Actividad 5 - El sistema Predador-Presa

Exactas Programa

Verano 2020

Los modelos de depredación y competencia forman parte de la batería de herramientas clásicas del ecólogo. Vito Volterra en Italia y Alfred Lotka en Estados Unidos fueron los precursores en este tema y crearon los modelos que, con diversas modificaciones y mejoras, seguimos usando hoy en día. El modelo de Volterra para depredación comienza suponiendo la existencia de dos poblaciones de animales, una de las cuales (el depredador) se alimenta de la otra (la presa). Se supone que las dos poblaciones están formadas por individuos idénticos, mezclados en el espacio.

La gran mayoría de los modelos estudiados tienen en cuenta principalmente la dinámica temporal de estos sistemas. Esta actividad tiene como objetivo implementar una versión diferente: Intentaremos analizar tanto el aspecto temporal como espacial del modelo. La idea es recrear un mundo que imaginaremos como un valle rodeado de montañas en el que existen depredadores (que llamaremos *leones* e identificaremos con la letra `L`) y presas (que llamaremos *antílopes* e identificaremos con la letra `A`). Este valle será bidimensional, y lo representaremos por medio de un tablero rectangular similar al que utilizamos en el desafío anterior para modelar el fenómeno de avalanchas.

1. Defina la función `crear_tablero(filas,columnas)`, que genere este valle, tomando como base la función `crear_tablero` del desafío del arenero. Las diferencias principales son:

   - Como el valle no necesariamente es cuadrado, vamos a tomar como parámetros la cantidad de filas y columnas de nuestro tablero. Prestar atención a que, como en el desafío anterior, debemos tener en cuenta los bordes (Es decir, el *array* que utilizaremos tendrá dimensiones un poco más grandes que `filas`$\times$`columnas`. ¿Cuáles serán?)
   - Las posiciones vacías del tablero, en vez de tener un 0, deben tener un espacio (`" "`).
   - En vez de estar notados con un -1, los bordes del tablero tendrán montes (representados por la letra `M`).

   Una vez definida `crear_tablero(m,n)`, cree un tablero de 3×4 haciendo `t = crear_tablero(3,4)`.

2. Agreguemos algunos animales a nuestro tablero. Ubique 4 (cuatro) antílopes (`A`), uno en cada una de las siguientes posiciones: $(1,3)$, $(2,1)$, $(2,3)$ y $(3,1)$ de `t`. Luego, agregue 1 (un) león (`L`) en la posición $(1,1)$.

   Para no tener que agregar estos animales haciendo las asignaciones una por una, podemos crear algunas listas y recorrerlas por medio de un ciclo `for`:

```
# definimos la coordenadas de los animales
filas = [1, 2, 2, 3, 1]
columnas = [3, 1, 3, 1, 1]
animal = ["A", "A", "A", "A", "L"]
# y ahora los asignamos dentro del tablero
for i in range(len(animal)):
  t[(filas[i], columnas[i])] = ___COMPLETAR___
```

   Usaremos estos animales para testear las funciones que vayamos armando.

3. Vamos a necesitar una función que permita identificar el vecindario de una posición. A diferencia del caso de la avalanca, esta vez vamos a incluir las diagonales. Además, vamos a precisar que las

celdas vecinas se listen en el siguiente orden, que es el de las agujas del reloj comenzando arriba a la izquierda:

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & (i,j) & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

Implemente una función `vecinos_de(tablero, coord)` que tome como entrada un tablero y las coordenadas de una posición (que, recordemos, es un par `(i,j)`), y devuelva una lista con las coordenadas de sus posiciones vecinas, en el orden solicitado. **Recuerde que los bordes no son vecinos válidos de una posición**.

4. Para poder replicar la dinámica del modelo, requerimos identificar si hay un animal o una celda vacía en el vecindario de otra. Implemente una función `buscar_adyacente(tablero, coord, objetivo)`, que tome un tablero, las coordenadas a partir de donde se quiera buscar, y aquello que se quiere buscar (que puede ser `L`, `A` o un espacio en blanco `" "`), y devuelva una lista que contenga al primero (según el orden de `vecinos_de`) de los vecinos de la celda `coord` que sea del tipo buscado. Si no hay vecinos de la coordenada `coord` con el elemento buscado, la lista debe estar vacía.

