Memoria: Conecta-4 Crush

Métodos de Búsqueda con Adversario

LOTHAR SOTO PALMA
Universidad de Granada
23 de mayo de 2015

1. Información

Atención, el apartado más importante de la memoria se llama **Comportamiento**, en específico el apartado **Función heurística** números 3.2.1 y 3.2.2 respectivamente.

2. Análisis del problema

El problema a tratar es jugar al conecta-4 crush, es una variante del conecta-4 usual que contiene una pieza llamada bomba que elimina las posiciones adyacentes. El problema es un juego de información perfecta para dos jugadores por lo que la idea principal es minimizar la perdida máxima propia a la hora de jugar, para ello su usa una implementación del algoritmo minimax con poda alfa-beta. La mayor dificultad es la busqueda de una función heurística que resuelva el problema.

2.1. Espacio de estados

En este problema es espacio de estados a tratar es muy sencillo de identificar porque es similar al usado en el ajedrez, el estado es el tablero en cada momento y los arcos son las acciones posibles que se pueden realizar sobre el tablero.

3. Descripción de la solución planteada

3.1. Idea Inicial

La idea inicial fue usar una heurística muy basica que consistía en la suma de las filas, columnas y diagonales de cada jugador que tienen al menos una pieza del jugador y restarlas, es decir:

La función heurística llamada en el programa Valoración() realizaría la siguiente operación:

$$jugactual = filas + columnas + diagonales$$

$$rival = filas + columnas + diagonales$$

$$valor = jugactual - rival$$

Sin embargo no daba los resultados esperados, y por tanto me planteé modificar la heurística ligeramente contemplando más casos que se podían tomar en el tablero, y la idea resultante fue la de bloquear al rival siempre cuando fuera posible dando más valor a las filas, columnas y diagonales de 3 piezas del rival que a las piezas propias, en el apartado comportamiento se explica la función heurística con más detalle.

3.2. Comportamiento

3.2.1. Minimax y poda Alfa Beta

Para realizar la implementación del minimax con poda alfa-beta en primer lugar es necesario una condición de parada puesto que el árbol de juego se va ha construir haciendo uso de recursividad, establecemos como condición de parada o bien cuando se alcanza la profundidad o bien cuando hay un nodo en el que ya se ha ganado o perdido de manera que también se considera un estado terminal. Para ir analizando en cada caso los estados de manera recursiva, cada vez que se llama a la función se crean todos los estados posibles con la función generateAllMoves(), y posteriormente se van analizando los estados y profundizando hasta que se llega a un nodo terminal donde se valora el estado con la función Valoracion(). Para la poda tan solo es necesario ir asignando el máximo de los estados min a la variable alpha, y e ir asignando el valor del mínimo de los nodos max en la variable beta para posteriormente comprobar la condición de poda si $alpha \geq beta$.

3.2.2. Función heurística

Vamos a entrar en detalles con la función heurística usada, en primer lugar es necesario mencionar cuales fueron las funciones usadas en esta heurística para evaluar la situación del tablero en cada momento:

- **RevisarTablero()**: Ya se explica en el guión de la práctica retorna el valor 1 si ha ganado el jugador 1, 2 si ha ganado el jugador 2 y 0 en caso contrario, se usa para comprobar si el juego ha terminado y retornar un valor muy grande o my pequeño en caso de ganar o perder.
- **filas_abiertas(const Environment &estado, int jugador)**: Trata de una función que se encarga de contar las filas abiertas (que contienen al menos una pieza del jugador) del jugador que se les pasa por argumento.
- **columnas_abiertas(const Environment &estado, int jugador)**: Trata de una función que se encarga de contar las columnas abiertas (que contienen al menos una pieza del jugador) del jugador que se les pasa por argumento.
- **diagonales_abiertas(const Environment &estado, int jugador)**: Trata de una función que se encarga de contar las diagonales abiertas (que contienen al menos una pieza del jugador) del jugador que se les pasa por argumento.
- **nfilas_3(const Environment &estado, int jugador)**: Es una función similar a la anterior, calcula el número de filas de tres piezas del jugador actual.
- **ncolumnas_3(const Environment &estado, int jugador)**: Es una función similar a las anteriores, calcula el número de columnas de tres piezas del jugador actual.

- **ndiagonales_3(const Environment &estado, int jugador)**: Es una función similar a las anteriores, calcula el número de diagonales de tres piezas del jugador actual.
- **Get_Casillas_Libres()**: Ya se explica en el guión de la práctica, retorna el número de casillas libres.

