Inteligencia Artificial

Segunda Práctica. Hito 2: Algoritmo Minimax y poda alfa-beta

Jose Domingo López López

21 de abril de 2009

Índice

1. Clase Casilla

```
from config import *
import psyco

psyco.full()

class Casilla:
    def __init__ (self, idCasilla, tipo):
        self.idCasilla = idCasilla
        self.tipo = tipo


    def setIdCasilla (self, idCasilla):
        self.idCasilla = idCasilla


    def getIdCasilla (self):
        return self.idCasilla


    def setTipo (self, tipo):
        self.tipo = tipo


    def getTipo (self):
        return self.tipo


    def convertirHierba (self):
        self.setTipo (T_HIERBA)


    def cavar (self):
        if self.getTipo() == T_HOYO: self.setTipo (T_ZANJA)
        else:
            if self.getTipo() <> T_ZANJA: self.setTipo (T_HOYO)


    def __eq__(self, other):
        return self.idCasilla==other.idCasilla


    def __str__(self):
        cadena = "CASILLA::: Id: %d\n" % self.getIdCasilla()
        cadena += "CASILLA::: Tipo: %d\n" % self.getTipo()
        return cadena
```

2. Clase Cliente

```
In  #!/usr/bin/python
# -*- coding: utf-8 -*-

import Ice, sys, time, threading, psyco, datetime

Ice.loadSlice('-I/usr/share/slice ../slice/Practica.ice')

import Practica

from tablero import *
from equipo import *
from estado import *
from nodo import *
from minimax import *


psyco.full()


def Inicializar(mapa, equipos, tablero):
    global_vars.casillasIniciales = []
    global_vars.banderasObjetivo = []
    global_vars.distanciaBanderas = {}
    # ID de usuario, MIN y MAX
    global_vars.idUsuario = ((mapa.idUsuario - 1)%2)+1
    print "idUsuario %d" % (global_vars.idUsuario)
                                            #test
    global_vars.MAX = global_vars.idUsuario - 1
    print "MAX %d" % (global_vars.MAX)
                                                    #test
    global_vars.MIN = (global_vars.MAX + 1)%2
    print "MIN %d" % (global_vars.MIN)
                                                    #test
    # Filas y columnas del tablero
    global_vars.filasTablero = mapa.dimy
    global_vars.columnasTablero = mapa.dimx
    # Inicializamos las casillas iniciales que nos da el servidor
    for index,casilla in enumerate(mapa.casillas):
        cas = Casilla(index+1,casilla)
        global_vars.casillasIniciales.append(cas)
        if casilla == T_BANDERA:
            tablero.banderas += 1
            global_vars.banderasObjetivo.append(index+1)
    # TEST PARA VER SI SE INICIALIZAN BIEN LAS CASILLASINICIALES
                              #test
    #print mapa.casillas
                                                                    #test
    #cad = "["
                                                                        #
        test
    #for i in global_vars.casillasIniciales: cad += str(i.tipo) + ", "
                          #test
    #cad += "]"
                                                                      #
        test
    #print cad
                                                                        #
        test
    # Inicializamos las distancias a cada bandera
    for i in global_vars.banderasObjetivo:
        distancias = []
        distancias = tablero.WayTracking(i)
```

```
            global_vars.distanciaBanderas[i] = distancias[:]
        # TEST PARA VER SI SE CALCULAN BIEN LAS DISTANCIAS
                                            #test
        #for j in global_vars.distanciaBanderas:
                                                        #test
        #    f=0

            #test
        #    for i in range(global_vars.filasTablero):
                                                    #test
        #        print global_vars.distanciaBanderas[j][f:(i+1)*global_vars.
            columnasTablero] #test
        #        f += global_vars.columnasTablero
                                                    #test
        # Inicializamos los equipos. Cada equipo inicializa sus jugadores en el
            constructor
        idEquipo = 1
        for i in mapa.jugadores:
            if idEquipo == i.equipo:
                eqAux = Equipo (idEquipo, mapa.jugadores)
                equipos.append(eqAux)
                idEquipo += 1
        # TEST PARA VER SI SE INICIALIZAN BIEN LOS EQUIPOS Y LOS JUGADORES
                            #test
        #for i in equipos: print str(i)
                                                            #test


class Client (Ice.Application):

    def printHelp(self):
        print "\nEjecucion del cliente para las practicas de IA: \n"
        print "- Para jugar una partida individual ejecutar dos instancias del
            programa con la linea: ./Client 1 --Ice.Config=(Ruta Archivo .config)\n
            "
        print "- Para jugar una competicion ejecutar una instancia del programa con
             la linea: ./Client 2 --Ice.Config=(Ruta Archivo .config)\n  Esperar a
            que el oponente se una a la partida.\n\n"

    def run(self, argv):
        base = self.communicator().stringToProxy("AutenticacionObject")

        autenticacion = Practica.AutenticacionPrx.checkedCast(base)
        if not autenticacion:
            print "ERROR"

        global_vars.deadline = input ("Introduce la duracion de cada turno: ")

        partida = autenticacion.login("71219116", "asdfqwer")
        print str(partida)

        mapa = partida.obtenerMapa("71219116")
        equipos = []
        tablero = Tablero()
        Inicializar (mapa, equipos, tablero)
        estado_actual = Estado (tablero, equipos)
        nodo_actual = Nodo (estado_actual)
        minimax = Minimax(nodo_actual)
        #print mapa
        print "Esperando a que se una otro jugador..."

