Monte Carlo aplicado al ajedrez vikingo

Propuesta de trabajo para la asignatura Inteligencia Artificial

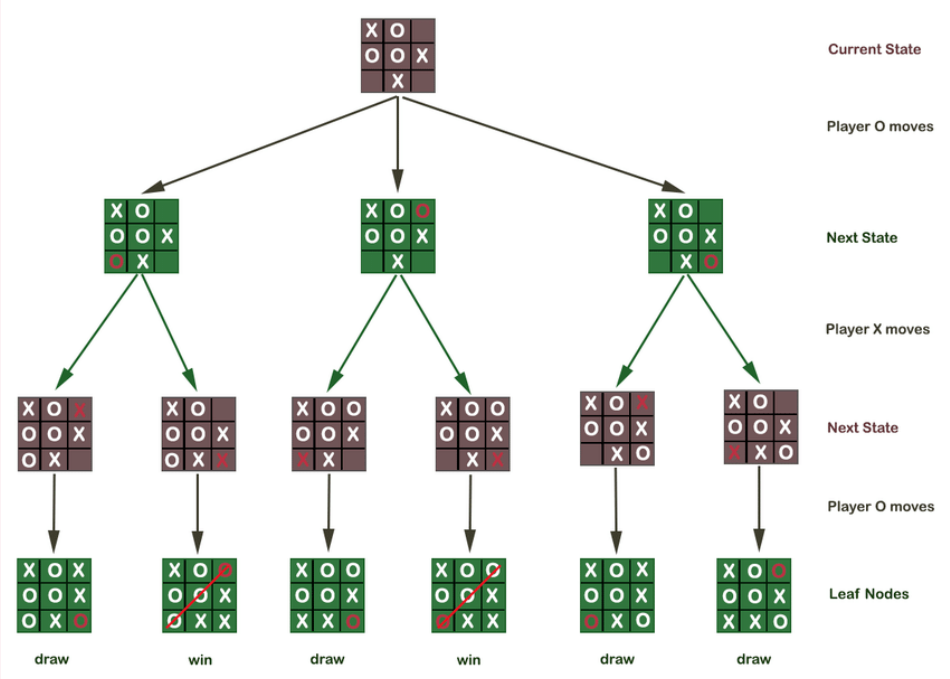
Profesor: Ignacio Pérez Hurtado de Mendoza

Grado en Ingeniería Informática - Ingeniería del Software
Curso 2021/2022

Introducción y objetivo

Los juegos en los que participan dos jugadores compitiendo entre ellos se conocen como *juegos con adversario*. La programación de este tipo de juegos es una interesante aplicación de la Inteligencia Artificial hoy en día. Para ello, es habitual usar árboles de búsqueda en donde cada nodo representa un estado del juego y cada arista representa una posible jugada del jugador al que le toca jugar.

Ejemplo: Árbol de búsqueda en el juego del tres en raya



La idea del algoritmo *Minimax* y sus variantes es explorar este tipo de árboles considerando que cada jugador va a elegir un movimiento que maximiza sus posibilidades

de ganar y minimiza las posibilidades de ganar del contrincante. En [3] y en [4], se puede encontrar buenas introducciones a esta categoría de algoritmos.

No obstante, este tipo de algoritmos requiere de un método para estimar la ventaja de cada jugador para cada estado del juego. Este método, conocido como *heurística*, implica un conocimiento avanzado sobre el juego. Por otra parte, el alto factor de ramificación del árbol de búsqueda en algunos juegos, imposibilita obtener buenas jugadas en un tiempo razonable.

Una aproximación relativamente reciente que supera las anteriores dificultades es aplicar un algoritmo de *Monte Carlo Tree Search* (MCTS), lo más sorprendente de este tipo de algoritmos es que el único conocimiento necesario sobre el juego son sus reglas. Dicho de otra forma, una persona que no sepa como ganar a un juego, pero que conozca sus reglas, puede programar una IA competitiva para dicho juego. En [1] y [2] se puede encontrar más información sobre los algoritmos MCTS.

El **objetivo** de este trabajo es implementar el juego del *ajedrez vikingo* usando la técnica MCTS, en particular, usando el algoritmo Upper Confident Tree (UCT). Para ello, se presentan en este documento algunas directrices, tales como una serie de estructuras de datos, una API de funciones y el pseudocódigo del algoritmo UCT, aunque estas indicaciones sólo se deben considerar como recomendaciones generales y el alumnado es libre de tomar decisiones de diseño. Es importante aportar una implementación propia del algoritmo UCT y de las estructuras de datos para el juego. Es decir, no se permite delegar en bibliotecas existentes sobre MCTS. El programa desarrollado se podrá realizar en Python, Java o C/C++ a elección del alumnado y deberá contar con un interfaz para jugar contra el usuario. La realización de implementaciones eficientes y mejoras del algoritmo propuesto serán valoradas. Para cualquier duda o consulta, pónganse en contacto con el profesor.

