Método de busqueda de Monte Carlo (UPC) aplicado al ajedrez vikingo

Pedro Luis Soto Santos
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
pedsotsan@alum.us.es)

Juan Pablo Cabezas Villalba
Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, España
juacabvil@alum.us.es)

***Resumen*—El objetivo principal del trabajo es la investigación e implementación del algorítmo de busqueda en arboles Monte Carlo UPC (Upper Confident Tree) a un juego de información perfecta, el Hnefatafl y sus variantes, de forma que sea posible jugar al juego contra una inteligencia artificial.**

***Palabras clave*—Monte Carlo, información perfecta, Hnefatafl, inteligencia artificial**

I. Introducción

En el siguiente documento se recogerá la información precisa para entender la naturaleza de esta tarea así como una documentación del proceso de aprendizaje según progresamos en ella.

El objetivo es esencialmente conseguir un programa mediante el cual un usuario pueda enfrentarse a una inteligencia artificial seleccionando la cantidad de tiempo (u otra condición de parada) que quiere dejar pensar a la misma, en un juego conocido como Hnefatafl o ajedrez vikingo.

Para comprender de qué trata éste trabajo es preciso comprender qué es el algoritmo de busqueda Monte Carlo, la variante Upper Confident Tree, así cómo el funcionamiento del propio juego, el ajedrez vikingo o Hnefatafl. Estas ideas y conceptos serán repetidos y definidos más adelante de forma que cualquiera pueda seguir el documento.

En primer lugar explicaremos el juego. El Hnefatafl consiste en un juego de tablero cuadriculado, similar al ajedrez moderno, en el cual 2 bandos luchan por un objetivo diferente. Comunmente el bando blanco conocido como los suecos trata de hacer escapar a su rey, mientras que el bando negro o moscovita trata de impedir este objetivo y acabar con los suecos. El juego puede resumirse en 6 reglas:

- Todas las fichas se mueven ortogonalmente cualquier cantidad de cuadros libres (el movimiento de la torre en el ajedrez). Las negras comenzarán el primer movimiento.
- Una ficha es capturada y sacada del tablero cuando el oponente ocupa ambos cuadros adyacentes en una fila o columna. Este método se denomina captura de los guardianes. Una ficha puede moverse en forma segura sobre un cuadro vacío entre dos fichas enemigas(no existe el suicidio).
- El rey es capturado de la misma forma excepto si se encuentra en el trono, si los cuatro cuadros alrededor de él son ocupados por fichas enemigas, o si es rodeado por tres lados por fichas enemigas y en la cuarta está el trono. Cuando el rey es capturado, se termina el juego y los moscovitas (negras) ganan.
- Los suecos (blancas) ganan si el rey llega a cualquier cuadro en la periferia del tablero (las esquinas).
- Si un bando no puede realizar nungún movimiento pierde.
- Si se supera el límite máximo de turnos se consideran tablas.

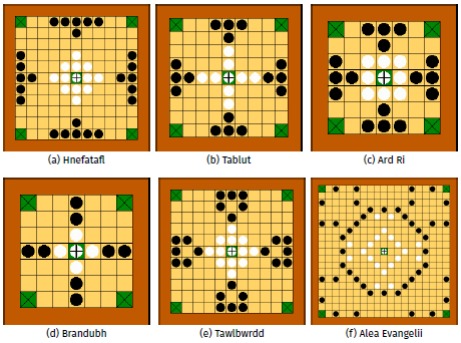

Fig. 1. Ejemplo de distribuciones de trablero. Imagen obtenida del enunciado de la práctica

Toda esa información escrita puede sonar abrumadora de golpe, por lo que recomendamos la visualización de un par de partidas cortas que a nosotros nos hizo comprender el juego de una manera amena y rápida [1].

Tal y como puede verse en *Fig. 1.* el juego puede comenzar desde distintas distribuciones las cuales estarán disponibles al comienzo de partida y variara el tamaño de tablero y piezas totales, haciendo más larga o corta la partida.

II. Preliminares

Existen numerosas implementaciones y algoritmos estudiados en la creación de inteligencia artificial o ”bots” en el ámbito de juegos. Los llamados juegos de información perfecta, es decir generalmente basados en turnos y donde los jugadores conocen el estado y posibilidades del juego además de no existir el azar se convierten en un genial blanco para probar estos algoritmos. En concreto, nosotros estudiamos la implementación del metodo de Monte Carlo.