        while True:
            infoJugada = partida.pedirTurno(global_vars.idUsuario)
            print infoJugada
            if infoJugada.resultado != 0:
                if infoJugada.resultado == 1:
                    print "Has ganado. Juguemos la siguiente partida"
```

```
            else:
                print "Has perdido. Juguemos la siguiente partida"
            break

        if infoJugada.mov.idJugador <> -1 and infoJugada.mov.mov <> -1:
            jug = (infoJugada.mov.idJugador-1)%(len(mapa.jugadores)/2)
            casillas = estado_actual.tablero.idCasillasVecinas(equipos[
                global_vars.MIN].jugadores[jug].casilla)
            estado_actual.actualizarEstado(global_vars.MIN, jug, tablero.
                casillaActual(casillas[infoJugada.mov.mov-1]))
            minimax.nodoMejor = copy.deepcopy(nodo_actual)
            print str(estado_actual.equipos[global_vars.MIN])

        print "ANTES"
        print "band ", estado_actual.tablero.banderas
        estado_actual.minimaDistancia(global_vars.MAX, 1)
        estado_actual.minimaDistancia(global_vars.MIN, 1)
        #raw_input()

        d1 = datetime.datetime.now()
        bella_durmiente = threading.Thread(target=minimax.Sync, args=())
        buscador = threading.Thread(target=minimax.decision_minimax, args=())
        bella_durmiente.start()
        buscador.start()
        bella_durmiente.join()
        #minimax.Sync()

        (jugador, movimiento) = minimax.nodoMejor.accion
        print datetime.datetime.now()-d1
        #jugador = int(raw_input("Jugador: "))
        if jugador == 0:
            break
        #movimiento = int(raw_input("Movimiento: "))
        try:
            print "idJugador = ", jugador
            print "mov = ", movimiento
            devuelto = partida.jugada(global_vars.idUsuario, jugador,
                movimiento, infoJugada.token)
            print "Movimiento realizado: " + str(devuelto)
        except Practica.MovimientoIncorrectoError, e:
            print e.reason
            print "Has hecho un movimiento incorrecto. Durmiendo %d segundos
                para perder automaticamente" % (global_vars.deadline)
            time.sleep(global_vars.deadline)
            break
        except Practica.TokenIncorrectoError, e:
            print e.reason
            print "Has agotado el tiempo. Juguemos la siguiente partida"
            break

        jug = (jugador-1)%(len(mapa.jugadores)/2)
        casillas = estado_actual.tablero.idCasillasVecinas(equipos[global_vars.
            MAX].jugadores[jug].casilla)
        estado_actual.actualizarEstado(global_vars.MAX, jug, tablero.
            casillaActual(casillas[movimiento-1]))
        print "DESPUES"
        print "band ", estado_actual.tablero.banderas
        estado_actual.minimaDistancia(global_vars.MAX, 1)
        estado_actual.minimaDistancia(global_vars.MIN, 1)
        minimax.nodoMejor = copy.deepcopy(nodo_actual)
        print str(estado_actual.equipos[global_vars.MAX])

raw_input("Presione ENTER para continuar...")

mapa = partida.obtenerMapa("71219116")
equipos = []
tablero = Tablero()
```

```
        Inicializar (mapa, equipos, tablero)
        estado_actual = Estado (tablero, equipos)
        nodo_actual = Nodo (estado_actual)
        minimax = Minimax(nodo_actual)
        #print mapa

        while True:
            #infoJugada = partida.pedirTurno(mapa.idUsuario)
            infoJugada = partida.pedirTurno(global_vars.idUsuario)
            print infoJugada
            if infoJugada.resultado != 0:
                if infoJugada.resultado == 1:
                    print "Has ganado"
                else:
                    print "Has perdido"
                break

            jugador = int(raw_input("Jugador: "))
            if jugador == 0:
                break
            movimiento = int(raw_input("Movimiento: "))
            try:
                #devuelto = partida.jugada(mapa.idUsuario, jugador, movimiento,
                    infoJugada.token)
                devuelto = partida.jugada(global_vars.idUsuario, jugador,
                    movimiento, infoJugada.token)
                print "Movimiento realizado: " + str(devuelto)
            except Practica.MovimientoIncorrectoError, e:
                print e.reason
                print "Has hecho un movimiento incorrecto. Durmiendo %d segundos
                    para perder automaticamente." % (global_vars.deadline)
                time.sleep(global_vars.deadline)
                break
            except Practica.TokenIncorrectoError, e:
                print e.reason
                print "Has agotado el tiempo y has perdido la partida."
