Lectura

Inteligencia Artificial

Texto basado en las notas de Wolfgang Ertel dadas en el libro Introduction to Artificial Intelligence de Springer.

---

1 Juegos con oponentes

Los juegos para dos jugadores, como el ajedrez, damas, Otelo y Go son deterministas. Por el contrario, el backgammon no es determinista porque un estado hijo depende del resultado de un lanzamiento de dados. Todos estos juegos son observables porque cada jugador siempre conoce el estado completo del juego. Muchos juegos de cartas, como el póquer, son sólo parcialmente observables porque el jugador no conoce las cartas de los otros jugadores o sólo tiene un conocimiento parcial sobre ellas. Los problemas discutidos hasta ahora son deterministas y observables. En estas notas nos limitaremos a los juegos de suma cero. Estos son juegos en los que cada ganancia que obtiene un jugador significa una pérdida del mismo valor para el oponente. La suma de la ganancia y la pérdida siempre es igual a cero. Esta propiedad se cumple para los juegos de ajedrez, damas, Otelo y Go, mencionados anteriormente.

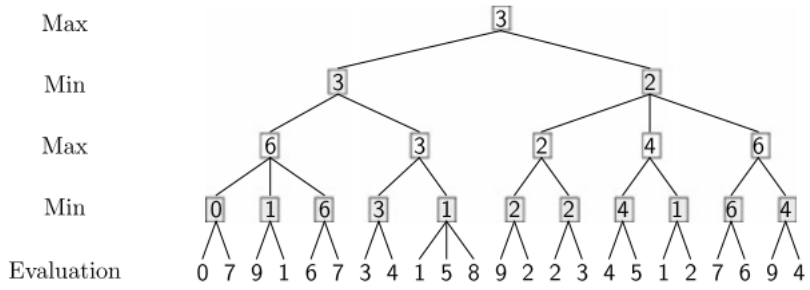
1.1 Búsqueda Minimax

El objetivo de cada jugador es realizar movimientos óptimos que resulten en la victoria. En principio, es posible construir un árbol de búsqueda y realizar una búsqueda completa para una serie de movimientos que resulten en la victoria. Sin embargo, hay varias peculiaridades a tener en cuenta:

1. El factor de ramificación efectivo en el ajedrez es de alrededor de 30 a 35. En un juego típico con 50 movimientos por jugador, el árbol de búsqueda tiene más de $30^{100} \approx 10^{148}$ nodos hojas. Por lo tanto, no hay posibilidad de explorar completamente el árbol de búsqueda. Además el ajedrez a menudo se juega con un límite de tiempo. Debido a este requisito de tiempo real, la búsqueda debe limitarse a una profundidad adecuada en el árbol, por ejemplo, ocho medios movimientos. Dado que entre los nodos hoja de este árbol de profundidad limitada normalmente no hay nodos de solución (es decir, nodos que terminan el juego), se utiliza una función de evaluación heurística B para las posiciones del tablero. El nivel de ejecución del programa depende en gran medida de la calidad de esta función de evaluación.

2. Llamaremos al jugador cuyo juego queremos optimizar `Max` y su oponente `Min`. Los movimientos del oponente `(Min)` no se conocen de antemano y por lo tanto, tampoco el árbol de búsqueda real. Este problema puede resolverse asumiendo que el oponente siempre hace el mejor movimiento que puede. Cuanto mayor sea la evaluación $B(s)$ para la posición s, mejor es la posición s para el jugador `Max` y peor es para su oponente `Min`. `Max` intenta maximizar la evaluación de sus movimientos, mientras que `Min` hace movimientos que resultan en una evaluación lo más baja posible.