El método de Montecarlo es un método de simulación que permite calcular estadísticamente el valor final de una secuencia de sucesos no deterministas (sujetos a variabilidad), como es el caso de una partida entre dos jugadores. Dentro del propio método de Monte Carlo existen adaptaciones, como Monte Carlo Tree Search, que es un planteamiento común para juegos de tablero, y en concreto una variante llamada Upper Confident Tree:

A. *Monte Carlo Tree Search*

En Monte Carlo Tree Search de manera convencional partiriamos de un estado origen y comenzaríamos a probar toda la combinatoria disponible con el fin de recaudar toda la información posible y a partir de la estadística obtenida analizar que solución nos concederá un mejor resultado. Es decir, al ser un método probabilístico no garantiza la optimización.

Encontramos las siguientes consecuencias:

- Sencilla implementación
- Coste de computación y tiempo muy elevado
- Cantidad de información de poca utilidad recaudada muy elevada

De estas taras nace la variante que trabajaremos:

B. *Variante Upper Confident Tree*

La variante Upper Confident Tree se especializa en ahorrar esfuerzo en muchas ramas que pudieran ser poco útiles y enfocandose en las ramas más prometedoras.

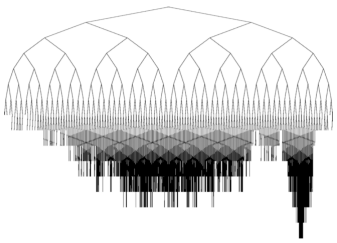

Fig. 2. Ejemplo de como se ve un árbol de montecarlo UPC [2]

En vez de hacer muchas simulaciones puramente aleatorias, esta variante hace muchas iteraciones de un proceso que consta de varias fases y que tiene como objetivo mejorar nuestra información del sistema (exploración) a la vez que potencia aquellas opciones más prometedoras (explotación).

C. *MinMaxing*

También a su vez las variantes son compatibles con la estrategia MinMax, en la cual se busca no sólo una solución para nuestro jugador, sino una solución que disminuya lo máximo posible las probabilidades de victoria del rival. UPC garantiza teóricamente la convergencia hacia el MinMax, en condiciones ideales de tiempo y presupuesto infinitos. [3]

III. METODOLOGÍA

A. *Resolución*

En primer lugar nos encontramos el primer desafío, conseguir un escenario en el cual aplicar la IA. Para ello necesitamos tableros, unas fichas y por encima de todo muchas reglas para hacer que todo funcione acorde al juego.

El primer objetivo fue conseguir un juego sin ningún tipo de IA, sólo jugador contra jugador y comprobar que era posible realizar todos los tipos de movimientos y capturas especiales que tiene este juego.

Probablemente la parte mas necesaria es conseguir obtener todos los movimiento ejecutables en cada turno, conseguir las consecuencias de dichos movimientos, ya sea una captura, el final de partida...

Las estructuras de datos se encuentran en el anexo Documentación del código, de forma que nos centraremos en describir la forma de encontrar la combinatoria de movimientos posibles para las piezas. La idea general del mismo es recorrer las direcciones ortogonales a la pieza en el tablero y comprobar si está disponible

obtiene_movimiento_peon $(tablero, jugador)$
Entrada: el tablero actual y el turno del jugador)
Salida: una lista con los posibles movimientos de $jugador$

```
1  movimientos ← lista vacía
2  numero_filas ←  longitud del tablero
3  movimiento ← movimiento vacío
4  Por cada fila :
5  si hay alguna pieza del jugador:
6      Por cada columna :
7      si casilla = jugador
8          newFila ← numero_filas - 1 - fila
9          izq ← columna-1
10         der ← columna+1
11         arr ← fila-1
12         abj ← fila-1
13         mientras izq > = 0:
14             si está vacía y no es esquina
15                 movimiento ← (columna,newFila,izq,newFila)
16                 movimiento añadir en movimientos
17             si izq > 0:
18                 izq ← izq-1
19             Parar si no
20         Parar si no
21 Continuar por cada dimensión restante...
```

Así pues, obteniendo la lista de movimientos completa que un jugador tiene en el tablero (método muy similar para el rey exceptuando las restricciones de casillas especiales), junto con una función que sea capaz de aplicarlos (pseudocódigo demasiado extenso para ser expuesto de forma precisa, de manera resumida):

