**Aplicación del algoritmo UCT para la creación de una IA para el juego de mesa Hnefatafl.**

**Pablo Espada Hoyo**

**Juan José Casamijtana Benitez**

Contenido

[Resumen 4](#_Toc107249804)

[Introducción 4](#_Toc107249805)

[Preliminares 4](#_Toc107249806)

[Desarrollo realizado 5](#_Toc107249807)

[Board 5](#_Toc107249808)

[Game 6](#_Toc107249809)

[PlayerColor 7](#_Toc107249810)

[Piece 7](#_Toc107249811)

[Move 7](#_Toc107249812)

[UCTNode 8](#_Toc107249813)

[Coordinate 8](#_Toc107249814)

[BoarLoader 8](#_Toc107249815)

[UCT 9](#_Toc107249816)

[Pruebas y experimentación 10](#_Toc107249817)

[Conclusiones 11](#_Toc107249818)

# Resumen

El algoritmo Upper Confidence Tree es una implementación del método de Montecarlo para exploración de árboles, en los cuales se escoge los nodos más prometedores para ser explorados, de forma que se reduce la complejidad de diseñar un algoritmo complejo haciendo uso del azar y escogiendo los mejores resultados obtenidos.

# Introducción

Aquí se presenta el proyecto realizado, los objetivos cumplidos y los problemas encontrados. El proyecto fue desarrollado en un principio en Java con el objetivo de usar la mayor eficiencia del lenguaje para encontrar mejores soluciones en el mismo tiempo que lo que se conseguiría con Python.

Sin embargo debido a problemas con la implementación se intentó hacer en Python. Al final esta opción resultó ser funcional, al contrario que la versión hecha en Java, que probablemente necesitaría alguna modificación para que el pseudocódigo propuesto fuese funcional.

# Preliminares

Para la las estructuras de datos se generaron varias clases distintas:

* Board:
  + boardDisplay
* Coordinate
  + rowIndex
  + columIndex
* Piece (enumerado): EMPTY, BLAK\_PAWN, WHITE\_PAWN, WHITE\_KING
* Move
  + initialCoordinate
  + finalCoordinate
* PlayerColor (enumerado): BLACK, WHITE
* Game:
  + board
  + turn
  + movesPlayed
  + moveLimit
* UCTNode
  + State
  + N
  + Q
  + I
  + Children
  + Father

Board en un entidad que guarda la matriz con el tablero y calcula los movimientos posibles para cada jugador. Coordinate guarda la fila y columna de una pieza en concreto dentro de la matriz y Move es una coordenada de origen y una de fin. De esta forma se limita los movimientos del jugador a solo movimientos permitidos y se facilita la indexación de los elementos de la matriz. El uso de enumerados ayuda a la lectura del código y a la claridad quedando en cada caso que significa cada uno de los valores enteros con los que se define la matriz y los jugadores. La entidad Game contiene el estado actual del juego y define a que jugador le toca jugar, que movimientos se han jugado y cual es el límite de movimientos y si este se ha alcanzado. En cuanto a UCTNode de la forma en la que se explica en el pseudocódigo propuesto guarda la información nodo a nodo conteniendo el estado del juego, el numero de partidas, el numero de victorias, los hijos y su padre.

# Desarrollo realizado

Como el proyecto fue finalmente realizado en Python aquí se mostrará y explicará el funcionamiento del código relacionado con cada una de las clases implementadas:

## Board

def integer\_display\_to\_board(integerDisplay):

Convierte un tablero basado en enteros a un tablero basado en el enumerado Piece para tratarlo internamente.

def copy(self):

Copia un tablero en concreto creando uno nuevo con las mismas propiedades.

def is\_surroundings\_of\_throne(self, c):

Comprueba si hay una pieza en las inmediaciones del trono.

def get\_player\_available\_moves(self):

Comprueba pieza a pieza los movimientos que puede hacer y se los asigna a un jugador en concreto.

def get\_available\_moves\_of\_piece(currentCoordinate, boardDisplay):

Coge una pieza en concreto y calcula los movimientos que esta puede realizar extendiendo una lista tras comprobar cada una de las direcciones en las que se puede mover una pieza.

def check\_pieces\_taken(nextBoard):

Comprueba si alguna de las piezas del tablero ha sido capturada.

