Modelo Reto

Instrucciones

Solución

Importar librerías

Se importan las librerías necesarias para el entorno y el agente.

In [ ]:
```
# Librerías para el agente y modelo
import agentpy as ap
import numpy as np
import sys
import json
from numpyencoder import NumpyEncoder
from collections import Counter

# Librearías de visualización y otros
import matplotlib.pyplot as plt
import IPython

# Dar a matplotlib el formato de diseño
plt.style.use("ggplot")
```

Definición de la variable donde vamos a guardar toda la información para después correrla sobre THREE.js

In [ ]:
```
information = {
    'frames': [],
    'traffic_lights_number': None,
    'steps': None,
    'cars_number': None,
    'cars': None,
    'traffic_lights': None,
}
```

Definición de las clases

Se define la clase agente `TrafficLight` , la cuál se encarga de simular a los semáforos, y la clase de modelo `TrafficLightModel` , la cuál se encarga de simular el entorno.

In [ ]:
```
class TrafficLight(ap.Agent):
    def setup(self):
        self.state = 0
        self.states = ["green", "yellow", "red"]
        self.current_state = None
```

```
        self.update_state()
        self.time = 0
        self.time_limits = self.p.traffic_times

        # Tiempo que dura cada paso de la simulación, 24fps -> 1/24
        self.step_time = 1 / 24

        # Dirección a la que apunta el semáforo
        self.direction = np.array([0, 1])

    def step(self):
        """
        Operaciones que realiza el semáforo con cada iteración de la simulación
        """
        # Aumentar el tiempo que lleva corriendo
        self.time += self.step_time

        # Cuando se llegue a los límites establecidos para la luz actual, cambiar de es
        if self.time >= self.time_limits[self.state] and self.state < 2:
            self.time = 0
            self.state = (self.state + 1) % 3
            self.update_state()

    def update_state(self):
        """
        Actualiza el estado del semáforo
        """
        self.current_state = self.states[self.state]

    def change_state(self, state: int):
        """
        Actualiza el estado del semáforo
        """
        self.current_state = self.states[state]
        self.state = state
        self.time = 0

    def total_waiting_time(self):
        """
        Regresa el tiempo total que los carros han esperado
        """
        cars_in_traffic_light = [car.waiting_time for car in self.model.cars[self.p.car
        return np.mean(cars_in_traffic_light)

    def vote(self, times: list) -> int:
        """
        Regresa el estado de la luz a partir de una lista de tiempos
        """
        return np.argmax(np.array(times))
```

In [ ]:

```
class Car(ap.Agent):
    """
    Clase que define al auto dentro del ambiente
    """

    def setup(self):
        """
        Método que define la manera de inicializar un nuevo auto dentro de la simulació
```

```
        """
        # Tiempo que dura cada paso de la simulación
        self.step_time = 1 / 24

        # Dirección a la que viaja el auto
        self.direction = np.array([1, 0])

        # Velocidad en metros por segundo
        self.speed = 0

        # Máxima velocidad en metros por segundo
        self.max_speed = np.random.normal(loc=4.0, scale=0.5, size=1)[0]

        # Estados del carro: 1 = ok, 0 = dead (choque)
        self.state = 1

        # Carril
        self.lane = 0

        # Tiempo que llega de espera
        self.waiting_time = 0

    def update_position(self):
        """
        Método que se usa para actualizar la posición del auto dentro del plano
        """

        # Verifica si el auto no ha chocado
        if self.state == 0:
            return

        # Actualiza la posición según la velocidad actual
        self.model.avenue.move_by(self, np.multiply(self.speed, self.direction))

    def update_speed(self):
        """
        Método que se usa para actualizar la velocidad a la que viaja el auto dentro de
        """

        # Verifica si el auto no ha chocado
        if self.state == 0:
            return

        # Obten la distancia más pequeña a uno de los autos que vaya en la misma direcc
        my_position = self.model.avenue.positions[self]

        # Inicializar una distancia muy grande, y después para cada uno de los autos ve
        #   y el carril por donde viajan
        min_car_distance = sys.maxsize

        for car in self.model.cars:
            if car != self and car.lane == self.lane:
                # Verifica si el carro va en la misma dirección
                dot_p1 = np.dot(self.direction, car.direction)

                # Verifica si el carro está atrás o adelante
                other_position = self.model.avenue.positions[car]
                dot_p2 = np.dot(np.array(
                    [[other_position[0] - my_position[0]], [other_position[1] - my_posi
```

```
            # Verifica si el carro está en el mismo carril, misma dirección y delan
            if dot_p1 > 0 and dot_p2 > 0:
                distance = np.linalg.norm(other_position - my_position)

                # Actualizar la distancia si es muy corta
                if distance < min_car_distance:
                    min_car_distance = distance

        # Obten la distancia al próximo semáforo
        min_trafficLight_distance = sys.maxsize
        trafficLight_state = 0
        for light in self.model.trafficAgents:

            # Verifica si el semáforo apunta hacia el vehículo usando el producto punto
            dot_p1 = np.dot(light.direction, self.direction)

            # Verifica si el semáforo está adelante o atrás del vehículo
            light_position = self.model.avenue.positions[light]
            dot_p2 = np.dot(np.array([[light_position[0] - my_position[0]],
                                      [light_position[1] - my_position[1]]]).T, self.di

            if dot_p1 < 0 and dot_p2 > 0:
                distance = np.linalg.norm(light_position - my_position)

                if min_trafficLight_distance > distance:
                    min_trafficLight_distance = distance
                    trafficLight_state = light.state

        ###################
        # Actualiza la velocidad del auto dependiendo de su cercanía con ciertos objeto

        # el carro ha chocado
        if min_car_distance < 2:
            self.speed = 0
            self.state = 1

        # los carros están muy cerca
        elif min_car_distance <= 5:
            self.speed = np.maximum(self.speed - 200 * self.step_time, 0)

        # los carros están algo cerca
        elif min_car_distance <= 15:
            self.speed = np.maximum(self.speed - 80 * self.step_time, 0)

        # hay luz amarilla muy cerca
        elif min_trafficLight_distance < 15 and trafficLight_state == 1:
            self.speed = np.minimum(self.speed + 30 * self.step_time, self.max_speed)

        # hay luz amarilla
        elif min_trafficLight_distance < 40 and trafficLight_state == 1:
            self.speed = np.maximum(self.speed - 50 * self.step_time, 0)

        # hay un alto
        elif min_trafficLight_distance <= 25 and trafficLight_state == 2:
            self.speed = np.maximum(self.speed - 300 * self.step_time, 0)

        # hay un alto a lo lejos
        elif min_trafficLight_distance < 35 and trafficLight_state == 2:
            self.speed = np.maximum(self.speed - 50 * self.step_time, 0)
```

```
        else:
            self.speed = np.minimum(
                self.speed + 5 * self.step_time, self.max_speed)

        # comprobar si el carro está avanzando o está parado
        if self.speed == 0:
            self.waiting_time += 1
        else:
            self.waiting_time = 0
```