                break

        print "Han finalizado las dos partidas"
        time.sleep(3)
        ret = autenticacion.finalizarPartida("71219116","asdfqwer")

Client().main(sys.argv)
```

3. Clase Equipo

```
from jugador import *
import psyco

psyco.full()


class Equipo:
    def __init__ (self, idEquipo, jugadores_ice, banderasCapturadas = 0):
        self.idEquipo = idEquipo
        self.banderasCapturadas = banderasCapturadas
        self.jugadores = []
        for jug in jugadores_ice:
            #jugadores_ice es una estructura de tipo Jugador definida en Practica.
                ice
            if jug.equipo == self.idEquipo:
                jugAux = Jugador(jug.idJugador, jug.equipo, jug.casilla, jug.
                    energia)
                self.jugadores.append(jugAux)


    def getIdEquipo (self):
        return self.idEquipo


    def getBanderasCapturadas (self):
        return self.banderasCapturadas


    def setBanderasCapturadas (self, valor):
        self.banderasCapturadas = valor


    def capturarBandera(self):
        self.setBanderasCapturadas(self.getBanderasCapturadas() + 1)


    def getJugadores(self):
        return self.jugadores


    def __str__ (self):
        cadena = "EQUIPO::: ID Equipo: %d\n" % self.getIdEquipo()
        cadena += "EQUIPO::: Banderas capturadas: %d\n" % self.
            getBanderasCapturadas()
        for i in self.jugadores: cadena += str(i)
        return cadena
```

4. Clase Estado

```
from config import *
import global_vars
import psyco

psyco.full()


class Estado:


    def __init__ (self, tablero, equipos):
        self.tablero = tablero
        self.equipos = equipos


    def esSolucion (self):
        if self.tablero.banderas==0: return True
        else: return False


    def actualizarEstado (self, eq, jug, casillaDestino):
        """ Actualiza las componentes del estado: tablero y jugadores.
        Del tablero se actualizan las casillas modificadas (si se ha modificado
        alguna) y de los jugadores se actualizan la vida y los objetos.
            RECIBE "jug" que es el indice del jugador en la lista de jugadores
        (normalmente idJugador-1) y la casilla a la que se mueve (tipo Casilla)
            DEVUELVE nada.
        """
        equipoActor = self.equipos[eq]
        jugadorActor = self.equipos[eq].jugadores[jug]
        accionEjecutada = True
        if jugadorActor.getCasilla() <> casillaDestino.getIdCasilla(): # El jugador
             se mueve
            self.equipos[eq].jugadores[jug].setCasilla(casillaDestino.getIdCasilla
                ())
            # Dirty indica si es necesario actualizar las casillas modificadas del
                tablero
            dirty = 0
            if casillaDestino.getTipo() == T_HIERBA:
                accionEjecutada = jugadorActor.perderEnergia(1)
            elif casillaDestino.getTipo() == T_AGUA:
                accionEjecutada = jugadorActor.usarBarca()
            elif casillaDestino.getTipo() == T_BARRO:
                accionEjecutada = jugadorActor.perderEnergia(2)
            elif casillaDestino.getTipo() == T_HOYO:
                accionEjecutada = jugadorActor.perderEnergia(4)
            elif casillaDestino.getTipo() == T_ZANJA:
                accionEjecutada = jugadorActor.perderEnergia(6)
            elif casillaDestino.getTipo() == T_BANDERA:
                dirty = 1
                equipoActor.capturarBandera()
                self.tablero.banderas -= 1
            elif casillaDestino.getTipo() == T_BARCA:
                dirty = 1
                jugadorActor.cogerBarca()
            elif casillaDestino.getTipo() == T_HACHA:
                dirty = 1
                jugadorActor.cogerHacha()
            elif casillaDestino.getTipo() == T_ZUMO:
                dirty = 1
                jugadorActor.beberZumo()
            elif casillaDestino.getTipo() == T_PALA:
```

```
                dirty = 1
                jugadorActor.cogerPala()
            elif casillaDestino.getTipo() == T_BOSQUE:
                (accionEjecutada, hacha) = jugadorActor.usarHacha()
                if accionEjecutada and hacha: dirty = 1
            # Si el dirty bit vale 1, actualizamos la casilla
            if dirty == 1:
                 casillaDestino.convertirHierba()
                 self.tablero.anadirCasillaModificada(casillaDestino)