La función heurística actua de la siguiente forma:

1. En primer lugar aplicamos la función RevisarTablero().
2. En caso de ganar se retorna el valor 999999.0 (double) en caso de perder se retorna el valor -999999.0.
3. Comprobamos si el tablero esta completo haciendo uso de la función Get_Casillas_Libres(), si vale 0 significa que no hay ninguna casilla libre y por tanto el tablero está lleno y por tanto se retorna 0.
4. Ahora si no ha ocurrido nada de lo anterior, realizamos algo parecido a lo que se explicó en la idea inicial, se evaluan el número de filas, columnas y diagonales con al menos una pieza para cada jugador y además se evaluan las filas, columnas y diagonales con 3 piezas para cada jugador, sin embargo, se multiplica por 1000 el valor de las filas, columnas y diagonales de 3 piezas del rival, es decir le damos más importancia, esto nos permitirá bloquear al rival en todo caso puesto si se forma una figura de tres piezas del rival obtendrá la mayor valoración a no ser que ya se haya ganado. En el caso de los pesos del jugador opuesto al rival tan solo le damos la misma importancia a las diagonales porque es una de las estructuras más difíciles de formar y nos interesa que se formen lentamente para intentar combinarlas con otras esructuras o figuras. Se realiza lo siguiente:

$$jugactual = filas + columnas + diagonales + filas3 + columnas3 + 1000 * diagonales3$$

$$rival = filas + columnas + diagonales + 1000 * (filas3 + diagonales3 + columnas3)$$

$$valor = jugactual - rival$$

Donde filas, columnas y diagonales se corresponden con filas_abiertas(), columnas_abiertas() y diagonales_abiertas(), y filas3, columnas3, diagonales3 se corresponden con nfilas_3(), ncolumnas_3() y ndiagonales_3() respectivamente y valor es el resultado final de la valoración.

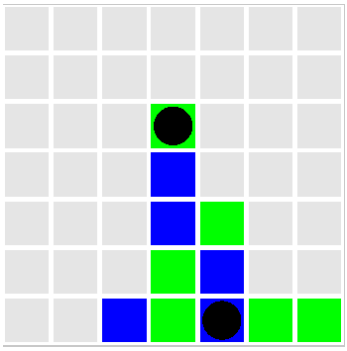
3.3. Resultados

En mi caso la manera de hacer pruebas para determinar lo buena o mala que es mi heurística inicialmente fue jugar contra el ninja, y la heurística usada consigue la victoria tanto si empieza como jugador 1 o si empieza como jugador 2, así que decidí probarlo con los compañeros de clase, y tan solo encontre un caso de derrota, de alrededor de 6 partidas contra heurísticas de distintas personas, por lo que aproximandamente obtiene un porcentaje de victoria de un 83 % contra las heurísticas de las 6 personas con las que se ha probado. La selección de la heurística fue un proceso de refinamiento de una idea básica o otra no tan sencilla con unos pesos más o menos originales.

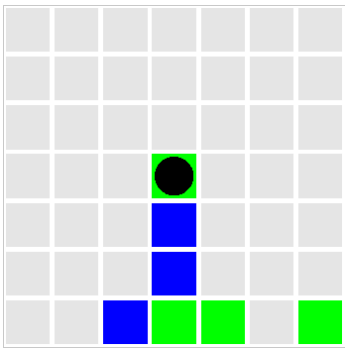
3.3.1. Cambios de la heurística eliminados

En un principio se iba a añadir una función evalua_bomba() para comprobar si resulta bueno o no explotar la bomba, realmente eso ya se tiene en cuenta en la poda alfa beta por lo que se

desechó este cambio, aunque despues se probó a implementar una versión que explotara siempre la bomba pero esto no resultao satisfactorio porque generaba situaciones muy poco beneficiosas como puede ser la siguiente:



En este caso al explotar la bomba siempre se dejaba un hueco que hace que el siguiente jugador pudiese ganar, y no buscamos eso con el algoritmo:



3.3.2. Errores que se han obtenido mediante la realización

Los errores principalmente derivaban de la función poda alfa beta, ya que hay una serie de requisitos que se tienen que dar que no se comprobaban o bien que los valores retornados cuando se ganaba o perdía eran demasiado grandes o chicos respectivamente.