Sabana africana

ARQUITECTURAS AVANZADAS DE COMPUTADORES

Eduardo José Barrameda Portieles

Índice de contenidos

Índice de figuras

1. Introducción

A continuación, se van a representar las ideas que han sido plasmadas en el diseño e implementación de la práctica “Juego de la sabana africana”.

Comentaré también como he resuelto problemas que han surgido durante el desarrollo de esta.

Destacar que hablaré de cómo he implementado el juego con el uso de programación orientada a objetos, en primer lugar, el juego representará la sabana con todos los animales ya posicionados, posteriormente, al comenzar el juego, la ejecución de este no terminará hasta que una manada de hienas o leones consiga 20 puntos.

Se hablará también de cómo se han implementado las acciones que pueden realizar los animales(comer, vivir y descansar) y las soluciones a diversos problemas que he tenido durante la implementación con hilos.

Por último, comentaré una breve conclusión en la que remarcaré los conceptos aprendidos en clase, las dificultades generales y futuras mejoras.

2. Semejanza del juego con el programa en Python

A continuación, explicaré las diversas clases que tiene el proyecto realizado(véase Ilustración 1), así como el porqué de las distintas variables que tenga la clase, métodos, objetivos de esta y su funcionamiento grosso modo.

La sabana está compuesta por hienas, cebras y leones, antes de nada, lo primero a decidir es la cantidad de leones ya que es la referencia sobre la que se define el número de hienas y de cebras. Por cada león habrá 6 cebras(1:6) y 3 hienas(1:3).

2.1. Animal

La clase Animal hereda de la clase thread, define un animal genérico, tiene varios atributos:

- especie: Define la especie del animal(león, hiena o cebra).
- tipo: Determina si un animal es carnívoro(león y hiena) o herbívoro(cebra).
- casillaN y casillaM: Representan las coordenadas en la matriz de la sabana.
- manada: Atributo utilizado para decir a que manada pertenece un animal.
- vivo: Booleano que nos indica si el animal en la casilla está vivo o muerto.

En la clase se encuentran los getters y setters de cada atributo, métodos propios de la programación orientada a objetos.

Destacar el método run(sobrescrito), es el método que hará que los animales realicen las distintas acciones durante la ejecución del programa hasta que haya un ganador o mueran. Si la acción a realizar por parte del animal es moverse entonces se llama al método getVelocidad (método que devuelve la velocidad del animal dependiendo de su especie y manada), se hace un sleep y se llama al método “moverse”, esto es debido a que cada animal tiene su velocidad correspondiente. Una vez haya un ganador o muera el animal se termina la ejecución de cada hilo.

Tenemos también las clases Herbívoro y Carnívoro que heredan de la clase Animal, además tenemos las clases León y Hiena que heredan de la clase Carnívoro y Cebra que hereda de la clase Herbívoro. Destacar que una cebra no puede cazar(sí comer hierba), las hienas sólo pueden cazar cebras y los leones pueden cazar hienas y cebras.

En esta clase he de destacar los métodos descansar y moverse cuyas implementaciones son las mismas para todos los animales.

Respecto al método moverse en primer lugar obtenemos las casillas adyacentes, para ello tenemos el método getAdyacente que nos devuelve una lista con las casillas libres y otra con las casillas ocupadas. Si existen casillas adyacentes libres el animal se moverá a una de ellas(se elige una casilla aleatoria de la lista), para ello se realizan varias acciones: (Véase Ilustración 2)

- Se bloquea el mutex de la casilla origen, se desocupa la casilla y se libera el mutex
- Se bloquea el mutex de la casilla destino, se ocupa la casilla y se desbloquea el mutex

Respecto al método descansar se realizará un sleep de un una cantidad aleatoria de segundos.