   Por ejemplo, si tomamos tablero `t`:

"M"	"M"	"M"	"M"	"M"	"M"
"M"	"L"	" "	"A"	" "	"M"
"M"	"A"	" "	"A"	" "	"M"
"M"	"A"	" "	" "	" "	"M"
"M"	"M"	"M"	"M"	"M"	"M"

Cuadro 1: Mundo de ejemplo

   y preguntamos:

   - `buscar_adyacente(t, (1, 2), "L")`, nos debe dar como respuesta: `[(1, 1)]`.
   - `buscar_adyacente(t, (1, 1), "A")`, nos debe dar como respuesta: `[(2, 1)]`.
   - `buscar_adyacente(t, (1, 1), " ")`, nos debe dar como respuesta: `[(1, 2)]`.
   - `buscar_adyacente(t, (2, 2), "A")`, nos debe dar como respuesta: `[(1, 3)]`.
   - `buscar_adyacente(t, (3, 3), "L")`, nos debe dar como respuesta: `[]`.

5. Estamos en condiciones de definir las funciones que llevarán a cabo la dinámica del sistema: Defina `mover(tablero, coord)`, que dado el tablero y un par de coordenadas, se fija si en esas coordenadas hay un animal, y en ese caso lo mueve a la primera posición vecina que esté vacía (O lo deja en su lugar, si no hay ninguna).

6. Defina `alimentar(tablero, coord)`, que dado el tablero y un par de coordenadas, verifica si esa posición está ocupada por un león y, en tal caso, busca al primer antílope disponible en la vecindad. De encontrarlo, el león se mueve a la posición ocupada por el antílope, y lo devora (el antílope desaparece del tablero y aparece en la panza del león).

7. Defina `reproducir(tablero, coord)`, que dado el tablero y un par de coordenadas, verifica si esa posición está ocupada por un animal. En caso de estarlo, busca en la vecindad un individuo de la misma especie y, si lo encuentra, busca la primer posición vacía, haciendo aparecer en esa posición a un nuevo individuo de la misma especie (león o antílope, según corresponda). Sólo se genera un nuevo individuo por reproducción.

8. Ahora aplicamos cada una de estas acciones a todo el tablero, de manera análoga a `desbordar_arenero` en el desafío del arenero. La idea es recorrer cada posición del tablero y realizar la acción requerida. Defina `fase_mover(tablero)`, que implementa la fase de movimiento recorriendo cada posición (i, j)

del tablero y aplicando `mover(tablero, (i,j))`, `fase_alimentacion(tablero)`, que recorre el tablero aplicando `alimentar(tablero, (i,j))`, y `fase_reproduccion(tablero)`, que hace lo mismo aplicando `reproducir(tablero, (i,j))`.

Defina `evolucionar(tablero)`, que tome un tablero y lo haga pasar por una fase de alimentación, una de reproducción y una de movimiento, en ese orden.

Como ejemplo, realicemos un ciclo de evolución a nuestro tablero `t`. Partiendo de la configuración original, obtenemos:

"M"	"M"	"M"	"M"	"M"	"M"
"M"	"L"	" "	"A"	" "	"M"
"M"	"A"	" "	"A"	" "	"M"
"M"	"A"	" "	" "	" "	"M"
"M"	"M"	"M"	"M"	"M"	"M"

Estado inicial

"M"	"M"	"M"	"M"	"M"	"M"
"M"	" "	" "	"A"	" "	"M"
"M"	" "	" "	"A"	" "	"M"
"M"	"L"	" "	" "	" "	"M"
"M"	"M"	"M"	"M"	"M"	"M"

Luego de alimentación

"M"	"M"	"M"	"M"	"M"	"M"
"M"	" "	"A"	"A"	"A"	"M"
"M"	" "	" "	"A"	"A"	"M"
"M"	"L"	" "	"A"	"A"	"M"
"M"	"M"	"M"	"M"	"M"	"M"

Luego de reproducción

"M"	"M"	"M"	"M"	"M"	"M"
"M"	"A"	"A"	"A"	"A"	"M"
"M"	"L"	"A"	"A"	"A"	"M"
"M"	" "	" "	" "	" "	"M"
"M"	"M"	"M"	"M"	"M"	"M"

Estado final (Luego de movimiento)

9. Defina `evolucionar_en_el_tiempo(tablero, k)`, que realiza la evolución del tablero k veces.

10. Ya tenemos toda la dinámica programada. Vamos a probarla con un tablero más grande y con distintas cantidades de leones y antílopes ubicados al azar.

    Hay muchas formas de realizar la asignación inicial al azar de casillas. Un ejemplo es el siguiente:

```
def mezclar_celdas(tablero):
  celdas = []
  for i in range (1 , ___COMPLETAR___) :
    for j in range (1 , ___COMPLETAR___) :
      celdas.append((i, j))
  random.shuffle(celdas)
  return celdas
```

    Cada vez que llamemos a `mezclar_celdas(tablero)`, vamos a obtener una lista de las celdas del tablero mezcladas en un orden distinto. Luego, para completar el tablero, vamos tomando los primeros elementos de la lista con `pop(0)` y asignándoles un leon o un antílope, según necesitemos.