En las siguientes secciones se detallan el juego del ajedrez vikingo y el algoritmo MCTS. También se describe el trabajo a realizar, se dan algunas directrices sobre su implementación y se enumeran los criterios de evaluación.

El Ajedrez vikingo

El **Hnefatafl** o comúnmente conocido como *ajedrez vikingo* es un antiguo juego de mesa germánico perteneciente a la familia de los juegos *Tafl* que se practicaba sobre una tabla cuadriculada. De hecho, la palabra *tafl* significa mesa o tablero en nórdico antiguo y actualmente es la palabra para *ajedrez* en islandés moderno. Se trata de un juego con las siguientes características generales:

- Es *determinista*: El efecto de cada acción es perfectamente conocido, no hay influencia del azar.
- De *dos jugadores* que juegan *por turnos*.
- Sólo existen tres resultados posibles: que gane un jugador, que gane el otro o que se produzcan tablas. Es, por tanto, un juego de *suma nula*.
- De *información perfecta*: Cada jugador tiene en todo momento información completa sobre el estado del juego.

- Es *asimétrico*: Cada jugador tiene unas reglas y unos objetivos diferentes para ganar.

Ejemplo: Tablero, fichas y disposición en el Hnefatafl

El tablero consiste en una cuadrícula en dónde van colocadas las piezas de los jugadores, existen tableros de diferentes tamaños. El objetivo de las blancas es proteger al rey (pieza central en la disposición inicial) hasta que consiga llegar a una casilla de fuga (cualquiera de las cuatro esquinas del tablero). El objetivo de las negras es capturar al rey blanco antes de que escape.

Piezas

Un jugador elije las blancas y otro las negras. Existen piezas de dos tipos:

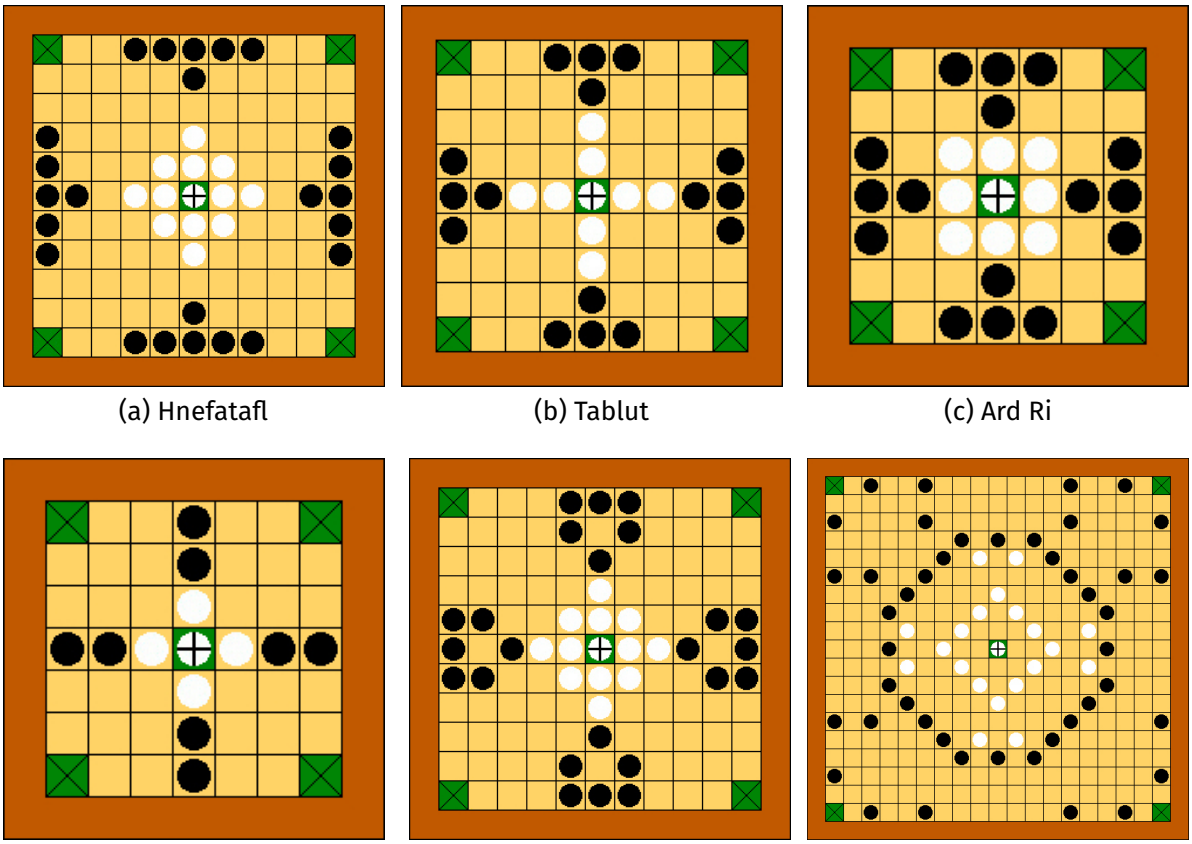
- **El rey** es la pieza más importante de las blancas. Generalmente se diferencia porque es la figura blanca más grande y mejor adornada.
- **Los peones** son el resto de las piezas. Existen peones blancos y negros. Cabe destacar que el jugador que juega con las negras sólo cuenta con peones, pero el que juega con blancas cuenta además con el rey.

