `std::chrono`. La estrategia es medir en cada iteración de la función `alphaBetaWithMemory` el tiempo total consumido, y si llega a un umbral predefinido, devolver recursivamente un código de error. Es por esto que **en todos los movimientos se consume casi todo el tiempo permitido**, lo que nos permite llegar a profundidades grandes sin miedo a quedarnos sin tiempo.

3. Heurística

En cuanto a la heurística elegida para el valor de los nodos hoja, se ha optado por una métrica sencilla pero efectiva: **la diferencia entre el número de semillas en nuestro granero y en el de nuestro oponente**.

En un principio se probaron otras heurísticas, que medían por ejemplo lo cerca que estaba el rival de ganar, la profundidad del nodo evaluado, o la obtención de turnos extra. También se hicieron pruebas combinando alguna de estas heurísticas, pero la conclusión fue que la heurística elegida (resta de graneros) era la más eficaz y segura, pues proporcionaba el mayor número de victorias.

La representación de la heurística se encapsula en una clase `Heuristic`, estando esta a su vez dentro de la clase `Node`. La función heurística necesita conocer si jugamos como `J1` o como `J2`.

Referencias

[1] *Aske Plaat (1997).* MTD(f): A Minimax Algorithm faster than NegaScout. https://people.csail.mit.edu/plaat/mtdf.html