In [ ]:

```
class StreetModel(ap.Model):
    def setup(self):
        global information

        # global time
        self.global_time = 0

        # definir los semáforos dentro de la simulación
        self.trafficAgents = ap.AgentList(self, self.p.traffic_lights, TrafficLight)
        self.trafficAgents.step_time = self.p.step_time
        self.trafficAgents.state = ap.AttrIter([0, 0, 2, 2])
        self.trafficAgents.update_state()
        self.trafficAgents[0].direction = np.array([0, 1])
        self.trafficAgents[1].direction = np.array([0, -1])
        self.trafficAgents[2].direction = np.array([1, 0])
        self.trafficAgents[3].direction = np.array([-1, 0])

        # definir los autos dentro de la simulación
        self.cars = ap.AgentList(self, self.p.cars, Car)
        self.cars.step_time = self.p.step_time
        self.cars.lane = ap.AttrIter(self.p.lanes)

        # Definir la posición de inicio de los autos
        cars_per_direction = self.p.cars_per_direction
        cars_per_direction_index = self.p.cars_per_direction_index

        for iter, direction, cpd, cpdi in zip(range(4), [[0, -1], [0, 1], [-1, 0], [1,
            for k in range(cpd):
                self.cars[k + (cpdi - cpd + 1)].direction = direction

        # Inicializa el entorno
        self.avenue = ap.Space(self, shape=[self.p.size, self.p.size], torus=True)

        # Agrega los semáforos al entorno
        self.avenue.add_agents(self.trafficAgents, random=True)
        self.avenue.move_to(self.trafficAgents[0], [self.p.size * 0.5 - 11.5, self.p.si
        self.avenue.move_to(self.trafficAgents[1], [self.p.size * 0.5 + 11.5, self.p.si
        self.avenue.move_to(self.trafficAgents[2], [self.p.size * 0.5 - 11.5, self.p.si
        self.avenue.move_to(self.trafficAgents[3], [self.p.size * 0.5 + 11.5, self.p.si

        # Agrega los autos al entorno
        lane_positions = [0, -11.1, -8.3, -5.5, 0, 5.5, 8.3, 11.1, 0][::-1]
        self.avenue.add_agents(self.cars, random=True)

        for iter, cpd, cpdi in zip(range(4), cars_per_direction, cars_per_direction_ind
            for k in range(cpd):
                if iter == 0:
```

```
                # carros que van bajando
                self.avenue.move_to(self.cars[k + (cpdi - cpd + 1)], [lane_position
            elif iter == 1:
                # carros que van subiendo
                self.avenue.move_to(self.cars[k + (cpdi - cpd + 1)], [lane_position
            elif iter == 2:
                # carros que van a la derecha
                self.avenue.move_to(self.cars[k + (cpdi - cpd + 1)], [15 * (k + 1),
            elif iter == 3:
                # carros que van a la izquierda
                self.avenue.move_to(self.cars[k + (cpdi - cpd + 1)], [self.p.size -



    # save information of the agents of the model
    information['cars'] = [{'id': car.id, 'lane': car.lane} for car in self.cars]
    information['traffic_lights'] = [{'id': light.id, 'state': light.state} for lig

def step(self):
    self.trafficAgents.step()
    self.cars.update_position()
    self.cars.update_speed()

    # print(self.trafficAgents.total_waiting_time())

    # aumentar el tiempo global
    self.global_time += 1 * self.p.step_time

    # realizar la votación
    if self.global_time > self.p.traffic_times[-1]:
        candidates = list(np.multiply(np.array(self.trafficAgents.total_waiting_tim
        winner = int(Counter(self.trafficAgents.vote(candidates)).most_common(1)[0]


        if winner in [0, 1]:
            self.trafficAgents[0].change_state(0)
            self.trafficAgents[1].change_state(0)
        else:
            self.trafficAgents[2].change_state(0)
            self.trafficAgents[3].change_state(0)

        self.global_time = 0

def update(self):
    global information

    frame_info = {
        'cars': [
            {
                'id': car.id,
                'x': self.avenue.positions[car][0] - 500,
                'y': self.avenue.positions[car][1] - 500
            } for car in self.cars
        ],
        'lights': [
            {
                'id': light.id,
                # 'x': self.avenue.positions[light][0] - 500,
                # 'y': self.avenue.positions[light][1] - 500,
                'state': light.state
```

```
                } for light in self.trafficAgents
            ],
        }

        information['frames'].append(frame_info)

    def end(self):
        pass
```