De manera genérica, denominaremos *pieza* a un peón (blanco o negro) o al rey.

Disposición

El rey va inicialmente en la casilla central del tablero, que se denomina *trono*, rodeado por los peones blancos. Los peones negros se disponen en torno a los bordes del

tablero, existen diferentes disposiciones iniciales y tamaños de tablero que se detallan a continuación.



(a) Hnefatafl (b) Tablut (c) Ard Ri

(d) Brandubh (e) Tawlbwrdd (f) Alea Evangelii

Disposiciones iniciales y tamaños de tablero en el ajedrez vikingo

Reglas

Como cabe suponer después de tantos siglos, es natural que las reglas no se conozcan con precisión. Existen algunas discrepancias sobre las normas del ajedrez vikingo, pero el reglamento que mejor conduce a unas partidas equilibradas es el que sigue. Las reglas son las mismas para todas las disposiciones iniciales y tamaños del tablero.

Un jugador controla al bando atacante (negras), compuesto de peones, y su adversario maneja al bando defensor (blancas), que además de peones cuenta con el rey. El objetivo de las negras es capturar al rey, mientras que las blancas tratarán de hacer que llegue a una de las cuatro esquinas del tablero y escape.

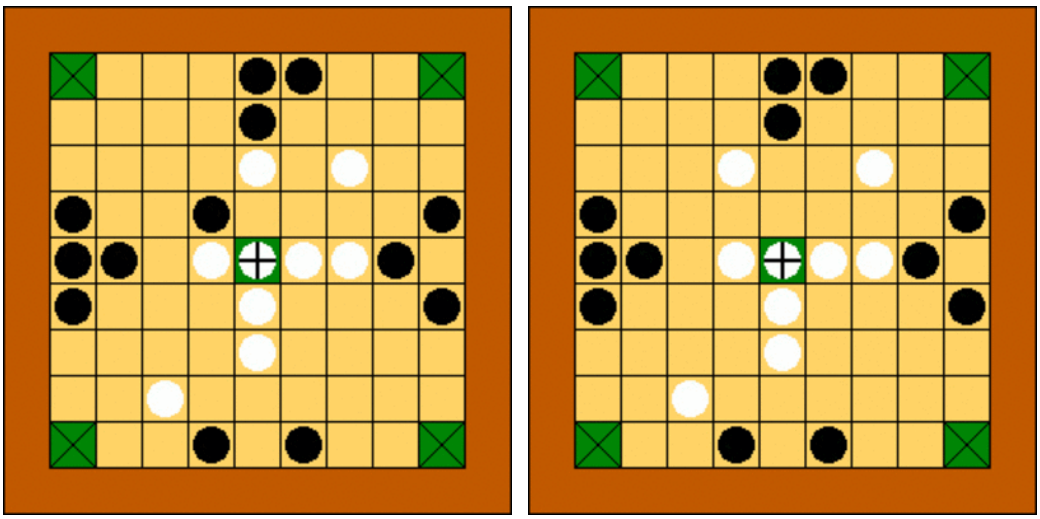
Los jugadores mueven una pieza por turno, empezando la partida el bando negro. En el tablero hay cinco casillas especiales que sólo pueden ser ocupadas por el rey: las cuatro esquinas por las que puede huir y el trono o casilla central. Los peones pueden pasar a través del trono siempre y cuando esté desocupado (pero no terminar su movimiento en el trono, al igual que no pueden terminar su movimiento en una esquina del tablero).