11. Realizar la función `generar_tablero_azar(filas, columnas, n_antilopes, n_leones)`, que toma las cantidades expresadas y devuelve un tablero al azar siguiendo las especificaciones. Probar la función generando un tablero de 10×10 con cinco leones y diez antílopes.

12. Podemos hacer una visualización gráfica de la evolución de nuestro valle. Para esto, es muy importante que todas las funciones tengan los nombres que nosotros les dimos en esta guía. El archivo con las definiciones de las funciones debe llamarse `predpresa.py`.

    Descomprima el archivo `visualizacion.zip` en la misma carpeta que su proyecto.

    Para llamar a la función `simular(x, y, cant_antilopes, cant_leones, ciclos_de_evolucion)`, cree un nuevo archivo `Python` con el siguiente contenido:

```
import visualizacion
import predpresa

visualizacion.simular(8, 6, 5, 2, 10)
```

Ejecute dicho archivo.

Pruebe hacer otras simulaciones. La función `simular` toma 5 parámetros: La cantidad de filas y columnas del tablero, la cantidad de antílopes y leones iniciales, y la cantidad de ciclos de la evolución.

Requisito: La visualización usa como base en el módulo `pygame` de `Python`. Si no lo tiene, deberá instalarlo antes de poder correrla. Para ello, en GNU+Linux abrir una terminal o en Windows la aplicacion `Anaconda Prompt`; y ahí escribir `pip install pygame` y apretar [Enter]. En Ubuntu se puede instalar usando el comando `pip3 install pygame` en la consola. La descarga e instalación dura menos de dos minutos.

13. **Optativo:** El modelo de Lotka y Volterra se enfoca en analizar cómo van variando las poblaciones de los depredadores y las presas a través del tiempo. Incorporemos ese análisis en nuestro programa:

    a) Definir `cuantos_de_cada(tablero)`, que dado un tablero devuelve una lista con dos elementos: el primero corresponde a la cantidad de antílopes y el segundo a la cantidad de leones que hay en el tablero. Por ejemplo, si devuelve $[14, 3]$, indica que hay 14 antílopes, y 3 leones.

    b) Definir `registrar_evolucion(tablero, k)`, función cuyo comportamiento deberá ser como el de `evolucionar_en_el_tiempo(tablero, k)` pero que además de hacer evolucionar el tablero, registre la cantidad total de individuos de cada especie en cada paso. La función debe armar una lista con los resultados `cuantos_de_cada` en cada uno de lo k pasos de la evolución del tablero. Por ejemplo, en una evolución de tres pasos podría dar algo como: $[[5, 1], [3, 1], [7, 1]]$, indicando que primero habían 5 antílopes y un león, luego 3 y 1, y finalmente 7 y 1.

    c) Vamos a guardar el resultado de `registrar_evolucion(tablero, k)` en un archivo cuyo nombre será `predpres.csv`. Si guardamos la lista con el resultado de `registrar_evolucion` en la variable `evolucion_especies`, podemos guardarla en un archivo `csv` haciendo:

```
#necesitamos importar el modulo csv
import csv

evolucion_especies = ___COMPLETAR___
with open("predpres.csv", "w", newline="") as csvfile:
  csv_writer = csv.writer(csvfile)
  csv_writer.writerow(["antilopes", "leones"])
  csv_writer.writerows(evolucion_especies)
```

    d) Podemos observar la evolución de las poblaciones por medio de un gráfico. Vamos a usar `NumPy` y `PyPlot` para graficar la cantidad de individuos de cada especie a partir del archivo `csv` que creamos, escribiendo:

```
valores = np.loadtxt('predpres.csv',delimiter=',',skiprows=1)
plt.ylabel('Cantidad de Individuos')
plt.xlabel('Ciclo')
plt.plot(valores[:,0], label = 'antilopes')
plt.plot(valores[:,1], label = 'leones')
plt.legend()
```

14. **Optativo+:** Definir `buscar_adyacente_aleatoria(tablero, i, j, objetivo)`, que tenga el mismo funcionamiento que `buscar_adyacente` pero que devuelva de manera aleatoria una posición posible que cumpla con el criterio de búsqueda. Actualizar las funciones de las distintas etapas para que utilicen esta nueva función.

15. **Optativo++:** Para hacer la representación de nuestro mundo más interesante y realista, vamos a suponer que es un toroide. Esto significa que al desplazarse más allá del la última columna de la derecha, el individuo aparecería en la primera columna de la izquierda. Lo mismo ocurre entre la última y la primer fila del tablero. Deberá modificar la generación del tablero para que las casillas

de borde marcadas con `M` no estén más y adecuar la función `mis_vecinos` para que tome en cuenta la nueva geometría del mundo.

Pista: El resto de dividir por la cantidad de filas o columnas proporciona una forma de *dar la vuelta* al mundo. Para pensarlo.