Minimax

June 21, 2017

Abstract

Minimax es una regla de decisión usada en la teoría de la decisión, la teoría de juegos, las estadísticas y la filosofía para minimizar la posible pérdida para un escenario del peor caso(pérdida máxima). O de la misma forma hablando de ganacias, se conoce como "maximin" - para maximizar la ganancia mínima. Originalmente formulado para la teoría de juego de dos jugadores de suma cero, que abarca tanto los casos en que los jugadores toman movimientos alternativos como aquellos en los que realizan movimientos simultáneos, también se ha extendido a juegos más complejos ya la toma de decisiones generales en presencia de incertidumbre. En el siguiente informe se realizará una presentación de un algoritmo donde se optimice la ganancia mínima utilizando balanceo de cargas.

keywords: Teoría de juego, Zero-sum game, Teoría de la decisión

1 Introducción

Existe un montón de aplicaciones en el campo de la IA, pero los juegos son los más interesantes y populares para el público. Estos se aplica en juegos de 2 jugadores como Tic Tac Toe, damas, ajedrez, go, etc. Todos estos juegos tienen al menos una cosa en común, son **juegos de lógica**. Esto significa que pueden ser **descritos por un conjunto de reglas y premisas**. Con ellos, es posible saber desde un punto determinado en el juego, cuáles son los próximos movimientos disponibles. Así que también comparten otras características, son "juegos de información completa". Cada jugador sabe todo acerca de los posibles movimientos del adversario.

En este sentido se tiene dos jugadores involucrados, **MAX y MIN**. Se genera un árbol de búsqueda, DFS, comenzando con la posición actual del juego hasta la posición final del juego. A continuación, se evalúa la posición final del juego desde el punto de vista de MAX. Posteriormente, los valores del nodo interno del árbol se rellenan de abajo hacia arriba con los valores evaluados. Los nodos que pertenecen al jugador MAX reciben el valor máximo de sus hijos. Los nodos para el jugador MIN seleccionarán el valor mínimo de sus hijos.

Entonces, los valores representan lo optimo que es un movimiento del juego ya sea para MAX o MIN. Así que el jugador MAX intentará seleccionar el movimiento con el valor más alto al final. Pero el jugador MIN tratará de seleccionar los movimientos que son mejores para él, minimizando así el resultado de MAX.

2 Optimización

Ahora bien se ha mencionado solo juegos simples que pueden tener su búsqueda del árbol completo en poco tiempo. Para la mayoría de los juegos esto no es posible, el universo probablemente desaparecería primero. Así que hay algunas optimizaciones para agregar al algoritmo.

Pero debemos saber que al optimizar estamos negociando la información completa sobre los eventos del juego con probabilidades y atajos. En lugar de conocer el camino completo que conduce a la victoria, las decisiones se toman con el camino que podría conducir a la victoria. Si la optimización no está bien escogida, o está mal aplicada, entonces podríamos terminar con una IA muda. Y habría sido mejor usar movimientos aleatorios.

- Una optimización básica es limitar la profundidad del árbol de búsqueda. Esto nos ayuda debido a que si tenemos en frente a un problema complejo generar el árbol completo podría tomar mucho tiempo y además poder recorrerlo. Si un juego tiene un factor de ramificación de 3, esto significaría que cada nodo tiene hijos de árbol.

Depth	Node Count
0	1
1	3
2	9
3	27
...	...
n	3^n

La secuencia muestra que a la profundidad n el árbol tendrá 3^n nodos. Para conocer el número total de nodos generados, necesitamos sumar el recuento de nodos en cada nivel. Así que el número total de nodos para un árbol con profundidad n es $\sum_{n=0}^{n} 3^n$. Para muchos juegos, como el ajedrez que tienen un factor de ramificación muy grande, esto significa que el árbol podría no encajar en la memoria.

Incluso si este lo hiciera, tomaría mucho tiempo para generar. Supongamos que cada nodo tome 1s para ser analizado, eso significaría que para la optimización anterior, cada árbol de búsqueda tomaría $\sum_{n=0}^{n} 3^n$. Para un árbol de búsqueda con profundidad 5, eso significaría $1 + 3 + 9 + 27 + 81 + 243 = 364 * 1 = 364s => 6minutos!$. Esto es demasiado largo para un juego. El jugador renunciaría a jugar el juego, si tuviera que esperar 6m por cada movimiento de la computadora.

- La segunda optimización es utilizar una función que evalúa la posición actual del juego desde el punto de vista de algún jugador. Esto lo hace dando un valor al estado actual del juego, como contar el número de piezas en el tablero, por ejemplo, el número de movimientos dejados al final del juego, o cualquier otra cosa que podamos usar para dar un valor a la posición del juego.

En lugar de evaluar la posición actual del juego, la función podría calcular cómo la posición actual del juego podría ayudar a terminar el juego. O en otras palabras, la probabilidad de que con la posición actual del juego, podamos ganar el juego. En este caso, la función se conoce como **función de estimación**.

Esta función tendrá que tener en cuenta algunas heurísticas. Heurística son los conocimientos que tenemos sobre el juego, y ayudar a generar mejores funciones de evaluación. Por ejemplo, en las fichas, las piezas en las esquinas y en las posiciones laterales no se pueden comer. Así podemos crear una función de evaluación que da valores más altos a las piezas que se encuentran en esas posiciones del tablero, dando así mayores resultados para los movimientos del juego que coloquen piezas en esas posiciones.

Una de las razones por las que la función de evaluación debe ser capaz de evaluar posiciones de juego para ambos jugadores es que no sabe a qué jugador pertenece la profundidad del límite. Sin embargo, se pueden evitar dos funciones si el juego es simétrico. Esto significa que la pérdida de un jugador es igual a las ganancias del otro. Estos juegos también se conocen como juegos ZERO-SUM. Para estos juegos una función de evaluación es suficiente, uno de los jugadores sólo tiene que negar el retorno de la función.

Aun así el algoritmo tiene algunos defectos, uno de los defectos es que si el juego es demasiado complejo la respuesta siempre tomará demasiado tiempo incluso con un límite de profundidad. Una solución es limitar el tiempo de búsqueda. Es decir, si el tiempo se agota se elige el mejor movimiento encontrado hasta el momento.

Un gran defecto es el problema horizonte limitado pues una posición de juego que parece ser muy buena puede resultar muy mala. Esto sucede porque el algoritmo no fue capaz de ver que unos pocos movimientos de juego por delante el adversario será capaz de hacer un movimiento que le traerá un gran resultado. El algoritmo perdió ese movimiento fatal porque estaba cegado por el límite de profundidad.

```
In [ ]:
```