El movimiento

Todas las piezas del juego se mueven ortogonalmente a través de casillas vacías tal como las torres en el ajedrez. No se puede, por tanto, saltar por encima de otras piezas ni que varias piezas ocupen una misma casilla.

La captura

Una pieza es eliminada cuando dos piezas rivales la atrapan entre sí, ya sea en horizontal o vertical. Ahora bien, las capturas sólo se efectúan cuando ha sido el atacante el que encierra a su rival, y no cuando éste coloca una de sus piezas entre dos piezas del contrincante. Dicho de otra forma, no existe el suicidio.



(a) Antes (b) Después

Captura de un peón negro

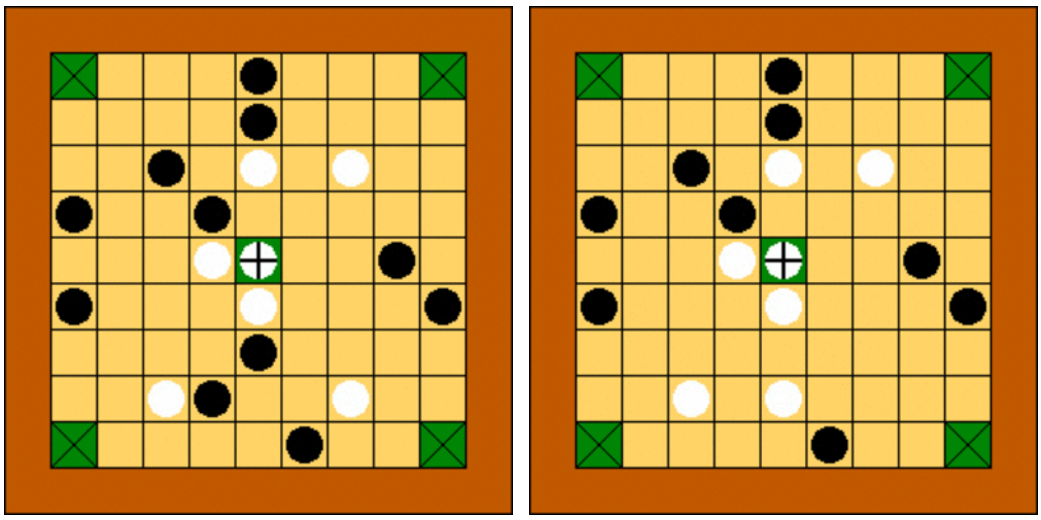
Para capturar a una pieza también se puede utilizar el trono (esté ocupado o no) y las casillas de fuga, que ejercen como una pieza aliada para hacer la captura. La acción de arrinconar una pieza contra el borde del tablero, por contra, queda sin efecto. Cabe decir que en un mismo movimiento se podrían capturar varias piezas enemigas.

El rey blanco captura y es capturado de la misma manera que los peones, con dos excepciones:

- Si se encuentra en el trono habrá de ser rodeado por los cuatro lados.
- Si se encuentra en una casilla adyacente al trono, habrá de ser rodeado por los otros tres lados (los que no son el trono).

Cómo ganar

Las blancas vencerán si el rey consigue alcanzar una de las cuatro esquinas. Por otra parte, las negras ganarán si logran capturar al rey. Si un jugador se queda sin movimientos y le toca jugar, pierde automáticamente.



(a) Antes (b) Después

Captura de dos peones negros

Tablas por repetición

Al igual que en el ajedrez, se pueden producir tablas por repetición. Es un tipo de empate que puede suceder debido a la regla de triple repetición de posiciones. Esta regla se aplica para evitar que las partidas continúen para siempre.

La regla de triple repetición dice que si una posición surge tres veces en una partida, la partida queda en tablas.

Es importante resaltar que las posiciones repetidas no necesitan repetirse consecutivamente. No importa en que momento de la partida sucede la posición repetida, a la tercera vez que suceda se declarará tablas.

No obstante, en este trabajo no se pide la implementación de tablas por repetición. Será un extra que se podrá implementar de forma opcional (ver criterios de evaluación más adelante).