        else: # El jugador no se mueve utiliza la pala
            if casillaDestino.getTipo() == T_HIERBA or casillaDestino.getTipo() ==
                T_ZANJA or casillaDestino.getTipo() == T_HOYO:
                if jugadorActor.usarPala(): # El jugador tiene pala para usarla
                    satisfactoriamente
                    if casillaDestino in self.tablero.getCasillasModificadas():
                        self.tablero.eliminarCasillaModificada(casillaDestino)
                    casillaDestino.cavar()
                    self.tablero.anadirCasillaModificada(casillaDestino)
                else: # El jugador NO tiene pala, por lo que no puede ejecutar la
                    accion
                    accionEjecutada = False
            else: # No se puede cavar en la casilla
                accionEjecutada = False
        return accionEjecutada


    def minimaDistancia (self, eq, a=0):
        """ RECIBE el identificador de equipo que hay que examinar
            DEVUELVE una tupla (idJugador, bandera, distancia)
        """
        equipo = self.equipos[eq]
        minimaDistancia = INFINITO
        jugador = -1
        bandera = -1
        for band in global_vars.banderasObjetivo:
            if self.tablero.casillaActual(band).getTipo() == T_BANDERA:
                distancias = global_vars.distanciaBanderas[band]
                for jug in equipo.jugadores:
                    dist = distancias[jug.casilla-1]
                    if a == 1: print "dist del jug %d (casilla %d) a la bandera %d:
                         %d" % (jug.idJugador,jug.casilla,band,dist)
                    if dist < minimaDistancia:
                        minimaDistancia = dist
                        jugador = jug.idJugador
                        bandera = band
        if a == 1: print "min: ", minimaDistancia
        #raw_input()
        return (jugador, bandera, minimaDistancia)


    def __eq__(self, other):
        return self.tablero==other.tablero and self.jugadores==other.jugadores
```

5. Clase Jugador

```
import global_vars, config
import psyco

psyco.full()

class Jugador:
    def __init__ (self, idJugador, equipo, casilla, energia, hacha = 0, barca = 0,
        pala = 0, banderas = 0, zumo = 0):
        self.idJugador = idJugador
        self.equipo = equipo
        self.casilla = casilla # Identificador (int) de la casilla actual del
            jugador
        self.energia = energia
        self.hacha = hacha
        self.barca = barca
        self.pala = pala


    def getIdJugador (self):
        return self.idJugador


    def getEquipo (self):
        return self.equipo


    def getCasilla (self):
        return self.casilla


    def setCasilla (self, valor):
        self.casilla = valor


    def getEnergia (self):
        return self.energia


    def setEnergia (self, valor):
        self.energia = valor


    def beberZumo (self):
        self.energia = self.energia + 20


    def perderEnergia (self, valor):
        energiaActual = self.energia - valor
        if energiaActual > 0:
            self.energia = energiaActual
            return True
        else:
            return False


    def getHacha (self):
        return self.hacha


    def setHacha (self, valor):
        self.hacha = valor
```

```
    def usarHacha (self):
        """ Cuando se entra en una casilla de bosque se utiliza el hacha siempre
        que se tenga. No hay eleccion a utilizarla o no.
            DEVUELVE True si ha utilizado el hacha y False si no la ha utilizado
        """
        if self.getHacha() > 0:
            if self.perderEnergia(4):
                self.setHacha(self.getHacha() - 1)
                return (True, True) # Se mueve y usa el hacha
            else:
                return (False, False) # No se puede mover asi que no usa el hacha
        else:
            if self.perderEnergia(8):
                return (True, False) # Se mueve y no usa el hacha
            else:
                return (False, False) # No se puede mover y no tiene hacha


    def cogerHacha (self):
        self.setHacha(self.getHacha() + 20)


    def getBarca (self):
        return self.barca


    def setBarca (self, valor):
        self.barca = valor


    def usarBarca (self):
        """ Cuando se entra en una casilla de agua se utiliza la barca siempre
        que se tenga. No hay eleccion a utilizarla o no.
