



UNIVERSIDAD  
POLITÉCNICA  
DE MADRID



## **INTELIGENCIA ARTIFICIAL - MUAR 2023/2024**

**GESTIÓN DEL TRÁFICO VEHICULAR Y PEATONAL**

**MEDIANTE EL USO DE CONTROLADORES BORROSOS**

David Redondo Quintero - Pablo Vela Silva

Carlos Y. Prudencio Torrico – Shota Nakvetauri

Universidad Politécnica de Madrid

15-01-2024

# ÍNDICE

1. INTRODUCCIÓN Y OBJETIVOS
2. HERRAMIENTAS UTILIZADAS
  - 2.1. SUMO
  - 2.2. NETEDIT
  - 2.3. LÓGICA BORROSA EN PYTHON
3. MAPA 1: AVENIDA CON PASO DE CEBRA
  - 3.1. DISEÑO DEL MODELO Y CREACIÓN DEL MAPA
  - 3.2. DISEÑO E IMPLEMENTACIÓN DEL CONTROLADOR BORROSO
    - 3.2.1. DEFINICIÓN DE ENTRADAS Y SALIDAS
    - 3.2.2. DISEÑO DE LAS VARIABLES Y VALORES LINGÜÍSTICOS, CONJUNTOS Y REGLAS FUZZY
    - 3.2.3. DESARROLLO DEL SCRIPT DE CONTROL
  - 3.3. RESULTADOS Y CONCLUSIONES. COMPARATIVA CON MODELO CLÁSICO DE SEMÁFOROS TEMPORIZADOS
4. MAPA 2: RED CON MÚLTIPLES INTERSECCIONES
  - 4.1. DISEÑO DEL MODELO Y CREACIÓN DEL MAPA
  - 4.2. DISEÑO E IMPLEMENTACIÓN DEL CONTROLADOR BORROSO
    - 4.2.1. DEFINICIÓN DE ENTRADAS Y SALIDAS
    - 4.2.2. DISEÑO DE LAS VARIABLES Y VALORES LINGÜÍSTICOS, CONJUNTOS Y REGLAS FUZZY
    - 4.2.3. DESARROLLO DEL SCRIPT DE CONTROL
  - 4.3. RESULTADOS Y CONCLUSIONES. COMPARATIVA CON MODELO CLÁSICO DE SEMÁFOROS TEMPORIZADOS
5. COMPARATIVA DEL CONTROLADOR FUZZY CON ALGORITMO GENÉTICO
  - 5.1. DISEÑO DEL ALGORITMO GENÉTICO
  - 5.2. ESTADÍSTICAS DEL ENTRENAMIENTO Y DESEMPEÑO
6. DISTRIBUCIÓN DEL TRABAJO Y REFERENCIAS
7. ANEXO. INSTALACIÓN E INSTRUCCIONES DE USO DEL PROGRAMA

## **1. INTRODUCCIÓN Y OBJETIVOS**

El objetivo del trabajo es diseñar un modelo inteligente de gestión del tráfico. Los atascos son un problema recurrente en las grandes ciudades, y una posible solución es el desarrollo de un sistema que permita optimizar el tiempo de espera de vehículos y peatones, dando prioridad a las avenidas con mayor flujo.

Para el control del tráfico real se realizan estudios estadísticos que determinan el flujo habitual de vehículos en cada calle de una ciudad, de forma que se ajustan los tiempos de los semáforos en función de esos datos.

En este trabajo se busca dar un paso más allá, creando un sistema global que permita saber cuántos vehículos y peatones hay en cada calle en tiempo real, de forma que si en algún momento se invierte el flujo el sistema sea capaz de reaccionar y modificar en consecuencia los tiempos de espera de cada semáforo. Para lograrlo, las carreteras y aceras deben estar sensorizadas, informando así del número de usuarios en cada tramo o calle. Somos conscientes de que esta premisa supone una dificultad grande a la hora de implementar este sistema en ciudades reales, ya que supondría colocar cámaras y sensores en cada calle. Por tanto, el trabajo consistirá en recrear un mundo con unas pocas vías, en el que se pueda apreciar cómo se regula automáticamente el tráfico según surgen vehículos y peatones en distintos puntos, y posteriormente comparar los resultados obtenidos con los del modelo clásico de semáforos temporizados.