Tablas por número máximo de jugadas

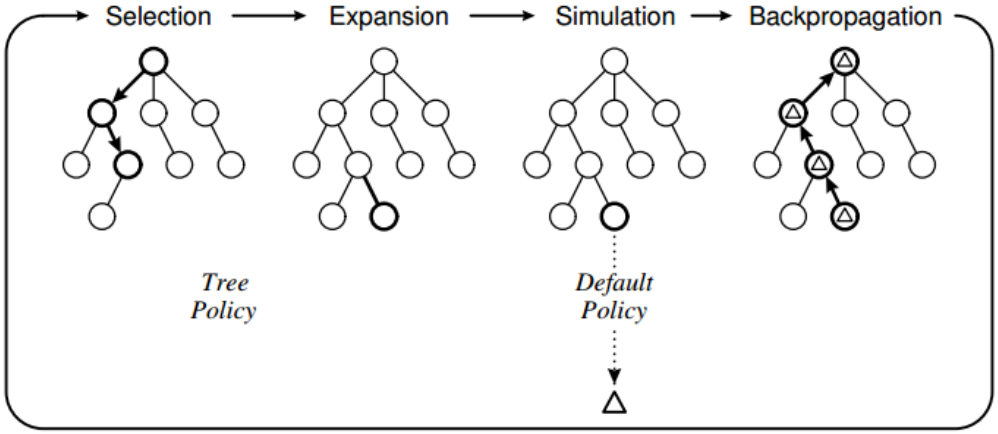
Los jugadores pueden acordar una partida con un número máximo de jugadas, llegando a tablas si se supera dicho número de jugadas. Esta será la condición de tablas que se utilizará en este trabajo.

Algoritmo MCTS

En general, en un algoritmo de Monte Carlo aplicado a un juego:

- Se generan simulaciones aleatorias desde la posición actual del tablero.
- Se almacenan estadísticas para cada posible movimiento.
- Se devuelve el movimiento con los mejores resultados.

El problema de este planteamiento general es que pueden haber pocos movimientos realmente buenos para una situación específica de un juego y puede ser muy difícil encontrarlos por puro azar. Para solucionar este problema, se plantea la aproximación de Monte Carlo Tree Search (MCTS) y, en particular, la variante Upper Confident Tree (UCT), que será la que se implementará en este trabajo. En esta variante, en lugar de hacer muchas simulaciones puramente aleatorias, se hace una gran cantidad de iteraciones de un proceso que consta de varias fases y que tiene como objetivo mejorar la información del sistema (exploración) a la vez que potencia aquellas opciones más prometedoras (explotación):



Fases de una iteración del algoritmo MCTS

En un algoritmo MCTS, se utiliza un árbol de búsqueda cuyos nodos representan estados del juego y cuyas aristas representan acciones. El algoritmo, a grandes rasgos, consta de las siguientes fases:

- Selección: En esta fase se selecciona el nodo del árbol que será explorado.
- Expansión: El nodo seleccionado se expande con un nuevo hijo.
- Simulación: Se realiza una simulación de juego desde el estado expandido hasta alcanzar un estado final, entonces se obtiene un valor que para un juego de suma cero podría ser 1, -1 o 0. Es decir, partida ganada, perdida o tablas.
- Retropropagación: Se asigna al estado expandido el valor obtenido de la simulación y se propaga hacia la raíz actualizando los valores estadísticos de cada nodo, que servirán para la fase de selección de las siguientes iteraciones del algoritmo.

Se recomienda encarecidamente leer la referencia [1], especialmente centrándose en el algoritmo UCT para dos jugadores, que es la propuesta en este trabajo.

Descripción del trabajo

El núcleo del trabajo consistirá en la implementación de un programa en Python, Java o C/C++ que permita a un usuario jugar contra el ordenador al juego del ajedrez vikingo. El

programa deberá implementar la variante UCT del algoritmo MCTS para que el ordenador pueda realizar jugadas. Se deberá escribir una documentación que incluya la explicación de los algoritmos y técnicas utilizadas, así como una documentación mínima del programa realizado.

El código fuente también deberá ser entregado, se valorará el buen estilo, la corrección, eficiencia y comentarios incluidos.

Finalmente, habrá que preparar una presentación del trabajo en la que todos los miembros del grupo deberán participar activamente y responder a las preguntas que sean planteadas por el profesor.

Desarrollo de software

Tal cómo se ha comentado, se permite realizar el código en Python, C/C++ o Java. Dejando la elección a los alumnos, de acuerdo a sus conocimientos y preferencias. A continuación, se dan algunas indicaciones de estructuras de datos considerando Python y se propone una API a implementar. Aunque las decisiones finales se dejan al criterio de los alumnos.