        """
        if self.getBarca() > 0:
            self.perderEnergia(3)
            self.setBarca(self.getBarca() - 1)
        else:
            self.perderEnergia(6)


    def cogerBarca (self):
        self.setBarca(self.getBarca() + 20)


    def getPala (self):
        return self.pala


    def setPala (self, valor):
        self.pala = valor


    def usarPala (self):
        if self.getPala() > 0:
            self.setPala(self.getPala() - 2)
            return True
        else:
            return False


    def cogerPala (self):
        self.setPala(self.getPala() + 10)


    def __str__ (self):
```

```
        cadena = "JUGADOR::: ID Jugador: %d\n" % self.getIdJugador()
        cadena += "JUGADOR::: Equipo: %d\n" % self.getEquipo()
        cadena += "JUGADOR::: Casilla: %d\n" % self.getCasilla()
        cadena += "JUGADOR::: Energia: %d\n" % self.getEnergia()
        cadena += "JUGADOR::: Hacha: %d\n" % self.getHacha()
        cadena += "JUGADOR::: Barca: %d\n" % self.getBarca()
        cadena += "JUGADOR::: Pala: %d\n" % self.getPala()
        return cadena


    def __eq__(self, other):
        if global_vars.algoritmo == config.A_ESTRELLA or global_vars.algoritmo ==
            config.PROFUNDIDAD or global_vars.algoritmo == config.
            PROFUNDIDAD_ITERATIVA:
            return self.idJugador==other.idJugador and self.equipo==other.equipo
                and self.casilla==other.casilla and self.hacha==other.hacha and
                self.barca==other.barca and self.pala==other.pala
        elif global_vars.algoritmo == config.ANCHURA:
            return self.idJugador==other.idJugador and self.equipo==other.equipo
                and self.casilla==other.casilla and self.energia==other.energia and
                 self.hacha==other.hacha and self.barca==other.barca and self.pala
                ==other.pala
```

6. Clase Minimax

```
import copy, psyco, time, global_vars, math
from config import *
from nodo import *


psyco.full()


class Minimax:
    def __init__(self, nodo_inicial):
        self.nodoInicial = nodo_inicial
        self.nodoMejor = copy.deepcopy(nodo_inicial)
        self.timeout = False


    def Sync (self):
        tiempo = 0
        self.timeout = False
        while tiempo < global_vars.deadline - SECURITY_RANGE:
            time.sleep(TIME_INCREMENT)
            tiempo += TIME_INCREMENT
        self.timeout = True
        print "Tiempo: %f" % (tiempo)


    def decision_minimax(self):
        limite = 0
        # TODO poner otra condicin por si la busqueda finaliza antes que el timeout
        while not self.timeout:
            print limite
            self.max_valor(self.nodoInicial, limite, -INFINITO, INFINITO)
            limite += 1


    def max_valor(self, nodo, limite, alfa, beta):
        if self.test_terminal (nodo, limite):
            return self.evaluacion(nodo, global_vars.MAX, global_vars.MIN)

        v = -INFINITO
        sucesores = self.expandir (nodo, global_vars.MAX)

        i = 0
        while not self.timeout and i < len(sucesores):
            v = max(v, self.min_valor (sucesores[i], limite, alfa, beta))
            #if not timeout????
            if v >= beta:
                return v
            alfa = max (alfa, v)
            # Asignamos al nodo MAX el mayor valor de utilidad de sus hijos
            #nodo.utilidad = v
            # Elegimos el mejor nodo de profundidad 1 (primer movimiento de MAX)
            if nodo.profundidad == PROF_INICIAL:
                #if nodo > self.nodoMejor: self.nodoMejor = copy.deepcopy(sucesores
                    [i])
                if v > self.nodoMejor.utilidad:
                    self.nodoMejor = copy.deepcopy(sucesores[i])
                    self.nodoMejor.utilidad = v
            i += 1
        return v


    def min_valor(self, nodo, limite, alfa, beta):
```

```
        if self.test_terminal (nodo, limite):
            return self.evaluacion(nodo, global_vars.MIN, global_vars.MAX)

        v = INFINITO
        sucesores = self.expandir (nodo, global_vars.MIN)

        i = 0
        while not self.timeout and i < len(sucesores):
            v = min(v, self.max_valor (sucesores[i], limite, alfa, beta))
            #if not timeout????
            if v <= alfa:
                return v
            beta = min (beta, v)
            # Asignamos al nodo MIN el menor valor de utilidad de sus hijos
            #nodo.utilidad = v
            i += 1
        return v



    def expandir(self, nodo, equipo):
        sucesores = []
        numero_jugadores = len(nodo.estado.equipos[equipo].jugadores)
        for jug in range(numero_jugadores):
            if nodo.estado.equipos[equipo].jugadores[jug].getEnergia() > 0:
                casillasVecinas = nodo.estado.tablero.casillasVecinasActuales(nodo.
                    estado.equipos[equipo].jugadores[jug].getCasilla())
                # ANadimos la casilla actual para hacer hoyos
                if nodo.estado.equipos[equipo].jugadores[jug].pala > 0:
                    casillasVecinas.append(nodo.estado.tablero.casillaActual(nodo.