Otras funciones que se han excluido por ser simples o autoexplanatorias:

* def print\_board(self):
* def is\_king\_alive(self):
* def get\_piece\_at\_coordinate(self, c):
* def check\_down\_moves(piece, currentCoordinate, columns, rows, boardDisplay, keySquares):
* def check\_up\_moves(piece, currentCoordinate, columns, rows, boardDisplay, keySquares):
* def check\_right\_moves(piece, currentCoordinate, columns, rows, boardDisplay, keySquares):
* def check\_left\_moves(piece, currentCoordinate, columns, rows, boardDisplay, keySquares):
* def is\_white(piece):
* def is\_black(piece):
* def get\_throne(self):
* def get\_num\_of\_columns(self):
* def get\_num\_of\_rows(self):

## Game

(Las funciones aquí son mas sencillas así que solo se explicara brevemente lo que hace cada una)

def copy(self):

Copia un un estado de juego.

def get\_initial\_state(variant, moveLimit):

Genera un estado inicial dependiendo de la variante escogida y del límite de movimientos.

def get\_moves(self):

Obtiene los movimientos del jugador activo.

def apply\_move(self, move):

Devuelve un estado del juego en el que se ha aplicado el movimiento.

def print\_state(self):

def is\_final\_state(self):

def white\_wins(self):

def black\_wins(self):

Las otras clases son sencillas y están hechas para contener información o representarla:

## PlayerColor

from enum import Enum

class PlayerColor(Enum):

    BLACK = 1

    WHITE = 2

## Piece

from enum import Enum

class Piece(Enum):

    EMPTY = 0

    BLACK\_PAWN = 1

    WHITE\_PAWN = 2

    WHITE\_KING = 3

## Move

from Coordinate import Coordinate

class Move:

    def \_\_init\_\_(self, initialCoordinate, finalCoordinate):

        self.initialCoordinate = initialCoordinate

        self.finalCoordinate = finalCoordinate

    def \_\_init\_\_(self, row1, col1, row2, col2):

        self.initialCoordinate = Coordinate(row1, col1)

        self.finalCoordinate = Coordinate(row2, col2)

    def \_\_eq\_\_(self, other):

        return self.initialCoordinate.\_\_eq\_\_(other.initialCoordinate) and self.finalCoordinate.\_\_eq\_\_(other.finalCoordinate)

    def \_\_str\_\_(self):

        return str(self.initialCoordinate) + "->" + str(self.finalCoordinate)

## UCTNode

class UCTNode:

    def \_\_init\_\_(self, state, n, q, i, children, father):

        self.state = state

        self.moves = state.get\_moves()

        self.n = n

        self.q = q

        self.i = i

        self.children = children

        self.father = father

    def \_\_str\_\_(self):

        #Node[isFinalState=%s, id=%s, n=%d, q=%d, i=%d, father=%s, children=%s, moves=%d]

        return "Node[isFinalState="+str(self.state.is\_final\_state())+", n="+str(self.n)+", q="+str(self.q)+ ", i="+str(self.i)+ ", children="+ str(len(self.children))+", moves="+str(len(self.moves))+"]"