Como es de esperar, en situaciones con un elevado número de usuarios será imposible evitar las esperas. Si hay muchos vehículos circulando por una avenida y muchos peatones que quieren cruzar es evidente que o los unos o los otros tendrán que esperar. Sin embargo, se espera que el sistema implementado funcione muy bien en situaciones con un bajo número de usuarios, como por ejemplo el tráfico nocturno, ya que se programará de tal forma que nunca se interrumpa el avance de los vehículos que circulen por grandes arterias de la ciudad en el caso de que no haya ningún otro usuario que quiera cruzar. Este comportamiento supone una mejora en los tiempos de espera globales con respecto a los sistemas tradicionales, en los que es habitual que los vehículos tengan que pararse aunque no haya otro vehículo o peatón cruzando.

## **2. HERRAMIENTAS UTILIZADAS**

Para poder poner en práctica la idea de este proyecto es necesario disponer de una herramienta o aplicación que permita diseñar redes de carreteras simples, así como de un mecanismo de control sencillo pero al mismo tiempo suficientemente robusto.

## 2.1. SUMO (Simulation of Urban Mobility)

Se trata de un paquete de simulación de tráfico continuo y de código abierto diseñado por el Instituto de Transporte del Centro Aeroespacial Alemán con la idea de manejar redes de carreteras. Permite la simulación intermodal, incluidos los peatones, e incluye un amplio conjunto de herramientas para la creación de escenarios. Presenta una gran cantidad de funcionalidades:

- Modelado de vehículos y peatones. Representación completa y realista de todo tipo de vehículos.
- Diseño de redes de tráfico y su topología. Definición de calles, carriles, intersecciones, semáforos y áreas peatonales.
- Simulación del tráfico. Simulación del movimiento de vehículos y personas a lo largo del tiempo. Permite analizar cómo las diferentes condiciones del tráfico, señales y regulaciones afectan al flujo vehicular y al tiempo de viaje.
- Control de semáforos y estrategias de tráfico. Evaluación e implementación de estrategias de control de semáforos y señales con la idea de mejorar la eficiencia y la seguridad.
- Evaluación ambiental. Medición de las emisiones de gases contaminantes y del consumo de combustible
- Integración con herramientas externas. Su conexión con entornos de programación como Python permite realizar análisis avanzados y personalizados.

## 2.2. NETEDIT

Herramienta utilizada en conjunto con Sumo enfocada a la creación y edición de las redes de tráfico. En esta aplicación se realiza el diseño de la topología de carreteras previo a la etapa de simulación del tráfico.

No solo permite la creación de la red urbana, sino también del resto de elementos necesarios, como son los semáforos, cruces de peatones, y definición de los flujos de vehículos y personas.



Imágenes 1 y 2. Logotipos de Sumo y Netedit

Cabe mencionar que antes de tomar la decisión de utilizar Sumo y Netedit se probaron otros simuladores gráficos, en especial el diseñado por Bilal Himate. Se intentó realizar un esquema de carreteras utilizando esta herramienta, pero tras varios diseños fallidos se llegó a la conclusión de que se trata de un entorno pensado para la simulación de una única intersección de carreteras, no de una red grande. Además el modelado de vehículos y personas y las

dificultades a la hora de detectar su presencia y de controlar los semáforos ponían en duda la viabilidad del proyecto.