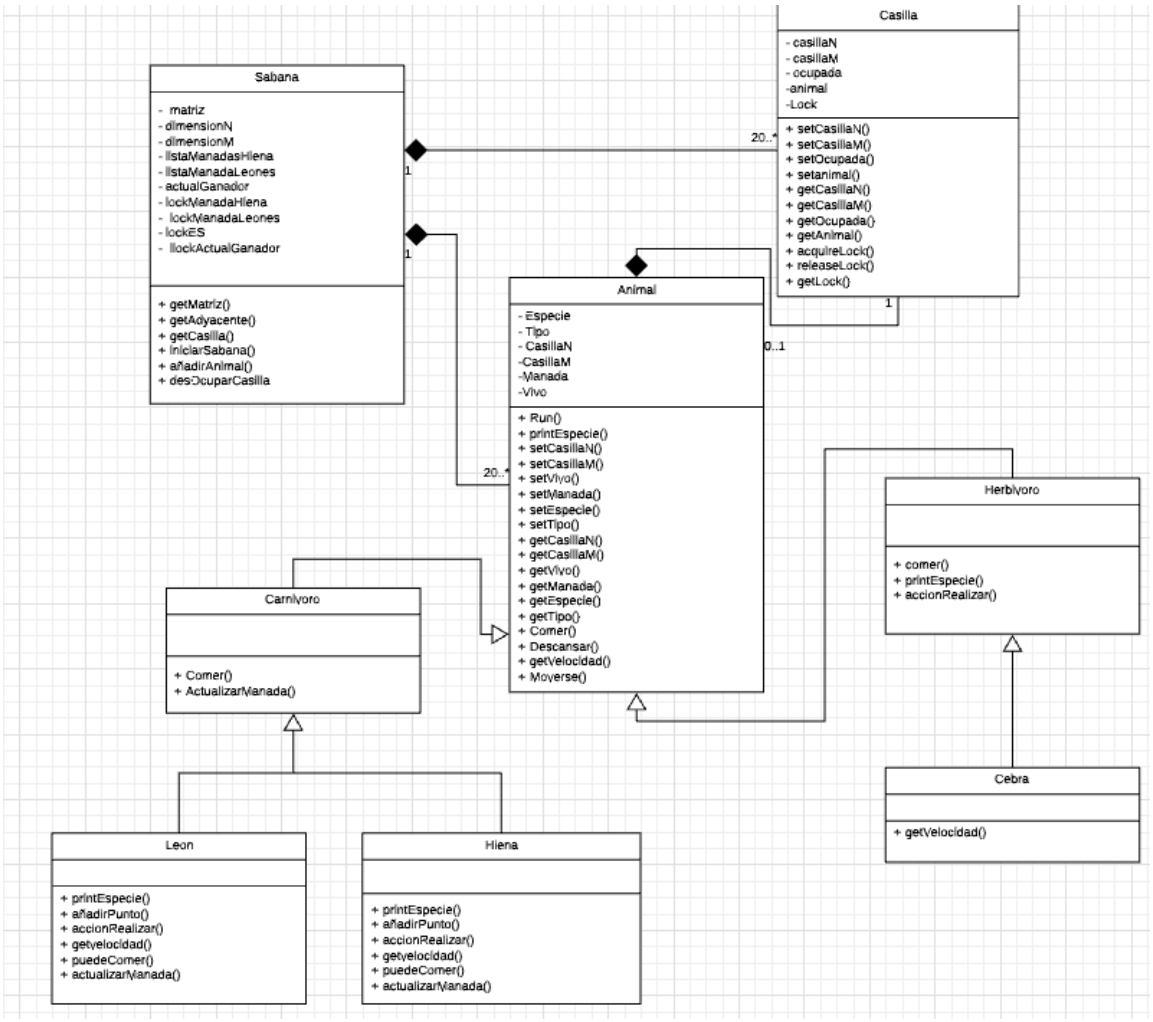


Ilustración 1: Representación de las clases en un diagrama UML, como se puede observar tenemos 2 clases que heredan de Animal, cada animal pertenece a una casilla, una sabana está formada por al menos 20 casillas.

```
def moverse(self):#Todos se mueven igual
    casillaN=self.casillaN
    casillaM=self.casillaM
    listaAdyaLi,listaAdyaOc,bool=self.sabana.getAdyacente(casillaN,casillaM)#obtenemos las adyacentes
    # Si no hay ninguna posicion libre el animal se queda en la misma casilla
    if len(listaAdyaLi) == 0:
        self.sabana.acquireLockES()
        especie=self.printEspecie()
        print(especie+ " se queda en la misma posicion con n " + str(casillaN) + " y m " + str(casillaM))
        self.sabana.releaseLockES()
        return
    else: #si hay casillas libres entonces pilla una aleatoria
        nrandom= random.randint(0,len(listaAdyaLi)-1)
        casillaRandom=self.sabana.getCasilla(listaAdyaLi[nrandom].getCasillaN(), listaAdyaLi[nrandom].getCasillaM())#Representará la casilla aleatoria
        self.sabana.acquireLockES()
        especie=self.printEspecie()
        print(especie + " en casilla N= %d" % casillaN + " casilla M: %d" % casillaM + " se quiere mover a N= %d" % casillaRandom.getCasillaN() + " M: %d" % casillaRandom.getCasillaM())
        self.sabana.releaseLockES()

        self.sabana.getCasilla(self.casillaN,self.casillaM).acquireLock()#bloqueamos mutex de casilla origen
        self.sabana.getCasilla(self.casillaN,self.casillaM).desOcuparCasilla()#Desocupar casilla origen
        self.sabana.getCasilla(self.casillaN, self.casillaM).releaseLock()#desbloqueamos mutex casilla origen

        self.sabana.getCasilla(casillaRandom.getCasillaN(),casillaRandom.getCasillaM()).acquireLock()#Bloquear mutex de casilla destino y ocuparla
        self.sabana.getCasilla(casillaRandom.getCasillaN(),casillaRandom.getCasillaM()).ocuparCasilla(self)
        self.sabana.getCasilla(self.casillaN,self.casillaM).releaseLock()#desbloqueamos mutex casilla origen

        self.sabana.acquireLockES()
        especie=self.printEspecie()
        print(especie + " se ha movido a la nueva fila " + str(self.casillaN) + " y columna " + str(self.casillaM))
        self.sabana.releaseLockES()
    return
```