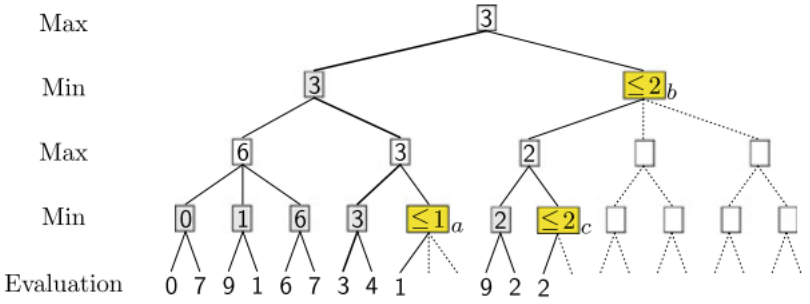
En la siguiente figura se muestra un árbol de búsqueda con cuatro medios movimientos y evaluaciones de todas las hojas. La evaluación de un nodo interno se deriva de forma recursiva como el máximo o mínimo de sus nodos hijos, según el nivel del nodo.



1.2 Poda alfa-beta

Alternar entre maximización y minimización, puede ahorrarnos mucho trabajo en algunas circunstancias. La poda alfa-beta funciona con el DFS hasta un límite de profundidad preestablecido. De esta forma, se busca en el árbol de búsqueda de izquierda a derecha. Al igual que en la búsqueda minimax, en los nodos mínimos, el mínimo se genera a partir del valor mínimo de los nodos sucesores y en los nodos máximos igualmente el máximo.

En la figura se representa este proceso para el árbol de la figura anterior:



En el nodo marcado a, todos los demás sucesores pueden ignorarse después de que el primer hijo se evalúe como el valor 1 porque el mínimo seguramente será ≤ 1. Incluso podría volverse más pequeño aún, pero eso es irrelevante ya que el máximo es ≥ 3 un nivel arriba. Independientemente de cómo resulte la evaluación de los sucesores restantes, el máximo mantendrá el valor 3. De manera análoga, el árbol se recortará en el nodo b.

Dado que el primer hijo de b tiene el valor 2, el mínimo que se generará para b solo puede ser menor o igual a 2. Pero el máximo en el nodo raíz seguramente será ≥ 3. Esto no se puede cambiar con valores ≤ 2. Por lo tanto, los subárboles restantes de b pueden podarse.

El mismo razonamiento se aplica al nodo c. Sin embargo, el nodo máximo relevante no es el padre directo, sino el nodo raíz. Este escenario se puede generalizar.

- En cada nodo hoja se calcula la evaluación.
- Para cada nodo máximo, el valor actual del hijo más grande se guarda en un archivo α.
- Para cada nodo mínimo, el valor actual del hijo más pequeño se guarda en β.
- Si en un nodo mínimo k el valor actual es $\beta \leq \alpha$, entonces la búsqueda bajo k puede terminar. Aquí α es el valor más grande de un nodo máximo en la ruta desde la raíz hasta k.
- Si en un nodo máximo l el valor actual es $\alpha \geq \beta$, entonces la búsqueda bajo l puede terminar. Aquí β es el valor más pequeño de un nodo mínimo en la ruta desde la raíz hasta l.

El algoritmo siguiente es una extensión del DFS con dos funciones que se llaman en alternancia. Utiliza los valores definidos anteriormente para α y β.

```
ALPHABETAMAX(Nodo,a,b)
If DepthLimitReached(Nodo) Return(Rating(Nodo))
NuevosNodos = Successors(Nodo)
While NuevosNodos             // distinto del vacio
    a = Maximum(a, ALPHABETAMIN(First(NuevosNodos), a,b))
    If a>b or a= b Return(b)
    NuevosNodos = Rest(NuevosNodos)
Return(a)

ALPHABETAMIN(Node,a,b)
If DepthLimitReached(Nodo) Return(Rating(Nodo))
NuevosNodos = Successors(Nodo)
While NuevosNodos             // distinto del vacio
    b = Minimum(b, ALPHABETAMAX(First(NuevosNodos), a,b))
    If b < a or b =a  Return(a)
    NuevosNodos = Rest(NuevosNodos)
Return(b)
```

La llamada de poda alfa-beta inicial se realiza con el comando ALPHABETAMAX(NodoRaiz, $-\infty, \infty$).