                        estado.equipos[equipo].jugadores[jug].getCasilla()))
                for mov in range (len(casillasVecinas)):
                    # Comprobamos que sea la misma casilla o, si es distinta
                        casilla que no sea muralla
                    #es decir, comprobamos que si nos movemos, la casilla destino
                        NO sea una muralla
                    if casillasVecinas[mov].getIdCasilla() == nodo.estado.equipos[
                        equipo].jugadores[jug].getCasilla() or casillasVecinas[mov
                        ].getTipo() != T_MURALLA:
                        estadoAux = copy.deepcopy(nodo.estado)
                        if estadoAux.actualizarEstado (equipo, jug, casillasVecinas
                            [mov]):
                            accion = (estadoAux.equipos[equipo].jugadores[jug].
                                getIdJugador(), mov+1)
                            nuevoNodo = Nodo (estadoAux, nodo, accion, nodo.
                                profundidad+1)
                            sucesores.append(nuevoNodo)
        return sucesores


    def test_terminal (self, nodo, limite):
        terminal = False
        # Comprueba si quedan banderas en el tabero, si se ha llegado a la
            profundidad
        #de corte (prof. iterativa) o si se ha acabado el tiempo
        if nodo.estado.esSolucion() or nodo.profundidad==limite: # or self.timeout:
            terminal = True
        #else: # Si no se cumple nada de lo anterior, miramos si quedan jugadores
            vivos
        #     for jug in nodo.estado[equipo].jugadores:
        #         if jug.energia > 0:
        #             terminal = False
        #             break
        return terminal


    def evaluacion(self, nodo, equipo1, equipo2):
```

```
estado = nodo.estado
totalBanderas = len(global_vars.banderasObjetivo)

bandEq1 = estado.equipos[equipo1].banderasCapturadas
bandEq2 = estado.equipos[equipo2].banderasCapturadas

"""ratioBandEq1 = bandEq1*totalBanderas#/totalBanderas
ratioBandEq2 = bandEq2*totalBanderas#/totalBanderas
ratioBand = ratioBandEq1 - ratioBandEq2
ratioBand = ratioBand * 20

filas = global_vars.filasTablero
columnas = global_vars.columnasTablero

bondadDist = max(columnas,filas)/2
(x,y,distEq1) = estado.minimaDistancia(equipo1)
#print "distEq1: ", distEq1
(x,y,distEq2) = estado.minimaDistancia(equipo2)
#print "distEq2: ", distEq2
ratioDistEq1 = max(columnas,filas) * math.exp((-1/bondadDist)*distEq1)
#print "ratioDistEq1: ", ratioDistEq1
ratioDistEq2 = max(columnas,filas) * math.exp((-1/bondadDist)*distEq2)
#print "ratioDistEq2: ", ratioDistEq2
ratioDist = ratioDistEq1 - ratioDistEq2
#raw_input()
evalEq1 = ratioBandEq1 + ratioDistEq1
evalEq2 = ratioBandEq2 + ratioDistEq2"""
evaluacion = 20 * (bandEq1 - bandEq2)
(x,y,distEq1) = estado.minimaDistancia(equipo1)
(x,y,distEq2) = estado.minimaDistancia(equipo2)
evaluacion += (10 * (1.0/distEq1)) - (10 * (1.0/distEq2))

#return ratioBand + ratioDist#evalEq1 - evalEq2
return evaluacion
```

7. Clase Nodo

```
from config import *
import psyco

psyco.full()

class Nodo:


    def __init__ (self, estado, padre=None, accion=(-1,-1), profundidad=
        PROF_INICIAL, utilidad=-INFINITO):
        self.estado = estado # Tipo estado
        self.padre = padre # Tipo nodo
        self.accion = accion # Tupla (jugador, accion)
        self.profundidad = profundidad # Tipo entero
        self.utilidad = utilidad


    def repetidoEnRama (self):
        """ Comprueba si el estado del nodo esta repetido en alguna de sus nodos
        antecesores (solo comprueba en su rama).
            DEVUELVE True si esta repetido; False si no esta repetido
        """
        repetido = False
        actual = self
        while not repetido and actual.profundidad > PROF_INICIAL:
            if self == actual.padre: repetido = True
            actual = actual.padre
        return repetido


    def __cmp__(self, other):
        return cmp(self.utilidad, other.utilidad)


    def __eq__(self, other):
        return self.estado==other.estado


    def __str__(self):
        cad = ""
        for i in self.estado.jugadores:
            cad += "Jugador " + str(i.idJugador) + "\tCasilla: " + str(i.casilla) +
                "\tEnergia: " + str(i.energia) + "\n"
        cad += "Accion: " + str(self.accion) + "\tCoste: " + str(self.g) + "\
            tProfundidad: " + str(self.profundidad)
        return cad
```

8. Clase Tablero

```
from casilla import *
import global_vars, copy
import psyco

psyco.full()


class Tablero:
    def __init__ (self, casillas=[], banderas=0):
        self.casillasModificadas = casillas[:]
        self.banderas=banderas


    def getCasillasModificadas(self):
        return self.casillasModificadas


    def setCasillasModificadas(self, seq):
        self.casillasModificadas.extend(seq)


    def eliminarCasillaModificada(self, casilla):
        self.casillasModificadas.remove(casilla)


    def anadirCasillaModificada(self, casilla):
        self.casillasModificadas.append(casilla)


    def idCasillasVecinas (self, casilla):
        """ RECIBE un entero que es el identificador de la casilla para la cual
        queremos encontrar las vecinas.