Ilustración 2: Captura tomada del código donde se muestra el método moverse del animal.

ETSII

2.2. Carnivoro

La clase hereda de la clase Animal y representará a los animales carnívoros de la simulación(leones y hienas), en primer lugar, destacaremos el método comer, los leones y hienas comen de la misma manera, sólo que, en situaciones diferentes, además, los leones comen hienas y cebras y las hienas sólo comen cebras.

Para comer, en primer lugar se obtienen las casillas adyacentes, recordar que el método getAdyacente nos devuelve una lista con las casillas libres y otra con las casillas ocupadas, si no hay adyacentes ocupadas el animal no podrá comer, por lo que se le manda a vivir, en caso contrario se elige una casilla random de la lista de casillas adyacentes ocupadas y se comprueba si se puede comer al animal elegido aleatoriamente, si no puede comérselo porque no cumple las condiciones se manda a vivir, en caso contrario se realiza lo siguiente(véase Ilustración 3):

- Bloqueamos el lock de la casilla destino, ponemos el animal a comer a muerto y desocupamos la casilla.
- Bloqueamos el lock de la casilla origen y la desocupamos
- Ocupamos casilla destino y liberamos locks de casillas origen y destino.
- Vemos la puntuación que le corresponde a la manada por comerse al animal(usamos método añadirPunto, nos devuelve un entero con la puntuación otorgada por comerse a un animal).
- Si el animal que come es un león, se bloquea el mutex de listaManadaLeones, se actualiza la puntuación de la manada y se desbloquea el mutex, en caso contrario se hace lo mismo con lista manadaHienas(método actualizarManada).
- Si el animal que se ha comido es una cebra se crea otro hilo con una cebra de la misma manada de la que murió(haciendo uso del método añadirAnimal de la clase sabana, nos devuelve una cebra ya posicionada).
- Bloquear mutex de actual ganador, actualizar el valor si es necesario y desbloquear mutex.(Véase Ilustración 4)

He de destacar que este método quizás sea el más complejo, tuve numerosos problemas a la hora de matar un animal, crear un hilo si se mataba a una cebra etc. Debido a que por cada cebra muerta se añade otra a la sabana de la misma manada, opté por no poner un join de la cebra creada, si lo hacía, el hilo que crea a la nueva cebra no avanza hasta que el hilo de la nueva cebra creada termine su ejecución(se supone que son dos animales distintos, no tiene por qué esperar uno por otro).