1.3 Complejidad

El tiempo de cálculo que se ahorra en la poda alfa-beta depende en gran medida en el orden en que se atraviesan los nodos hijos. En el peor de los casos, la poda alfa-beta no ofrece ninguna ventaja. Para un factor de ramificación constante b el número n_d de nodos de hojas para evaluar a la profundidad d es igual a:

$$n_d = b^d$$

En el mejor de los casos, cuando los sucesores de los nodos máximos se ordenan de forma descendente y los sucesores de los nodos mínimos se ordenan de forma ascendente, el factor de ramificación efectivo se reduce a $\sqrt{b}$. En el ajedrez, esto significa una reducción sustancial del factor de ramificación efectivo de 35 a aproximadamente 6 . Entonces solo

$$n_d = \sqrt{b}^d = b^{d/2}.$$

nodos de hoja se crearían. Esto significa que el límite de profundidad y por lo tanto, también el horizonte de búsqueda se duplican con la poda alfa-beta. Sin embargo, esto solo es cierto en el caso de sucesores ordenados de manera óptima porque las calificaciones de los nodos hijos se desconocen en el momento en que se crean. Si los nodos hijos se ordenan aleatoriamente, entonces el factor de ramificación se reduce a $b^{3/4}$ y el número de nodos hoja a:

$$n_d = b^{\frac{3}{4}d}.$$

Con la misma potencia de cálculo, una computadora de ajedrez que usa poda alfa-beta puede, por ejemplo, calcula ocho medios movimientos hacia adelante en lugar de seis, con un factor de ramificación efectivo de aproximadamente 14.

Para duplicar la profundidad de búsqueda como se mencionó anteriormente, necesitaríamos que los nodos hijos estén ordenados de manera óptima, lo que no es el caso en la práctica. De lo contrario, la búsqueda sería innecesaria. Con un simple truco podemos conseguir un orden de nodos relativamente bueno. Conectamos la poda alfa-beta con la profundización iterativa por encima del límite de profundidad. Por lo tanto, en cada nuevo límite de profundidad, podemos acceder a las calificaciones de todos los nodos de niveles anteriores y ordenar los sucesores en cada rama. De este modo alcanzamos un factor de ramificación efectivo de aproximadamente 7 a 8, que no está lejos del óptimo teórico de $\sqrt{35}$.

1.4 Juegos no deterministas

La búsqueda Minimax se puede generalizar a todos los juegos con acciones no deterministas, como el backgammon. Cada jugador lanza antes de su movimiento, que está influenciado por el resultado del lanzamiento de dados. En el árbol del juego ahora hay tres tipos de niveles en la secuencia.

```
Max, dados, Min, dados,... ,
```

donde cada nodo de lanzamiento de dados se ramifica de seis maneras. Debido a que no podemos predecir el valor del dado, promediamos los valores de todos los lanzamientos y realizamos la búsqueda como se describe con los valores promedio como lo indica el libro de S. Russell y P. Norvig. Artificial Intelligence: A Modern Approach.

2 Funciones de evaluación heurística

¿Cómo encontramos una buena función de evaluación heurística para la tarea de búsqueda? Aquí hay fundamentalmente dos enfoques. La forma clásica utiliza el conocimiento de expertos humanos. Al ingeniero del conocimiento se le asigna la tarea normalmente difícil de formalizar el conocimiento implícito del experto en forma de programa de computadora. Ahora queremos mostrar cómo se puede simplificar este proceso en el ejemplo del programa de ajedrez.

En el primer paso, se consulta a los expertos sobre los factores más importantes en la selección de un movimiento. Luego se intenta cuantificar estos factores. Obtenemos una lista de características o atributos relevantes. A continuación, se combinan (en el caso más simple) en una función de evaluación lineal $B(s)$ para posiciones, que podría verse así:

```
B(s) = a1.material + a2.pawn_structure +a3.king_safety +
          a4.knight_in_center + a5. bishop_diagonal_coverage + ... (1)

donde 'material' es la característica más importante y se calcula como:

  material = material (own_team) - material (opponent)

con

  material(team) = num_pawns(team).100 + num_knights(team).300
        + num_bishops(team).300 + num_rooks(team).500
        + num_queens(team).900
```

Casi todos los programas de ajedrez hacen una evaluación similar del material. Sin embargo, existen grandes diferencias para todas las demás características como se indica aquí: `http://www.frayn.net/beowulf/theory.html`.

