MEMORIA PRÁCTICA 3: MANCALA

Vladislav Nikolov Vasilev, 2ºA

June 3, 2018

1 Introducción

En esta práctica se ha pedido a los alumnos que implementen un jugador automático para el juego *Mancala*. Para ello, se han tenido que utilizar algoritmos de búsquedas en juegos, como por ejemplo el *minimax* o la *poda alfa-beta*. Además, como restricción, el tiempo que debían tomar las búsquedas no podían ser superiores a los 2 segundos, o lo que viene siendo lo mismo, no explorar más de 140.000 nodos.

En esta memoria se explicará el diseño que se ha realizado de los estados, del algoritmo de búsqueda que se ha implementado y de la heurística empleada para evaluar los estados.

2 Diseño de la búsqueda

Vamos a comenzar hablando del diseño de los estados que se ha utilizado. En lo referente a este punto no existe mucho misterio, ya que ha utilizado la estructura de datos *Gamestate* para manejar los distintos estados del juego posibles, tanto a la hora de generar un nuevo estado a partir de uno ya dado como para realizar la evaluación de éstos y decantarse por uno u otro.

Pasemos ahora a hablar del algoritmo de búsqueda que se ha utilizado. En esta versión se ha implementado la *poda alfa-beta*, ya que se ha considerado que es la más adecuada tanto por las restricciones que se han impuesto inicialmente (tiempo/número de nodos) como por la cantidad de nodos diferentes que puede explorar. Esto último se debe a que, al realizar la poda, permite llegar a una mayor profundidad de la que la técnica *minimax* ofrece, y por tanto, en función de la herurística, permite determinar con mayor precisión cuál es la mejor jugada posible.

Las restricciones explicadas anteriormente a las que está sujeta la búsqueda implican que no se puede explorar todo el árbol de estados hasta llegar a una solución satisfactoria. Por tanto, en esta implementación se ha tenido que imponer un límite de profundidad hasta el que se puede descender. Para el *jugador 1*, este límite es fijo y tiene un valor de 11. Para el *jugador 2*, en cambio, el límite inicial es 11, pero a medida que se va explorando, si el número de nodos baja de un cierto umbral, este límite se ve incrementado en una cierta cantidad. El umbral inicial es 60.000 nodos explorados, y cada vez que se baje de ese umbral, éste se ve reducido a la mitad y el nivel de profundidad se ve incrementado en una cantidad variable. Esta cantidad depende del número de veces que se haya superado el umbral. Si se ha superado una vez, el límite actual se ve incrementado en 1, y pasaría a 12. Si se supera una segunda vez, el límite actual (teniendo en cuenta el cambio anterior) se ve incrementado en 2, y por tanto, pasaría a 14, y así sucesivamente. Se ha decidido hacer de esta forma ya que el jugador tiene una mayor desventaja siendo *jugador 2*, ya que no es

capaz de realizar el/los primer/os movimiento/s, y que además el número de nodos explorados comienza a bajar una vez que se han realizado un número de movimientos en la partida, y por tanto no hay peligro de que se pase en cuanto al número de nodos explorados o que se pase de tiempo. Para hacer que esto fuese posible, el jugador tiene un atributo *nodosExplorados* que almacena cuantos nodos se han explorado en cada jugada (se actualiza cada vez que se realiza una llamada a la *poda alfa-beta*.

La implementación del algoritmo es como la vista en clase y no necesita ser explicada de nuevo, aunque hay que destacar algunos aspectos:

- Un estado es evaluado si se ha llegado al límite de profundidad o si se trata de un estado terminal.
- Los nodos *max* se corresponden con los estados en los que el jugador tiene que realizar un movimiento, mientras que los *min* se corresponden con aquellos en los que el oponente tiene que realizar un movimiento. Para determinar si un estado es *max* o *min* a la función de búsqueda se le pasa un parámetro que es el *jugador actual* en el turno inicial, y aquellos estados en los que se corresponda el jugador actual con el que se le ha pasado como parámetro, son nodos *max*, y en caso contrario serán nodos *min*. La implementación de esto es trivial, ya que se trata de una mera comprobación que se tiene que realizar cada vez que se llame a la función.
- Para determinar el mejor movimiento a realizar, el nodo correspondiente al estado inicial comprueba si el valor que ha obtenido de realizar la búsqueda para un movimiento es mejor que el alfa actual, y si lo es, guarda el movimiento que ha llevado a esa valoración. Para poder guardar el movimiento, como la búsqueda devuelve un valor entero, se ha incluido un atributo en la clase del jugador que es *movimiento*. Una vez terminada la búsqueda, la función *nextMove* devuelve este movimiento.

Pasemos, finalmente, a analizar la heurística utilizada para evaluar los estados. Al principio se había utilizado una heurística muy sencilla, que es la *diferencia de graneros*. Siendo G_j la cantidad de semillas en el granero del jugador y G_c la cantidad de semillas del contrincante en su granero, entonces G (diferencia) se puede obtener como:

$$G = G_j - Gc$$

A pesar de su sencillez, la heurística ofrecía unos resultados medianamente aceptables. Sin embargo, la diferencia de los graneros no es el único factor que debería influir, ya que pudiera ser que la cantidad de semillas que tiene un jugador en su campo es muchísimo mayor que la que tiene el otro, y por tanto, puede llegar a meter más semillas en su campo. Por tanto, se han considerado también otros factores, como la diferencia entre las casillas vacías del jugador y el contrincante, V, la diferencia entre el número de *semillas seguras*, S, (semillas

que independientemente del movimiento permanecen en el campo del jugador, y a estas se les resta el número de semillas seguras del contrincante), la diferencia entre el máximo número de semillas que puede llegar a perder el contrincante y las que puede llegar a perder el jugador, P, (se pretende dar más prioridad a aquellos estados en los que el contrincante puede llegar a perder una mayor cantidad de semillas, es decir, que pasen al campo del jugador) y la diferencia entre el máximo de semillas que puede robar el jugador y el máximo de semillas que le pueden robar a él, R, (se pretende dar más prioridad a aquellos estados en los que el jugador puede robarle una mayor cantidad de semillas al contrincante).

Por tanto, combinando los anteriores parámetros y dándoles un peso adecuado, la heurística *h(n)* quedaría de la siguiente forma:

$$h(n) = G + V + 3S + 2P + 2R$$

El peso que se le ha dado a cada parámetro se ha determinado a base de prueba y error, ya que es difícil realizar un ajuste teórico de los parámetros para saber cuál tiene un mayor o menor peso. Se ha intentado balancear el peso de cada parámetro en función de la profundidad máxima o promedia a la que puede llegar la búsqueda para que ofrezca los mejores resultados posibles.

3 Otros comentarios

La heurística utilizada ha dado resultados bastante buenos, ya que poda un número elevado de nodos con una profundidad relativamente alta (11 o más, dependiendo del lado del jugador) y en ningún momento se pasaba ni de tiempo ni de nodos. Además, ofrece muy buenos resultados cuando el jugador realiza el primer movimiento, ya que gana en casi todas las ocasiones, o en el peor de los casos, empata. No obstante, en el caso en el que el jugador no comience primero, no ofrece unos resultados tan buenos contra otros jugadores creados por otros alumnos, ya que sus heurísticas les permiten obtener una mayor ventaja del hecho que han comenzado primeros. A pesar de eso, el jugador puede llegar a ganar a éstos, o incluso empatar, cuando él no realiza la primera jugada, auqne estos casos no se dan con mucha frecuencia. Cabe mencionar, por último, que el jugador es capaz de ganarle por mucha diferencia al *GreedyBot* independientemente del lado en el que se encuentre.