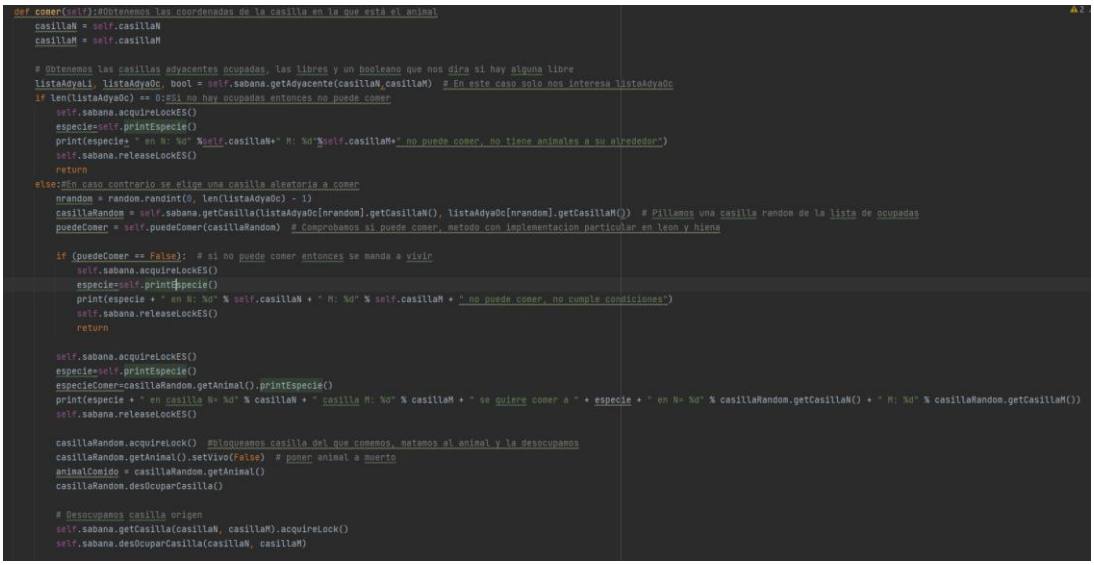
2.2.1 Hiena

Clase que hereda de la clase Carnivoro y Threading, en ella hay que destacar el método añadirPunto(recibe un animal a comer) que devuelve la puntuación que le corresponde al animal por haberse comido a otro.

El método puedeComer(recibe como parámetro una casilla a comer), las hienas sólo pueden comer si hay superioridad numérica de estas respecto a las cebras. El método actualizarManada, permite actualizar la puntuación de la manada tras comerse a un animal