Estructuras de datos

Se utilizarán las siguientes estructuras de datos:

- **Tablero de juego**: Estructura de datos que representa un tablero. Se trata de una estructura que debe representar una matriz de números de N filas y M columnas. En donde los números representan lo siguiente: el 0 significa una casilla vacía, el 1 significa una casilla ocupada por un peón negro, el 2 significa una casilla ocupada por un peón blanco y el 3 significa una casilla ocupada por el rey blanco. La forma de implementar esta matriz queda a decisión de los alumnos (se pueden usar, por ejemplo, listas de listas, tuplas de tuplas, arrays de numpy, matrices dispersas, representación binaria, una clase propia, etc.) Por ejemplo, usando tuplas de tuplas, el tablero inicial en la variante *Ard ri* vendría representado por la siguiente estructura: ((0,0,1,1,1,0,0), (0,0,0,1,0,0,0), (1,0,2,2,2,0,1), (1,1,2,3,2,1,1), (1,0,2,2,2,0,1), (0,0,0,1,0,0,0), (0,0,1,1,1,0,0)).
- **Estado del juego**: Tupla de dos elementos. El primero es un tablero, el segundo es un número que indica el jugador activo, es decir, el jugador al que le toca jugar, que puede ser 1 (negras) o 2 (blancas).
- **Movimiento**: Para representar un movimiento, se utilizará una tupla de 4 números que representan respectivamente el índice de la fila y la columna de inicio del movimiento y la fila y la columna del final del movimiento. Todos los índices comienzan en cero. Por ejemplo, para el tablero anterior, los dos únicos movimientos del peón negro que se encuentra en la posición (0,4) serían los siguientes: (0,4,0,5) y (0,4,1,4).

API a implementar

Considerando las estructuras de datos descritas, se propone la implementación de las siguientes funciones:

- **obtiene_estado_inicial(variante)**: Función que recibe un número entre el 1 y el 6 y obtiene el estado inicial de la correspondiente variante del juego, tal como se enumera a continuación:

  1. Hnefatafl
  2. Tablut
  3. Ard Ri
  4. Brandubh
  5. Tawlbwrdd
  6. Alea Evangelii

  En un estado inicial, el jugador activo es 1 (negras).

- **obtiene_movimientos(estado)**: Función que recibe un estado y devuelve una lista con todos los posibles movimientos del jugador activo. Esta lista deberá devolverse ”barajada” tras aplicar la función random.shuffle.

- **ganan_negras(estado,número_movimientos)**: Función que recibe un estado y el número de posibles movimientos del jugador activo y devuelve TRUE si las negras han ganado, lo cual ocurre en una de las dos siguientes situaciones:

  - El rey no se encuentra en el tablero (ha sido capturado).
  - El jugador activo es 2 (blancas) y el número de movimientos es 0.

- **ganan_blancas(estado,número_movimientos)**: Función que recibe un estado y el número de posibles movimientos del jugador activo y devuelve TRUE si las blancas han ganado, lo cual ocurre en una de las dos siguientes situaciones:

  - El rey se encuentra en cualquiera de las cuatro esquinas (ha llegado al punto de fuga).
  - El jugador activo es 1 (negras) y el número de movimientos es 0.

- **es_estado_final(estado,número_movimientos)**: Función que recibe un estado y el número de posibles movimientos del jugador activo y devuelve TRUE si ganan negras o ganan blancas.

- **imprime_estado(estado,número_movimientos)**: Función que recibe un estado y el número de posibles movimientos del jugador activo e imprime el correspondiente tablero por pantalla de tal forma que el usuario pueda entenderlo. Se puede usar cualquier combinación de símbolos ASCII y saltos de línea. Se recomienda también mostrar los números de fila y columna del tablero. Si se trata de un estado final, indicar ”Ganan negras”, ”Ganan blancas” o ”Tablas”. En caso contrario, indicar a que jugador le toca jugar mediante el texto ”Juegan negras” o ”Juegan blancas”. Indicar también el número de posibles movimientos del jugador activo: ”Posibles movimientos: X”.

- **aplica_movimiento(estado,movimiento)**: Función que recibe un estado y un movimiento válido para el jugador activo y devuelve un nuevo estado tras aplicar dicho movimiento. En dicho nuevo estado, el jugador activo pasa a ser el otro jugador y cualquier pieza capturada es eliminada del tablero. El estado anterior no debe ser modificado. Ojo con los objetos mutables en Python.

- **busca_solución(estado,tiempo)**: Función que recibe un estado del juego y un número real que indica un tiempo de computación en segundos. La función devuelve el mejor movimiento a realizar por parte del jugador activo, mediante la aplicación del algoritmo UCT, cuyo pseudocódigo se describe más adelante.