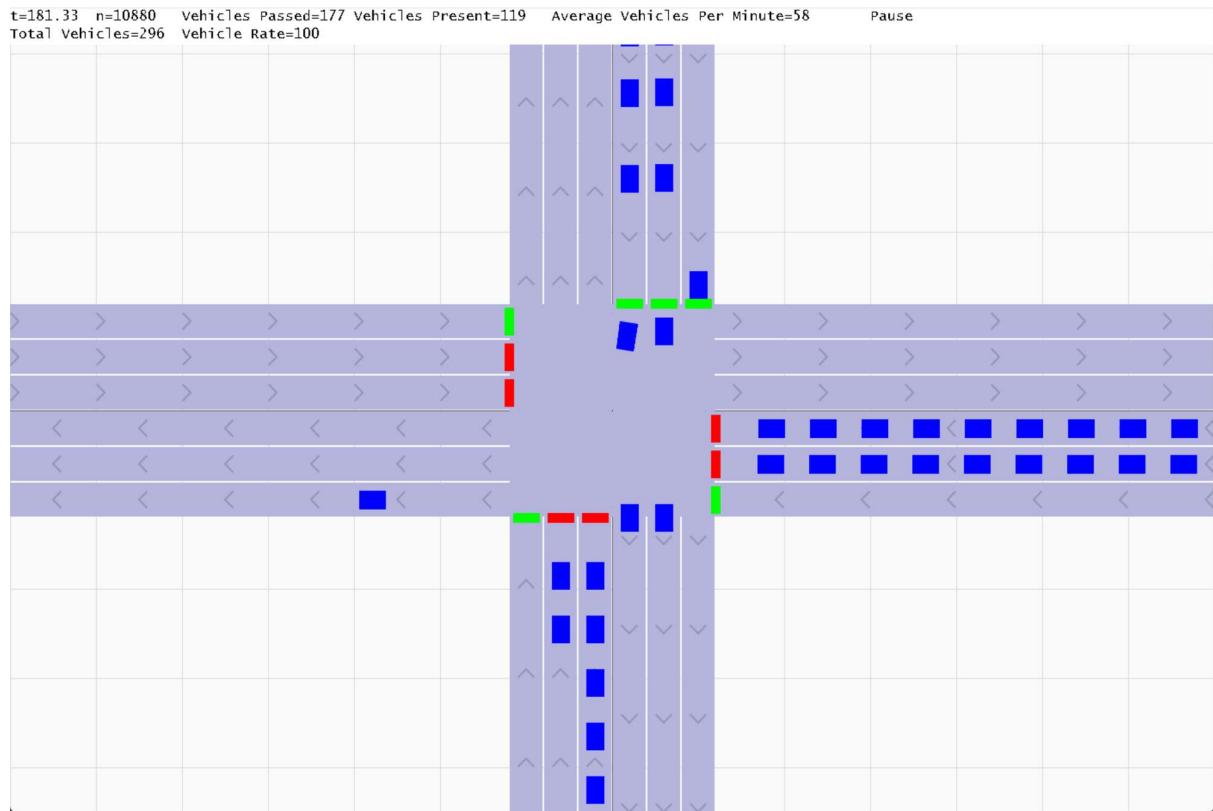


Imagen 3. Intersección creada con el simulador de Bilal Himite

### 2.3. LÓGICA BORROSA EN PYTHON

Se ha considerado que usar un controlador borroso es una de las mejores alternativas para este trabajo, principalmente debido a la complejidad del sistema. El tráfico urbano es tremadamente complejo y no lineal, con múltiples variables interrelacionadas, como el flujo de vehículos, patrones de comportamiento de los conductores, semáforos, condiciones meteorológicas y eventos especiales. Los controladores borrosos son capaces de manejar esta complejidad y adaptarse a las dinámicas cambiantes sin depender de modelos matemáticos precisos.

Por otra parte, estos controladores son bastante fáciles de implementar y ajustar. Existen interfaces como la toolbox de Matlab que permiten desarrollar un sistema de control borroso de forma rápida e intuitiva. Además es sencillo modificar los parámetros en caso de que el sistema no funcione como se espera.