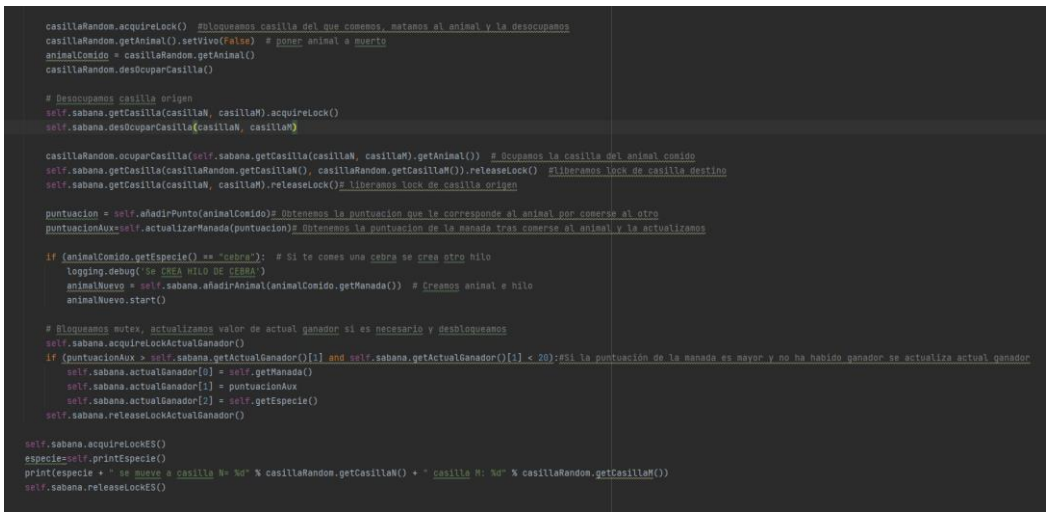
2.2.2 León

Clase muy parecida a la clase hiena, como diferencia notable destacar el método puedeComer ya que un león puede comer hienas y cebras, sólo come hienas si hay el mismo número o más de leones y hienas, siempre puede comerse a una cebra.



```
def comer(self):#Obtenemos las coordenadas de la casilla en la que está el animal
    casillaN = self.casillaN
    casillaM = self.casillaM

    # Obtenemos las casillas adyacentes ocupadas, las libres y un booleano que nos dira si hay alguna libre
    listaAdyaLi, listaAdyaOc, bool = self.sabana.getAdyacente(casillaN,casillaM)  # En este caso solo nos interesa listaAdyaOc
    if len(listaAdyaOc) == 0:#Si no hay ocupadas entonces no puede comer
        self.sabana.acquireLockES()
        especie=self.printEspecie()
        print(especie+ " en N: %d" %self.casillaN+" M: %d"%self.casillaM+" no puede comer, no tiene animales a su alrededor")
        self.sabana.releaseLockES()
        return
    else:#En caso contrario se elige una casilla aleatoria a comer
        nrandom = random.randint(0, len(listaAdyaOc) - 1)
        casillaRandom = self.sabana.getCasilla(listaAdyaOc[nrandom].getCasillaN(), listaAdyaOc[nrandom].getCasillaM())  # Pillamos una casilla random de la lista de ocupadas
        puedeComer = self.puedeComer(casillaRandom)  # Comprobamos si puede comer, metodo con implementacion particular en leon y hiena

        if (puedeComer == False):  # si no puede comer entonces se manda a vivir
            self.sabana.acquireLockES()
            especie=self.printEspecie()
            print(especie + " en N: %d" % self.casillaN + " M: %d" % self.casillaM + " no puede comer, no cumple condiciones")
            self.sabana.releaseLockES()
            return

        self.sabana.acquireLockES()
        especie=self.printEspecie()
        especieComer=casillaRandom.getAnimal().printEspecie()
        print(especie + " en casilla N= %d" % casillaN + " casilla M: %d" % casillaM + " se quiere comer a " + especie + " en N= %d" % casillaRandom.getCasillaN() + " M: %d" % casillaRandom.getCasillaM())
        self.sabana.releaseLockES()

        casillaRandom.acquireLock()  #bloqueamos casilla del que comemos, matamos al animal y la desocupamos
        casillaRandom.getAnimal().setVivo(False)  # poner animal a muerto
        animalComido = casillaRandom.getAnimal()
        casillaRandom.desOcuparCasilla()

        # Desocupamos casilla origen
        self.sabana.getCasilla(casillaN, casillaM).acquireLock()
        self.sabana.desOcuparCasilla(casillaN, casillaM)
```

Ilustración 3: Captura tomada del código que representa los primeros pasos a la hora de comerse un animal.



```
        casillaRandom.acquireLock()  #bloqueamos casilla del que comemos, matamos al animal y la desocupamos
        casillaRandom.getAnimal().setVivo(False)  # poner animal a muerto
        animalComido = casillaRandom.getAnimal()
        casillaRandom.desOcuparCasilla()

        # Desocupamos casilla origen
        self.sabana.getCasilla(casillaN, casillaM).acquireLock()
        self.sabana.desOcuparCasilla(casillaN, casillaM)

        casillaRandom.ocuparCasilla(self.sabana.getCasilla(casillaN, casillaM).getAnimal())  # Ocupamos la casilla del animal comido
        self.sabana.getCasilla(casillaRandom.getCasillaN(), casillaRandom.getCasillaM()).releaseLock()  #liberamos lock de casilla destino
        self.sabana.getCasilla(casillaN, casillaM).releaseLock()# liberamos lock de casilla origen

        puntuacion = self.añadirPunto(animalComido)# Obtenemos la puntuacion que le corresponde al animal por comerse al otro
        puntuacionAux=self.actualizarManada(puntuacion)# Obtenemos la puntuacion de la manada tras comerse al animal y la actualizamos

        if (animalComido.getEspecie() == "cebra"):  # Si te comes una cebra se crea otro hilo
            logging.debug('Se CREA HILO DE CEBRA')
            animalNuevo = self.sabana.añadirAnimal(animalComido.getManada())  # Creamos animal e hilo
            animalNuevo.start()

        # Bloqueamos mutex, actualizamos valor de actual ganador si es necesario y desbloqueamos
        self.sabana.acquireLockActualGanador()
        if (puntuacionAux > self.sabana.getActualGanador()[1] and self.sabana.getActualGanador()[1] < 20):#Si la puntuación de la manada es mayor y no ha habido ganador se actualiza actual ganador
            self.sabana.actualGanador[0] = self.getManada()
            self.sabana.actualGanador[1] = puntuacionAux
            self.sabana.actualGanador[2] = self.getEspecie()
        self.sabana.releaseLockActualGanador()

    self.sabana.acquireLockES()
    especie=self.printEspecie()
    print(especie + " se mueve a casilla N= %d" % casillaRandom.getCasillaN() + " casilla M: %d" % casillaRandom.getCasillaM())
    self.sabana.releaseLockES()
```

Ilustración 4: Captura tomada del código que representa los últimos pasos a la hora de comerse un animal.

Destacar también el método accionRealizar(devuelve la acción que realizará el animal) ya que un león es más propenso a descansar que los otros animales, el 60% de las veces descansará.

2.3. Herbivoro

La clase hereda de la clase Animal y representará a los animales herbívoros de la sabana(cebras), en esta clase hay que destacar la implementación del método comer ya que las cebras sólo comen hierba.(Véase Ilustración 5)

2.3.1 Cebra

Las cebras, se mueven como cualquier otro animal, comen hierba y descansan, son las 2º que más descansan.

