Memoria de prácticas

PRÁCTICA 3

AGENTES EN ENTORNOS CON ADVERSARIO

UNIVERSIDAD DE GRANADA

Realizado por: Sergio Azañón Cantero

Grupo B3

Diseño de estado

ACLARACIÓN: Esta memoria la he ido realizando mientras iba realizando la práctica por lo que es una evolución desde el algoritmo minimax a la poda alfa-beta.

Para representar un estado en el juego del Mancala empezé utilizando una clase Nodo que tenía como parte privada un Gamestate, es decir, un estado actual del tablero, un movimiento asociado a dicho nodo (gráficamente sería como el arco que llega a dicho nodo, el movimiento que hay que hacer para llegar a ese estado) y una list de la stl donde guardaba los hijos de cada nodo. Como métodos públicos utilizaba prácticamente los mismos que ya tenía de por sí la clase Gamestate excepto un setMovimiento que modificaba el movimiento (pero que casi no se utilizaba) y un método para insertar un hijo.

También había declarado un struct llamado Accion_Valor que tenía un valor y un movimiento asociado. Este struct lo utilizaba para cuando la funcion minimax devolvia un valor y un movimiento los guardaba ahí y comparaba para ver si lo guardaba para devolver o no. Esta estructura tampoco tenía mucho sentido ya que sólo me interesa el valor heurístico de la funcion minimax y cuando tenga ese valor, devuelvo la accion asociada. Por lo que la funcion minimax ahora devuelve un entero y he eliminado ese struct.

Como esta estructura no era muy eficiente, era más compleja de definir, utilizaba los métodos de otra clase y además la lista de hijos no era necesaria ya que recursivamente ibas opteniendo los valores de cada nodo, he optado por cambiarlo por un struct Nodo (declarado en Nask.h) que tiene un Gamestate y un movimiento asociado (como ya he explicado anteriormente).

Pero en comparación de tiempos con mis compañeros me salía mucho más, por lo que al final también eliminé ese struct. Actualmente un estado es un Gamestate que se le va pasando a la función minimax.

Por lo que la funcion base (funcion que iniciará la recursividad), que en este caso es NextMove, le pasa a la función min_max es un Gamestate (estado), una variable bool que indicará si está minimizando o maximizando (true → minimizar, false → maximizar), una profundidad, el alfa y el beta.

Explicación del diseño del algoritmo implementando

Inicialmente el algoritmo que he implementado es un algoritmo minimax con una profundidad fija ya que no se podría generar el árbol de estados entero en un tiempo razonable y viable. El algoritmo empieza, en la funcion base (nextMove) inicializando el valor heuristico a INT_MIN que es el valor mínimo que cabe en un entero ya que a la hora de elegir qué valor coger, la primera iteración siempre será mayor que ese valor (ya que nosotros seríamos MAX)y la profundidad la inicializamos a 1 .Le pasamos al nodo raíz (struct explicado anteriormente) el estado que nos han pasado por parámetro. Generamos los nodos de las posibles acciones que podemos realizar, que en este caso, son 6 movimientos posibles de las 6 casillas que tenemos en nuestra parte del tablero, pero solo generamos los hijos que en su casilla tenga mas de 0 semillas para no generar los movimientos

inválidos (ya que si escogiéramos un movimiento en el que no hay semillas perderíamos la partida por 48 a 0).

Al generar los hijos, si el turno del estado hijo es del siguiente jugador llamamos a la funcion minimax recursiva indicando que vamos a minimizar, es decir con el parámetro booleano a true. Si el siguiente turno es nuestro (porque tenemos un turno extra) llamamos a la funcion minimax con false ya que vamos a maximizar.

La función recursiva minimax empieza mirando si el nodo que le han pasado es un nodo terminal, ya que ha terminado el juego, o si ha llegado a la profundidad fijada (que en mi caso por ahora está a 10). Si no es terminal o no ha llegado a la profundidad, inicializa el valor heurístico a INT_MIN si estamos minimizando o a INT_MAX si estamos maximizando. Generamos los hijos del nodo que nos han pasado como parámetro (sólo los hijos que tienen mas de 0 semillas en la casilla) y volvemos a llamar a la funcion minimax maximizando o minimizando dependiendo del turno del siguiente jugador. Cuando va retornando recursivamente comparamos el valor que tenemos con el de cada nodo y nos vamos quedando con el menor si estamos minimizando o con el mayor si estamos maximizando. Hasta que retorne a la primera iteración de la función base, comparemos si el valor retornado es mayor que el que tenemos y volvemos a llamar a la función recursiva.

Finalmente, he mejorado el algoritmo minimax anterior en un algortimo poda alfa-beta. Sin explicar lo que teóricamente supone un algoritmo alfa-beta, en código solo se diferencia con el minimax en una pequeña parte. He añadido dos enteros alfa y beta donde inicializo alfa a menos infinito y beta a más infinito. Sólo hay que añadir la parte de si cuando estamos minimizando el valor que tenemos guardado es menor o igual que alfa podamos y si el valor que tenemos guardado es mayor que beta actualizar beta. Para cuando estamos maximizando sería totalmente al revés.

Como última actualización, ahora el parámetro que indica el estado lo paso por referencia constante en vez de por valor, a la función recursiva ya que no se modifican los estados y mejora mucho en tiempo de ejecución.

Heurística utilizada

He utilizado una heurística bastante simple para no hacer calculos numéricos muy grandes y que la calcule lo más rápido posible. Mi función heurística inicialmente crea una variable que inicializa a 0 donde almacenará el valor heurístico que devolverá y calcula la diferencia de puntuación entre un jugador y el otro (sin valor absoluto) almacenándola en una variable. Si el estado del nodo es victoria para él, le sumo 100 puntos y el valor absoluto de la diferencia, por el contrario, si el estado del nodo es derrota , le resto 100 puntos y el valor absoluto de la diferencia. Si no hay ganador aún, le sumo/resto (ya que ahora es sin valor absoluto) la diferencia de puntuación. Y devuelvo la variable donde he almacenado el valor.

Probé con pequeña modificaciones en la heurística como multiplicar por un porcentaje del numero de semillas que te quedaban en tus casillas o restarle a lo anterior explicado la profundidad actual pero en algunos casos no llegaba a funcionar correctamente y opté por suprimirla.

Por último he añadido a la heurística la suma de las semillas de cada casilla del jugador actual. Ya que así podemos saber cuantas semillas tenemos en nuestra parte del tablero para mejorar la heurística un poco más.