Por último necesitamos comprobar si se acabó la partida:

aplica_movimiento $(estado, movimiento)$
Entrada: el estado actual y el movimiento a realizar)
Salida:el nuevo estado de la partida
Seleccionar fichas que corresponden al jugador y rival
Ajustar coordenadas de cartesianas a tablero
Realizar el movimiento en una copia del tablero:
Marcar origen vacío en la copia del tablero:
Mirar si existe pieza enemiga capturable
Comprobar:
    si la pieza a capturar esta en el centro**:**
        comprobar si está rodeada:
            centro = 0
    si la pieza a capturar esta próxima al centro**:**
        comprobar si es flanqueada por 3 lados
            casilla = 0
    si caso normal de captura o con esquinas
        comprobar si es flanqueada
Hacer cambio de turno en el estado

es_estado_final $(estado, numero\_de\_movimientos, numero\_t$
Entrada: el estado actual, y los turnos hechos y límites)
Salida:el nuevo estado de la partida
sinumero de turnos es 0 **:**
    devolver True
si ganan negras o blancas**:**
    devolver ganan negras o ganan blancas

ganan_negras $(estado, numero\_de\_movimientos)$
Entrada: el estado actual, y los turnos hechos)
Salida:el nuevo estado de la partida
$tablero \leftarrow estado$.tablero
$jugador \leftarrow estado$.jugador
$resultado \leftarrow$ True
si no quedan movimientos y el turno es blancas**:**
    devolver resultado
si no:
    si el rey sigue en el tablero**:**
        devolver False
El método ganan_blancas es muy similar
sólo que buscando el rey en las esquinas...

El juego puede resumirse en un bucle con condición de parada en final de partida dependiente de la funcion*es_estado_final*

Habiendo conseguido una jugabilidad completa, ahora pues el orden seguiría por abordar a fondo con la busqueda de la solución.

Entrando en detalle, con nuestro algoritmo tratamos de maximizar la siguiente fórmula:

$$Argmax\frac{Q(v')}{N(v'} + c\sqrt{\frac{2 * LnN(v)}{N(v')}} \quad (1)$$

Véase como el primer miembro ajusta la calidad de un hijo respecto al numero de veces que ha sido visitado, mientras que el segundo argumento, que es ajustable pondera en resumidas cuentas el factor de exploración.

El codigo propuesto en resumen es capaz de generar un árbol de nodos de forma que se trabaja el algoritmo UPC expandiendo un nivel inferior al nodo padre y explorando las ramas más prometedoras.

Para poder llevar a cabo la tarea hemos creado una clase con los siguientes atributos:

```
#Esta es la clase nodo la cual usaremos en el algoritmo
class nodo:
    def __init__(self,estado,padre,numero_turno):
        self.estado = estado
        self.movimientos = obtiene_movimientos(estado)
        self.n = 0
        self.q = 0
        self.i = 0
        self.indMov=0
        self.turno = numero_turno
        self.hijos = []
        self.padre = padre
```

Fig. 3. Clase nodo implementada en código

El atributo indMov es un añadido a lo propuesto para poder rescatar la ubicación dentro de la lista movimientos de un nodo padre el movimiento que conduce al nodo hijo.

El resto de métodos han sido implementados de manera similar a la propuesta de pseudo código dada por el profesor excepto cambios leves que hablaremos en conclusiones en respuesta a los resultados experimentados.

B. Extras

- En consecuencia a la forma de desarrollar la aplicación, de manera natural decidimos implementar la versión de jugador vs jugador, ya que fue una forma rápida de comprobar que el juego funcionaba de manera adecuada.
- Más tarde, y de la misma manera, la feature de sugerencia de movimiento fue una forma de ir revisando las decisiones del algoritmo en base al estado de la partida.
- Incluir el tercer modo de juego fue simplemente hacer pequeños cambios en lo ya construido y la manera más interesante de observar el progreso de la partida, además parametrizando a cada equipo para comprobar que en efecto un bando con más tiempo para pensar suele hacer movimientos más adecuados.

IV. Resultados

El primer resultado obtenido consistió en una primera versión del juego, sólo disponible para dos personas el cual fue probado a fondo para cerciorarnos de que la IA al probar miles de movimientos no encontraría ningún caso extraño.