En el siguiente paso, se deben determinar los pesos a_i de todas las características. Estas se establecen intuitivamente después de una discusión con expertos, luego se cambian después de cada juego en función de experiencias positivas y negativas. El hecho de que este proceso de optimización sea muy caro y además que la combinación lineal de funciones sea muy limitada sugiere el uso del aprendizaje automático.

2.1 Aprendizaje de heurística

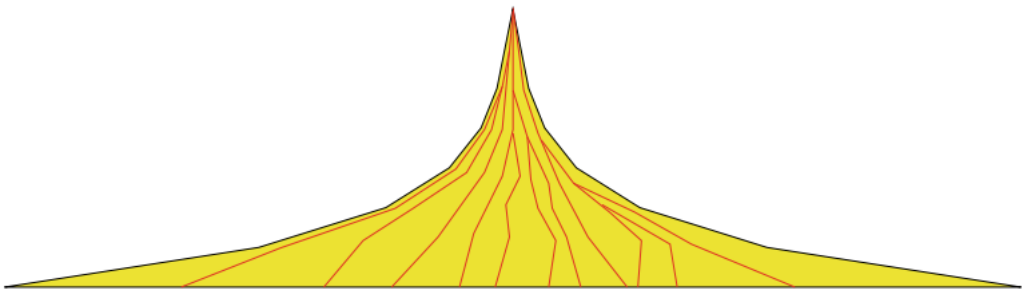
Ahora queremos optimizar automáticamente los pesos a_i de la función de evaluación $B(s)$. En este enfoque, solo se le pregunta al experto sobre las características relevantes $f_1(s), \ldots, f_n(s)$ para los estados del juego s. Luego, se utiliza un proceso de aprendizaje automático con el objetivo de encontrar una función de evaluación que sea lo más cercana a la óptima posible.

Nosotros comenzamos con una función de evaluación inicial preestablecida (determinada por el proceso de aprendizaje) y luego dejamos que el programa de ajedrez juegue. Al final del juego, se deriva una calificación del resultado (victoria, derrota o empate). Con base en esta calificación, la función de

evaluación se cambia con el objetivo de cometer menos errores la próxima vez. En principio, la misma tarea que hacía el desarrollador, ahora se realiza automáticamente por el proceso de aprendizaje.

Esto es muy difícil en la práctica. Un problema central con la mejora de la calificación de la posición basada en partidos ganados o perdidos se conoce hoy como problema de asignación de crédito. De hecho, tenemos una calificación al final del juego, pero no para los movimientos individuales. Por lo tanto, el agente realiza muchas acciones pero no recibe ninguna retroalimentación positiva o negativa hasta el final. ¿Cómo debería asignar esta retroalimentación a las muchas acciones tomadas en el pasado?, ¿y cómo debería mejorar sus acciones en ese caso? El apasionante campo del aprendizaje por refuerzo se ocupa de estas preguntas.

La búsqueda de árbol de Monte Carlo (MCTS) funciona de manera bastante similar. Para mejorar la calificación heurística de los estados de un juego s, se explora y se evalúa un número aleatorio de ramas del árbol de búsqueda a partir de este estado, o se detiene a una cierta profundidad y luego los nodos hoja se evalúan heurísticamente. La evaluación $B(s)$ del estado s se da como la media de todas las puntuaciones de los nodos hoja. El uso de rutas MCTS requiere que se busque solo una pequeña parte de todo el árbol se dispare xponencialmente. Esto se ilustra en la siguiente figura:



En el gráfico de un árbol de búsqueda, se muestran en rojo varias rutas MCTS a los nodos hoja. Debes observar que solo se busca en una pequeña parte del árbol.

Para muchos juegos simulados por computadora, como el ajedrez, este algoritmo se puede usar para lograr un mejor juego para el mismo esfuerzo computacional o para reducir el esfuerzo computacional para el mismo nivel de dificultad. Este método fue utilizado junto con algoritmos de aprendizaje automático en 2016 por el programa AlphaGo, que fue el primer programa de Go en derrotar a jugadores humanos de clase mundial.