2.4. Casilla

La clase Casilla define las casillas que forman la sabana(“plano” sobre el que se mueven los animales), tiene varios atributos:

```
def comer(self):#Una cebra come hierba
    self.sabana.acquireLockES()
    especie=self.printEspecie()
    print(especie+" en N: %d"%self.casillaN+" M: %d"%self.casillaM+" está comiendo hierba porque es herbivoro")
    self.sabana.releaseLockES()
```

Ilustración 5: Captura tomada del código en la que se representa como comen las cebras.

- ocupada: Atributo que nos indica si una casilla está ocupada o no.
- casillaN, casillaM: Indica las coordenadas en las que se encuentra la casilla.
- lock: Representa al mutex de la casilla.
- animal: Una casilla puede tener o no un animal

Para poder crear una casilla es necesario pasarle como parámetros las coordenadas N y M, se inicializarán lock, las coordenadas casillaN y casillaM y se pondrá ocupada a false(véase Ilustración 2).

A parte de los métodos propios de la programación orientada a objetos(getters y setters) tenemos otros de especial relevancia, se indican a continuación:

- ocuparCasilla: Método que sirve para ocupar una casilla.
- desOcuparCasilla: Método para desocupar la casilla.
- acquireLock: Nos permiten bloquear el lock.
- releaseLock: Nos permite liberar el lock.

2.5. Sabana

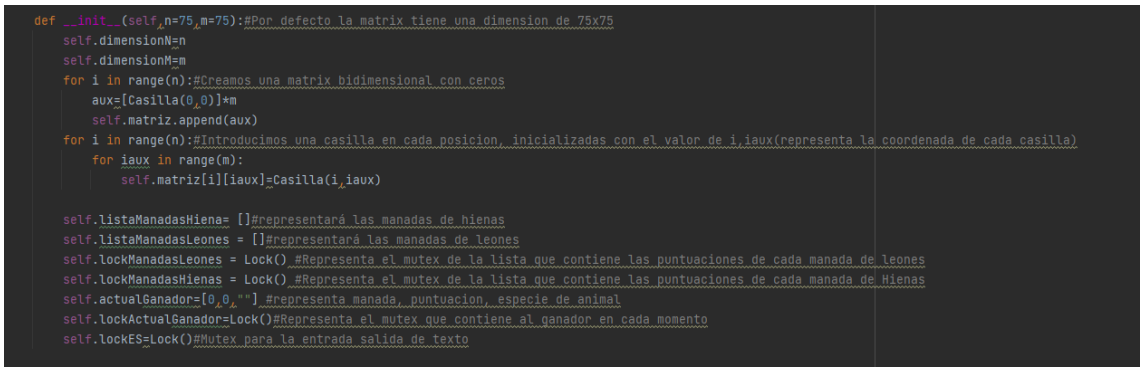
Dicha clase representará el lugar donde se encuentran los animales(donde viven), he definido las siguientes variables:

- matriz: representará la matriz de la sabana.
- dimensionN y dimensionM: Representarán las dimensiones M y N de la matriz.
- listaManadasHiena: Es una lista en la que se irán almacenando las puntuaciones de cada manada de los animales de especie hiena, siendo posición 0 la manada 0, posición 1 la manada 1…
- listaManadasLeones: Es una lista en la que se irán almacenando las puntuaciones de cada manada de los animales de especie león, siendo posición 0 la manada 0, posición 1 la manada 1…
- lockManadasHiena: representa el mutex de la lista de manadas de hiena.
- lockManadasLeones: Es el mutex de la lista de manadas de leones.
- actualGanador: Es una lista en la que se almacena la especie ganadora en cada momento, la manada y la puntuación.
- lockActualganador: Es el mutex del atributo “actualGanador”.
- lockES: Representará el mutex de la entrada y salida de texto por consola.

A continuación, comentaré los métodos más destacables de la clase.