## Coordinate

class Coordinate:

    def \_\_init\_\_(self, rowIndex, columnIndex):

        self.rowIndex = int(rowIndex)

        self.columnIndex = int(columnIndex)

    def \_\_str\_\_(self):

        return "(" + str(self.rowIndex) + "," + str(self.columnIndex) + ")"

    def \_\_eq\_\_(self, other):

        return self.rowIndex == other.rowIndex and self.columnIndex == other.columnIndex

## BoarLoader

class BoardLoader:

    def load\_board(variant):

        if variant == 1:

            return BoardLoader.board\_reader("initialBoards/HNEFATAFL.txt")

        elif variant == 2:

            return BoardLoader.board\_reader("initialBoards/TABLUT.txt")

        elif variant == 3:

            return BoardLoader.board\_reader("initialBoards/ARD\_RI.txt")

        elif variant == 4:

            return BoardLoader.board\_reader("initialBoards/BRANDUBH.txt")

        elif variant == 5:

            return BoardLoader.board\_reader("initialBoards/TAWLBWRDD.txt")

        elif variant == 6:

            return BoardLoader.board\_reader("initialBoards/ALEA\_EVANGELII.txt")

        else:

            raise Exception("Variante no válida")

    def board\_reader(file):

        lines = []

        with open(file) as f:

            lines = f.readlines()

        integerDisplay = []

        for line in lines:

            row = line.split(",")

            integerDisplay.append(row)

        return Board.integer\_display\_to\_board(integerDisplay)

## UCT

Aquí es donde se encuentra contenido el cuerpo del algoritmo.

def look\_for\_solution(s0, t):

        v0 = UCT.create\_node(s0, None)

        timePassed = 0

        while(timePassed < t):

            start = time.time()

            v1 = UCT.tree\_policy(v0)

            delta = UCT.default\_policy(v1)

            UCT.backup(v1, delta)

            end = time.time()

            timePassed += (end - start)

        return v0.moves[UCT.best\_child(v0,0)]

La versión implementada de busca\_solucion()

def create\_node(s, father):

        return UCTNode(s, 0, 0, 0, [], father)

La versión implementada correspondiente a crear\_nodo()

def tree\_policy(v):

        while(not v.state.is\_final\_state()):

            if v.i < len(v.state.get\_moves()):

                return UCT.expand(v)

            else:

                v = v.children[UCT.best\_child(v, 1/sqrt(2))]

        return v

def expand(v):

        s = v.state.apply\_move(v.moves[v.i])

        v.i = v.i +1

        child = UCT.create\_node(s,v)

        v.children.append(child)

        return child

def best\_child(v,c):

        iRes = 0

        valMax = -99999999.0

        for i in range(0, len(v.children)):

            value = ((v.children[i].q/v.children[i].n) + c\*sqrt(2\*math.log(v.n)/v.children[i].n)).real

            if value > valMax:

                valMax = value

                iRes = i

        return iRes

def default\_policy(v):

        s = v.state

        movs = v.moves

        player = v.father.state.turn

        while(not s.is\_final\_state()):

            a = movs[random.randint(0, len(movs)-1)]

            s = s.apply\_move(a)

            movs = s.get\_moves()

        if (s.white\_wins() and player == 2) or (s.black\_wins() and player == 1):

            return 1

        else:

            return -1

# Pruebas y experimentación

Entre las pruebas realizadas se ha hecho sencillamente partidas de la máquina con sigo misma, un color usando el algoritmo y el otro sencillamente realizando movimientos al azar. De esta forma si gana de forma consistente podemos asegurar que el algoritmo funciona mejor que “un mono lanzando dardos a una diana”.

En un primer momento este modo de prueba parece ser eficiente, aunque se puede comprobar que hay situaciones en las que a pesar de que puede seleccionar un movimiento ganador puede decidir no hacerlo. Esto no significa que haya situaciones en las que pierda, mas haya de eso, es bastante consistente.

Aquí se muestran algunos ejemplos:

Texto

Descripción generada automáticamente

Aquí vemos los resultados de una de las partidas de la IA jugando como las negras, en esta prueba se realizaron 15 partidas un lado de la maquina usando el algoritmo y el otro eligiendo movimientos totalmente al azar, los resultados fueron 13-2 para el algoritmo jugando como negras.

# Conclusiones