Simulación

Se definen los parámetros, se crea un objeto de la clase `TrafficLightModel` , y se ejecuta la simulación.

In [ ]:

```
def animation_plot_single(model, ax):
    ax.set_title(f"Avenida t={model.t * model.p.step_time:.2f}")

    colors = ["green", "yellow", "red"]

    pos_s1 = model.avenue.positions[model.trafficAgents[0]]
    ax.scatter(*pos_s1, s=20, c=colors[model.trafficAgents[0].state])

    pos_s2 = model.avenue.positions[model.trafficAgents[1]]
    ax.scatter(*pos_s2, s=20, c=colors[model.trafficAgents[1].state])

    pos_s3 = model.avenue.positions[model.trafficAgents[2]]
    ax.scatter(*pos_s3, s=20, c=colors[model.trafficAgents[2].state])

    pos_s4 = model.avenue.positions[model.trafficAgents[3]]
    ax.scatter(*pos_s4, s=20, c=colors[model.trafficAgents[3].state])

    ax.set_xlim(0, model.avenue.shape[0])
    ax.set_ylim(0, model.avenue.shape[1])

    for car in model.cars:
        pos_c = model.avenue.positions[car]
        ax.scatter(*pos_c, s=5, c="black")

    ax.set_axis_off()
    ax.set_aspect('equal', 'box')


def animation_plot(m, p):
    fig = plt.figure(figsize=(10, 10))
    ax = fig.add_subplot(111)
    animation = ap.animate(m(p), fig, ax, animation_plot_single)
    return IPython.display.HTML(animation.to_jshtml(fps=24))
```

In [ ]:

```
parameters = {
    'step_time': 1 / 24,                    # Paso de tiempo
    "traffic_lights": 4,                    # Número de semáforos
    "traffic_times": [4, 2, 6],             # Tiempos de cada estado (green, yellow, re
    "steps": 2000,                          # Número de pasos
    "size": 1000,                           # Tamaño del entorno
    "cars_per_direction": [50, 35, 5, 5],   # número de carros por dirección
    "lane_size": 20,                        # tamaño del carril
```

```
    "lanes": []
}
parameters['lanes'].extend(np.random.randint(1, 3+1, int(parameters['cars_per_direction
parameters['lanes'].extend(np.random.randint(5, 7+1, int(parameters['cars_per_direction
parameters['lanes'].extend(np.random.randint(5, 7+1, int(parameters['cars_per_direction
parameters['lanes'].extend(np.random.randint(1, 3+1, int(parameters['cars_per_direction
parameters['cars'] = sum(parameters['cars_per_direction'])
temp = 0
temp_array = []
for i in range(4):
    temp_array.append(parameters['cars_per_direction'][i] + temp - 1)
    temp += parameters['cars_per_direction'][i]
parameters['cars_per_direction_index'] = temp_array
```

In [ ]:

```
model = StreetModel(parameters)
```

In [ ]:

```
# guardar los parámetros dentro de la variable de información
information['traffic_lights_number'] = parameters['traffic_lights']
information['steps'] = parameters['steps']
information['cars_number'] = parameters['cars']
```

In [ ]:

```
results = model.run()
```

```
Completed: 2000 steps
Run time: 0:01:53.801689
Simulation finished
```

In [ ]:

```
with open('./Simulation/data.json', 'w') as file:
    json.dump(information, file, indent=0, sort_keys=True, separators=(',', ':'), ensur
```

In [ ]:

```
# animation_plot(StreetModel, parameters)
```

In [ ]:

```

```