- **interfaz_usuario()**: Función que implementa un programa de texto para jugar al ajedrez vikingo contra el ordenador, el programa deberá dar un mensaje de bienvenida y pedir al usuario que seleccione la variante del juego. Luego le preguntará si desea jugar con blancas o con negras y el tiempo de computación en segundos que dedicará la máquina a pensar cada jugada, lo cual determinará el nivel de dificultad. El programa imprimirá el estado actual y si es un estado final, el juego terminará. Si el jugador activo es el usuario, se le preguntará por el movimiento a realizar, que lo podrá indicar escribiendo los cuatro números del movimiento a realizar o mediante otro método a criterio de los alumnos. Si el movimiento no es válido, se indicará y se volverá a preguntar por un movimiento. Cuando le toque jugar al ordenador, se ejecutará la función *busca_solución(estado,tiempo)* y se imprimirá por pantalla el movimiento realizado por el ordenador. Luego se actualizará el estado y se repetirá el procedimiento en bucle hasta que acabe la partida.

Pseudocódigo del algoritmo UCT

A continuación se propone un pseudocódigo basado en [1] para implementar la función *busca_solución(estado,tiempo)*[1]. En este algoritmo se utilizará una estructura nodo para representar un nodo del árbol de búsqueda, que contendrá:

- Un estado del juego.
- La lista de movimientos del jugador activo.
- Un número real n.
- Un número real q.
- Un número entero i
- La lista de nodos hijos.
- El nodo padre.

[1]Se trata sólo de una propuesta, los alumnos son libres de implementar un pseudocódigo alternativo siempre y cuando se documente debidamente y siga el esquema general del algoritmo UCT

Algorithm 1 Función **busca_solución(s0,t)**

```
v0 ← crea_nodo(s0, None)
while Hay tiempo de computación do
    v1 ← tree_policy(v0)
    Δ ← default_policy(v1)
    backup(v1, Δ)
end while
Devolver v0.movimientos[best_child(v0, 0)]
```

Algorithm 2 Función **crea_nodo(s,padre)**

```
v ← nodo_vacío()
v.estado ← s
v.movimientos ← obtiene_movimientos(s)
v.n ← 0
v.q ← 0
v.i ← 0
v.hijos ← lista vacía.
v.padre ← padre.
Devuelve v
```

Algorithm 3 Función **tree_policy(v)**

```
while not es_estado_final(v.estado,|v.movimientos|) do
    if v.i < |v.movimientos| then
        Devolver expand(v)
    else
        v ← v.hijos[best_child(v, C_p)]        ▷ Se recomienda C_p = 1/√2
    end if
end while
```

Documentación

No hay límite mínimo ni máximo en la documentación a entregar, aunque se recomienda 6-12 páginas. Se valorará la utilización del sistema LaTeX [5] y se recomienda seguir una plantilla de artículo científico, tal cómo el formato *IEEE Conference Proceedings* [6], aunque no es obligatorio. En general, se recomiendan las siguientes secciones:

- Resumen o *Abstract*
- Introducción
- Preliminares
- Desarrollo realizado
- Pruebas y experimentación
- Conclusiones

Algorithm 4 Función **expand(v)**

```
s ← aplica_movimiento(v.estado, v.movimientos[v.i])
v.i ← v.i + 1
hijo ← crea_nodo(s, v)
Añadir hijo al final de v.hijos
Devolver hijo
```

Algorithm 5 Función **best_child(v,c)**

Devolver $\arg\max_{i \in [0,|v.hijos|)} \frac{v.hijos[i].q}{v.hijos[i].n} + c\sqrt{\frac{2\log(v.n)}{v.hijos[i].n}}$

Algorithm 6 Función **default_policy(v)**

```
s ← v.estado
movs ← v.movimientos
jugador ← v.padre.jugador
while not es_estado_final(s,|movs|) do
    a ← movimiento elegido de forma aleatoria uniforme en movs
    s ← aplica_movimiento(s, a)
    movs ← obtiene_movimientos(s)
end while
if ganan_blancas(s,|movs|) y jugador == blancas then
    Devuelve 1
else if ganan_negras(s,|movs|) y jugador == negras then
    Devuelve 1
else
    Devuelve −1
end if
```

- Bibliografía
- Anexo 1: Documentación del código
- Anexo 2: Manual de usuario

Se permite la entrega en castellano o en inglés.