Más tarde después de haber implementado *busca_solución*, obtuvimos una primera versión del juego humano vs IA con las siguientes características:

- Las referencias a memoria nos jugaron una muy mala pasada haciendo que en la primera iteración el nodo padre se sobreescribiese con el primer estado final encontrado en el bucle, de forma que la partida se lanzaba, la persona podía realizar un movimiento y automáticamente acababa el juego

Tras darnos cuenta de lo ocurrido, procedimos a realizar copias de estados para evitar dicho suceso. Además, en el proceso de lanzar la IA pudimos observar pequeños detalles que se nos escaparon en *aplica_movimiento* que hacia que las capturas no fueran del todo correctas, sobre todo respecto al centro.

Los resultados obtenidos continuaban sin tener sentido, lo que nos llevó a descubrir errores en *default_policy*, *expand* y *backup* relacionados con las condiciones de parada. En este punto del proceso observamos las siguientes características:

- La IA realiza movimientos que, si se encuentran a poca distancia, midiendo en nodos, son muy precisos y adecuados en una mayoría de ocasiones, sin embargo con encontrarse a 3 turnos o más comenzaba a realizar movimientos principlamente aleatorios, lo que nos condujo a revisar *best_child* y *default_policy*

Por último, introdujimos cambios en las recompensas *delta* puesto que no consideramos que sea igualmente negativo empatar que perder. Además, en el cálculo de best_child inicializabamos la variable a sustituir por uno de los hijos en 0, de forma que al comparar un estado malo del tablero nunca se sustituía por el mejor hijo, y por tanto estabamos recibiendo movimientos aleatorios por culpa de la mezcla de movimientos en el cálculo. Otro cambio muy interesante fue el cambio de la variable *C*, es decir la constante de exploración ya que pudimos comprobar que en el caso aplicado con la constante recomendada la IA insistía demasiado en ramas que realmente no eran tan interesantes por culpa de no conocer más posibilidades, es decir, un exceso de explotación frente a exploración.

V. Conclusiones

Finalmente, y tras mucha experimentación, cambios y revisiones de código llegamos a las siguientes conclusiones respecto al proyecto:

- El algoritmo se ve muy afectado por el tiempo que tiene disponible, haciendo que sin mucho margen devuelva soluciones prácticamente aleatorias. Es decir, es un algoritmo muy costoso.
- Ante una partida perdida, el algoritmo propuesto intenta alargar la partida lo máximo posible buscando un error en su adversario.
- Hemos comprobado por la manera de jugar de la IA contra la de un humano que la primera actua con un foco mucho más alejado de los movimientos inminentes. Una persona aunque no fuera capaz de ver la partida a larga distancia probablemente apostaría por eliminar piezas y conservar las propias sin embargo observamos como curiosamente nuestra implementación apuesta por elminar piezas del camino incluso si es a pesar de perder fichas propias con objetivo de limpiar más rapido un camino posible, es decir se nota la falta de una recompensa por pieza comida.
- Se echa en falta algún tipo de heurística que apoye las decisiones más que simplemente encontrar un estado final interesante y explotarlo en mayor o menor medida. A no ser que dispongamos de un tiempo inasumible para una partida convencional, estamos propensos a realizar movimientos que podrían ser poco naturales.

Respecto a propuestas de mejora u objetivos a cumplir en el futuro probablemente optariamos por:

- Búsqueda del parámetro de exploración más adecuado para cada juego.
- Encontrar alguna forma de recompensar posiciones estratégicas o numero de piezas generando un híbrido entre Montecarlo y aprendizaje por refuerzo.
- Implementar una interfaz gráfica para mayor usabilidad de la implementación.

VI. Anexo

1) ajedrez_vikingo (enunciado).
2) manual de usuario.
3) documentación del código.

Referencias

[1] https://youtu.be/AHk85vLmeos
Partida de Hnefatafl del usuario "Conversa De Taverna" (en portugués, aunque no es realmente importante) en la cual se puede observar las reglas del juego en la práctica y una victoria de cada bando.

[2] C.-W. Chou, O. Teytaud, and S.-J. Yen, "Revisiting Monte-Carlo tree search on a normal form game: NoGo," in Proc. Appl. Evol. Comput., 2011, pp. 73–82

[3] https://www.chessprogramming.org/UCT.
Fuente usada durante todo el documento