            DEVUELVE una lista con los identificadores de las casillas vecinas
        dada una casilla en un tablero de casillas hexagonales
        """

        casillasVecinas = [-1, -1, -1, -1, -1, -1]
        columnas = global_vars.columnasTablero
        filas = global_vars.filasTablero
        cruzarAncho = filas*columnas-columnas
        if casilla > 0 and casilla <= columnas: # Primera fila
            casillasVecinas[0] = casilla + cruzarAncho
            if (casilla%2) != 0: # Primera fila e impar
                casillasVecinas[1] = casilla + cruzarAncho + 1
                casillasVecinas[2] = casilla + 1
                casillasVecinas[3] = casilla + columnas
                if casilla == 1: # Primera fila y primera casilla
                    casillasVecinas[4] = columnas
                    casillasVecinas[5] = columnas*filas
                else: # Primera fila y resto de casillas impares
                    casillasVecinas[4] = casilla - 1
                    casillasVecinas[5] = casilla + cruzarAncho - 1
            else: # Primera fila y par
                if casilla == columnas: # Primera fila y ultima casilla(par)
                    casillasVecinas[1] = 1
                    casillasVecinas[2] = casilla + 1
                else: # Primera fila y resto de casillas pares
                    casillasVecinas[1] = casilla + 1
                    casillasVecinas[2] = casilla + columnas + 1
                casillasVecinas[3] = casilla + columnas
                casillasVecinas[4] = casilla + columnas - 1
                casillasVecinas[5] = casilla - 1
```

```
        elif casilla > cruzarAncho and casilla <= columnas*filas: # Ultima fila
            casillasVecinas[0] = casilla - columnas
            if (casilla%2) != 0: # Ultima fila e impar
                casillasVecinas[1] = casilla - columnas + 1
                casillasVecinas[2] = casilla + 1
                casillasVecinas[3] = casilla - cruzarAncho
                if casilla == cruzarAncho + 1: # Ultima fila y primera casilla (
                    impar)
                    casillasVecinas[4] = columnas*filas
                    casillasVecinas[5] = casilla - 1
                else: # Ultima fila y resto de casillas impares
                    casillasVecinas[4] = casilla - 1
                    casillasVecinas[5] = casilla - columnas - 1
            else: # Ultima fila y par
                if casilla == columnas*filas: # Ultima fila y ultima casilla (par)
                    casillasVecinas[1] = casilla - columnas + 1
                    casillasVecinas[2] = 1
                else: # Ultima fila y resto de casillas pares
                    casillasVecinas[1] = casilla + 1
                    casillasVecinas[2] = casilla - cruzarAncho + 1
                casillasVecinas[3] = casilla - cruzarAncho
                casillasVecinas[4] = casilla - cruzarAncho - 1
                casillasVecinas[5] = casilla - 1
        else: # No es ni primera ni ultima fila
            casillasVecinas[0] = casilla - columnas
            if (casilla%2) != 0: # Impar
                casillasVecinas[1] = casilla - columnas + 1
                casillasVecinas[2] = casilla + 1
                casillasVecinas[3] = casilla + columnas
                if ((casilla-1)%columnas == 0): # Borde izquierdo
                    casillasVecinas[4] = casilla + columnas - 1
                    casillasVecinas[5] = casilla - 1
                else:
                    casillasVecinas[4] = casilla - 1
                    casillasVecinas[5] = casilla - columnas - 1
            else: # Par
                if (casilla%columnas) == 0: # Borde derecho
                    casillasVecinas[1] = casilla - columnas + 1
                    casillasVecinas[2] = casilla + 1
                else:
                    casillasVecinas[1] = casilla + 1
                    casillasVecinas[2] = casilla + columnas + 1
                casillasVecinas[3] = casilla + columnas
                casillasVecinas[4] = casilla + columnas - 1
                casillasVecinas[5] = casilla - 1
        return casillasVecinas


    def casillaActual (self, idCasilla):
        """ RECIBE un identificador de casilla
            DEVUELVE un objeto casilla con su identificador y tipo actual
        """
        cas = copy.deepcopy(global_vars.casillasIniciales[idCasilla-1])
        # Si la casilla NO es una muralla, miramos si ha sido modificada
        if cas.getTipo() != T_MURALLA:
            if cas in self.casillasModificadas:
                indice = self.casillasModificadas.index(cas)
                cas.setTipo(self.casillasModificadas[indice].getTipo())
        return cas


    def casillasVecinas (self, casillasActuales, casilla):
        """ RECIBE la lista de casillas actuales del tablero y el identificador
        de la casilla de la cual queremos obtener las casillas vecinas.