Presentación y defensa

Como parte de la evaluación del trabajo se deberá realizar una defensa del mismo, para lo que se citarán a los alumnos de manera conveniente. El día de la defensa se deberá realizar una pequeña presentación (PDF, PowerPoint o similar) de 10-15 minutos en la que deberán participar activamente todos los miembros del grupo. Cada miembro debe hablar por igual. Esta presentación deberá seguir a grandes rasgos la misma estructura que la memoria del trabajo, haciendo especial mención a los resultados obtenidos y al análisis crítico de los mismos. En los siguientes 10 minutos, el profesor procederá a hacer

Algorithm 7 Función **Backup(**v,Δ**)**

```
while v sea distinto de None do
    v.n ← v.n + 1
    v.q ← v.q + Δ
    Δ ← −Δ
    v ← v.padre
end while
```

preguntas a cada uno de los miembros del grupo, que podrán ser preguntas sobre la memoria, el código realizado o cuestiones técnicas, teóricas o algorítmicas.

Mejoras opcionales

La exquisita implementación, documentación y presentación del trabajo descrito en las secciones anteriores es condición suficiente para obtener la máxima nota. No obstante, se plantean a continuación una serie de mejoras para subir nota. Para que estas mejoras sean valoradas, el algoritmo UCT ha tenido que ser implementado de forma completa y correcta. Es decir, no se puede pretender subir nota implementando un interfaz gráfico si el algoritmo UCT está mal implementado. Por tanto, estas mejoras sólo podrán suplir carencias en la documentación, presentación o código poco eficiente o mal documentado. Se recomienda hacer una copia de seguridad del trabajo antes de implementar las mejoras y entregarla también.

- Mostrar información por pantalla sobre el proceso de búsqueda, tal como número de nodos en el árbol, profundidad actual del árbol, etc.
- Implementar un interfaz gráfico. Se valorará la usabilidad y elegancia, no se tendrá en cuenta si el interfaz consiste simplemente en volcar en una ventana el texto proveniente del interfaz de texto.
- Incluir una opción para dar una pista al usuario, es decir, cuando le toque al usuario jugar, puede tener una opción adicional que lanzará la función *busca_solución* para un tiempo prefijado y mostrará por pantalla la sugerencia de acción a realizar.
- Incluir una opción para que un jugador humano pueda jugar contra otro jugador humano.
- Incluir una opción para que la máquina juegue contra sí misma, enfrentando diferentes versiones del algoritmo o usando diferentes parámetros para cada versión.
- Implementación de tablas por repetición.
- Otras posibles mejoras, tras consultar con el profesor.

Criterios de evaluación

Para que el trabajo pueda ser evaluado, se deberán satisfacer los objetivos mínimos del mismo:

- Correcta implementación del algoritmo UCT.
- Interfaz de texto para jugar contra el usuario.
- Que sea posible jugar contra la máquina.
- Documentación.

Para la evaluación se tendrá en cuenta el siguiente criterio de valoración, considerando una nota máxima de 4 en total para el trabajo:

- **Código fuente y experimentación (hasta 2 puntos)**: se valorará la claridad y buen estilo de programación, corrección, eficiencia y usabilidad de la implementación, y calidad de los comentarios. En ningún caso se evaluará un trabajo con código copiado directamente de Internet o de otros compañeros.
- **Documentación (hasta 1 punto)**: se valorará la claridad de las explicaciones, el razonamiento de las decisiones, el análisis y presentación de resultados y el correcto uso del lenguaje. La elaboración de la memoria debe ser original, por lo que no se evaluará el trabajo si se detecta cualquier copia del contenido.
- **Presentación y defensa (hasta 1 punto)**: se valorará la claridad de la presentación y la buena explicación de los contenidos del trabajo, así como, las respuestas a las preguntas realizadas por el profesor.
- **Mejoras**: Se valorarán hasta con **1 punto** extra sin superar el máximo de 4 puntos totales del trabajo. No se considerarán las mejoras si no se llegan a los mínimos del trabajo.

IMPORTANTE: Cualquier plagio, compartición de código o uso de material que no sea original y del que no se cite convenientemente la fuente, significará automáticamente la calificación de cero en la asignatura para todos los alumnos involucrados. Por tanto, a estos alumnos no se les conserva, ni para la actual ni para futuras convocatorias, ninguna nota que hubiesen obtenido hasta el momento. Todo ello sin perjuicio de las correspondientes medidas disciplinarias que se pudieran tomar.

Bibliografía

[1] Cameron Browne et al. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4:1(1):1-43. *https://ieeexplore.ieee.org/document/6145622*

[2] *http://www.cs.us.es/~fsancho/?e=189*

[3] *https://www.cs.us.es/~fsancho/?e=107*

[4] *https://www.cs.us.es/cursos/ia1/temas/tema-07.pdf*

[5] *https://es.wikibooks.org/wiki/Manual_de_LaTeX*

[6] *https://www.ieee.org/conferences/publishing/templates.html*