2.5.1. Constructor

El método constructor recibe como parámetros la dimensión de la matriz(n y m, por defecto es 75), crea una matriz bidimensional con “0” y posteriormente se añade a cada posición una casilla inicializada a la posición de la matriz en la que se encuentra.(Véase Ilustración 6)



```
def __init__(self,n=75,m=75):#Por defecto la matrix tiene una dimension de 75x75
    self.dimensionN=n
    self.dimensionM=m
    for i in range(n):#Creamos una matrix bidimensional con ceros
        aux=[Casilla(0,0)]*m
        self.matriz.append(aux)
    for i in range(n):#Introducimos una casilla en cada posicion, inicializadas con el valor de i,iaux(representa la coordenada de cada casilla)
        for iaux in range(m):
            self.matriz[i][iaux]=Casilla(i,iaux)

    self.listaManadasHiena= []#representará las manadas de hienas
    self.listaManadasLeones = []#representará las manadas de leones
    self.lockManadasLeones = Lock() #Representa el mutex de la lista que contiene las puntuaciones de cada manada de leones
    self.lockManadasHienas = Lock() #Representa el mutex de la lista que contiene las puntuaciones de cada manada de Hienas
    self.actualGanador=[0,0,""] #representa manada, puntuacion, especie de animal
    self.lockActualGanador=Lock()#Representa el mutex que contiene al ganador en cada momento
    self.lockES=Lock()#Mutex para la entrada salida de texto
```

Ilustración 6: Captura de código en la que se muestra el constructor de la sabana.

2.5.2. GetAdyacente

El método recibe como parámetros las coordenadas de la casilla sobre la que se quieren las adyacentes, devuelve una lista con las casillas adyacentes ocupadas, una lista con las casillas adyacentes libres y un booleano que nos indica si hay casillas libres o no(véase Ilustración 7).

El método se utiliza tras comenzar el juego, es por ello que tenemos que hacer uso de los locks de cada casilla cada vez que la vayamos a revisar.

Como se puede observar en la Ilustración 3 se bloquea el mutex de la casilla, se añade a la lista correspondiente y se desbloquea. Destacar que en este método estaba mi principal problema ya que no desbloqueaba los locks en esta clase, sino en la clase Juego, eso me provocaba un interbloqueo entre los hilos.

2.5.3. AñadirAnimal

El método recibe como parámetro la manada de la cebra que se va a añadir al juego porque se han comido a otra de esa misma manada.

Como tenemos que revisar las casillas para encontrar una libre tenemos que hacer uso de los locks de cada una de ella y posteriormente liberarlos, una vez encontrada una casilla libre se crea una cebra, se le indica la manada a la que pertenece y se añade a la casilla, por último, se devuelve el animal.(Véase Ilustración 8)

2.5.4. IniciarSabana

Este método inicializará la sabana y devolverá una lista con los leones, otra con las hienas y otra con las cebras, devuelve también la cantidad de manadas que hay de cada especie.

En el enunciado se indica que por cada especie mínimo hay dos manadas, es por eso que supongo que mínimo habrá 2 leones por lo que tendremos 6 hienas y 12 cebras(20 animales en total), por lo que el mínimo de casillas ha de ser 20.

Destacar que si tenemos una matriz 5x5 el máximo número de animales es 20(2 leones máximo), si es de 6x6 el máximo número de animales es 30(3 leones máximo)…

En primer lugar, se elige de manera aleatoria la cantidad de leones, con ello se calculará la cantidad de hienas y cebras, posteriormente se eligen cuantas manadas habrán(Véase Ilustración 9).

Por último, por cada especie, se creará un animal, se añadirá a su lista correspondiente, se le asignará una manada y una casilla.(Véase Ilustración 10)

```
def getAdyacente(self,casillaN,casillaM):#Nos devuelve las casillas adyacentes a una coordenada d
    listaAdyaLi=[]
    listaAdyaOc= []
    bool=False
    for i in range(casillaN - 1, casillaN + 2, 1):
        for j in range(casillaM - 1, casillaM + 2, 1):
            if i == casillaN and j == casillaM:
                continue
            if i == self.dimensionN or j == self.dimensionM:
                break
            if i >= 0 and j >= 0:
                self.matriz[i][j].acquireLock()#bloqueamos casilla y añadimos animal a lista
                if(self.matriz[i][j].getOcupada()==True):
                    listaAdyaOc.append(self.matriz[i][j])
                if(self.matriz[i][j].getOcupada() ==False):
                    listaAdyaLi.append(self.matriz[i][j])
                    bool=True
                self.matriz[i][j].releaseLock()#desbloqueamos casilla

    return listaAdyaLi,listaAdyaOc,bool#Devuelve lista con casilals libres, lista conj ocupadas y
```