            DEVUELVE las 6 casillas vecinas en el tablero actual
        """
        # Obtenemos los identificadores de las casillas vecinas
```

```
        idVecinas = self.idCasillasVecinas(casilla)
        # Construimos las casillas vecinas actuales
        casillasVecinas = []
        for i in idVecinas:
            cas = Casilla (i, casillasActuales[i - 1].getTipo())
            casillasVecinas.append (cas)
        return casillasVecinas


    def casillasVecinasActuales (self, idCasilla):
        """ RECIBE el identificador de la casilla sobre la cual queremos calcular
        las casillas vecinas actuales.
            DEVUELVE las 6 casllas vecinas actuales
            Utilizar esta funcion es mejor que llamar a tableroActual y luego
            llamar a casillasVecinas.
        """
        # Obtenemos los identificadores de las casillas vecinas
        idVecinas = self.idCasillasVecinas(idCasilla)
        # Construimos las casillas vecinas actuales
        casillasVecinas = []
        for i in idVecinas:
            cas = Casilla (i, global_vars.casillasIniciales[i-1].getTipo())
            if cas in self.casillasModificadas:
                indice = self.casillasModificadas.index(cas)
                cas.setTipo(self.casillasModificadas[indice].getTipo())
            casillasVecinas.append(cas)
        return casillasVecinas


    def tableroActual (self):
        """ DEVUELVE una lista con las casillas del tablero actual """
        casillas = global_vars.casillasIniciales[:]
        for i in range(len(self.getCasillasModificadas())):
            casillas[self.casillasModificadas[i].getIdCasilla()-1] = self.
                casillasModificadas[i]
        return casillas


    def __WayTrackingBack (self, k, idCasillaDestino, calculadas, filas, columnas,
        casillasTablero, distancias):
        if calculadas == filas*columnas: return
        else:
            for i in range(len(distancias)):
                if distancias[i] == k:
                    adyacentes = self.casillasVecinasActuales(i+1)
                    for j in adyacentes:
                        if j.tipo != T_MURALLA and distancias[j.idCasilla - 1] > k
                            +1:
                            distancias[j.idCasilla - 1] = k+1
                            calculadas += 1
            self.__WayTrackingBack (k+1, idCasillaDestino, calculadas, filas,
                columnas, casillasTablero, distancias)


    def WayTracking (self, idCasillaDestino):
        columnas = global_vars.columnasTablero
        filas = global_vars.filasTablero
        casillas = global_vars.casillasIniciales[:]
        # Inicializo la lista de distancias a INFINITO
        distancias = []
        for i in range(filas*columnas):
            distancias.append(INFINITO)
            #casillas.append(Casilla(i+1,1))                                #test
        #ponerMurallas(casillas)                                            #test
        # Etiqueto la casilla de la bandera con distancia 0
        calculadas = 1
        distancias[idCasillaDestino-1] = 0
```

```
        # Etiqueto las murallas con distancia -1
        for i in casillas:
            if i.tipo == T_MURALLA:
                distancias[i.idCasilla-1] = -1
                calculadas += 1
        #Imprimir (distancias)                                                  #
            test
        self.__WayTrackingBack(0, idCasillaDestino, calculadas, filas, columnas,
            casillas, distancias)
        return distancias


    def __contains__(self, element):
        return element in self.casillasModificadas


    def __eq__(self, other):
        return self.casillasModificadas==other.casillasModificadas
```

9. Constantes

```
# Constantes que indican el tipo de cada casilla
T_HIERBA = 1
T_AGUA = 2
T_BARRO = 3
T_MURALLA = 4
T_HOYO = 5
T_ZANJA = 6
T_BANDERA = 7
T_BARCA = 8
T_HACHA = 9
T_ZUMO = 10
T_PALA = 11
T_BOSQUE = 12

INFINITO = 99999
PROF_INICIAL = 0
MAX_ACCIONES = 7

TIME_INCREMENT = 0.4
SECURITY_RANGE = TIME_INCREMENT
```

10. Variables globales

```
idUsuario = -1
casillasIniciales = []
banderasObjetivo = [] # ID de la casilla
distanciaBanderas = {}
filasTablero = 0
columnasTablero = 0
MIN = -1
MAX = -1
deadline = 8
```