Ilustración 7: Captura tomada del código, representa el método con el que obtenemos las casillas adyacentes de una matriz.

```
def añadirAnimal(self,manada):#nos devuelve una cebra ya posicionada
    nAleatorio = random.randint(0, self.dimensionN - 1)
    mAleatorio = random.randint(0, self.dimensionM - 1)

    self.matriz[nAleatorio][mAleatorio].acquireLock()#Bloqueamos mutex casilla a mirar
    while (self.matriz[nAleatorio][mAleatorio].getOcupada() == True):#En este bucle buscamo
        self.matriz[nAleatorio][mAleatorio].releaseLock()#Desbloqueamos mutex de casilla qu
        nAleatorio = random.randint(0, self.dimensionN - 1)
        mAleatorio = random.randint(0, self.dimensionM - 1)
        self.matriz[nAleatorio][mAleatorio].acquireLock()#bloqueamos mutex de siguiente cas

    animal=Cebra("cebra")
    animal.setManada(manada)
    self.matriz[nAleatorio][mAleatorio].setAnimal(animal)
    self.matriz[nAleatorio][mAleatorio].ocuparCasilla(animal)
    self.matriz[nAleatorio][mAleatorio].releaseLock()#Desbloqueamos mutex de la casilla que

    return animal
```

Ilustración 8: Captura tomada del código, representa el método con el cuál añadimos una cebra a la sabana

```
iniciarSabana(self):#Inicializa la sabana creando y posicionando los animales
#Lista con los animales correspondientes a cada especie
lista_leones=[]
lista_hienas=[]
lista_cebras=[]
num_max_animales=self.dimensionN*self.dimensionM #Num max de animales
num_max_leones=int(num_max_animales/10) #La cantidad de leones es la parte entera de esa div
leonesRandom=random.randint(2,num_max_leones) #pilla un numero de leones logico, min 2 porque min 2 manadas
num_cebras=leonesRandom*6
num_hienas=leonesRandom*3
#el mínimo numero de manadas es 2
manadas_leones=random.randint(2,leonesRandom)-1
manadas_cebras=random.randint(2,num_cebras)-1
manadas_hienas=random.randint(2,num_hienas)-1

#casilla aleatoria
nAleatorio = random.randint(0, self.dimensionN - 1)
mAleatorio = random.randint(0, self.dimensionM - 1)
```

Ilustración 9: Dividiendo el método iniciarSabana en dos, esta ilustración representa la primera parte, en ella se define la cantidad de leones y con ello, de cebras y hienas, así como la cantidad de manadas de cada especie.

```
for i in range(0,leonesRandom):
    #Creamos un leon
    leon=Leon("leon")
    lista_leones.append(leon)
    #buscamos casilla desocupada y la ocupamos
    while(self.matriz[nAleatorio][mAleatorio].getOcupada()==True):
        nAleatorio = random.randint(0, self.dimensionN - 1)
        mAleatorio = random.randint(0, self.dimensionM - 1)
    leon.setManada(random.randint(0,manadas_leones))#Le asignamos una manada
    self.matriz[nAleatorio][mAleatorio].ocuparCasilla(leon)

nAleatorio = random.randint(0, self.dimensionN - 1)
mAleatorio = random.randint(0, self.dimensionM - 1)
```

Ilustración 10: 2º parte del método iniciarSabana, se asigna a cada animal una manada y una posición en la sabana

2.6. Main

En esta clase primeramente se pedirá por pantalla las dimensiones de la matriz de la sabana, por defecto se hará una matriz de 75x75, después se instanciará un objeto de tipo Sabana, posteriormente se inicializará la sabana(recordar que el método nos posicionaba los animales y devolvía una lista de animales de cada especie).

Una vez tenemos la sabana recorreremos cada lista de animales(devueltas por el método iniciarSabana) y los añadiremos a la lista de hilos, por último, empezamos la ejecución de cada uno de ellos y después los sincronizamos con un join.

A continuación, imprimimos por pantalla la manada ganadora con su correspondiente especie y el tablero final.

3. Conclusión

En esta práctica hemos visto como realizar una simple simulación de la sabana africana, en este caso cada hilo representa un animal de la simulación, simulación que se podría volver mucho más compleja si ponemos variables como que un animal necesite beber agua, que los animales tengan género y puedan reproducirse…

Destacar que se puede implementar con señales, por ejemplo, si un animal se quiere mover a una casilla ocupada puede esperar hasta que se desocupe, dicha implementación parece más compleja de lo que realmente es, no se ha incluido en esta entrega, pero tengo pensado añadir la funcionalidad para permitir la ejecución con condiciones y sin condiciones.

La práctica te permite conseguir una amplia visión sobre el funcionamiento de los hilos y de los mutex. Durante el desarrollo de la práctica he de destacar que la programación con hilos es algo compleja, hay que tener mucho cuidado con las variables que tengan los hilos en común, posibles interbloqueos…