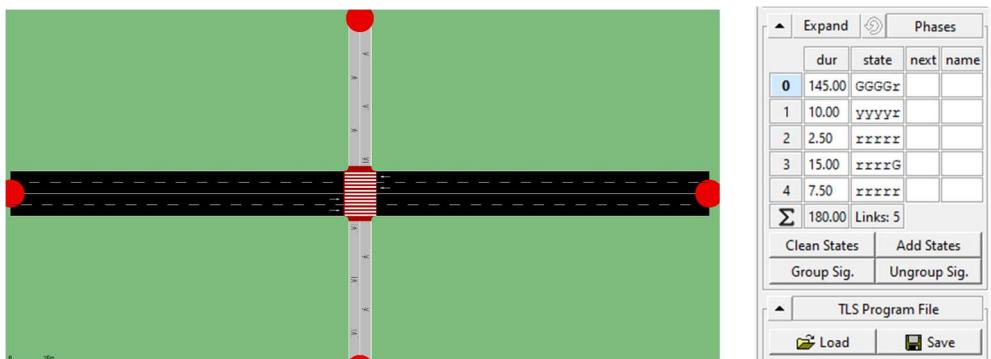
Otro factor es la escalabilidad, de forma que un sistema de control inteligente aplicado a una red de carreteras pequeña puede ampliarse fácilmente en un futuro a un modelo más grande.

### 3. MAPA 1: AVENIDA CON PASO DE CEBRA

#### 3.1. DISEÑO DEL MODELO Y CREACIÓN DEL MAPA

La explicación del diseño y creación del mapa se va a realizar de forma superficial, debido a que el principal objetivo del proyecto es el diseño de un controlador inteligente.

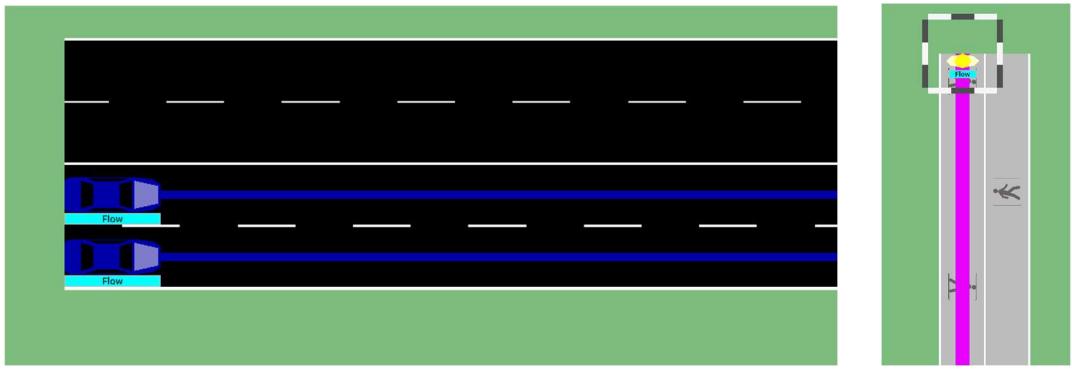
- El primer paso es crear el layout con las carreteras y calles utilizando la herramienta que se muestra en la imagen 10. Es posible habilitar y deshabilitar la creación de aceras para los peatones, así como la creación de carreteras de doble sentido. Como lo que se busca es recrear un paso de cebra que permita cruzar la avenida de lado a lado, será necesario añadir una vía peatonal que desemboque en un cruce, el cual se puede definir utilizando el icono mostrado en la imagen 11. El esquema final se muestra en la imagen 4.
- Una vez diseñado el cruce, es necesario añadir un semáforo que regule el paso de peatones y vehículos, utilizando el icono mostrado en la imagen 12. El semáforo es totalmente configurable, siendo posible definir cada una de sus fases manualmente tal y como se muestra en la imagen 5, o cargando algún archivo que se encargue de su control. En el siguiente apartado se realizará una explicación más detallada del código desarrollo para el control del semáforo para este cruce.



Imágenes 4 y 5. Layout de la avenida y panel de configuración de un semáforo

- El siguiente paso es definir los flujos de peatones y vehículos que se van a representar en la simulación. Para ello se utilizan los iconos mostrados en las imágenes 13 y 14. Los flujos definen la cantidad de individuos generados por unidad de tiempo, lo que permite representar situaciones de mayor o menor acumulación. En este caso se ha definido un flujo de vehículos para cada uno de los 4 carriles de la carretera, y también dos flujos de peatones, que parten desde la vía peatonal superior hasta llegar a la vía peatonal inferior y viceversa.

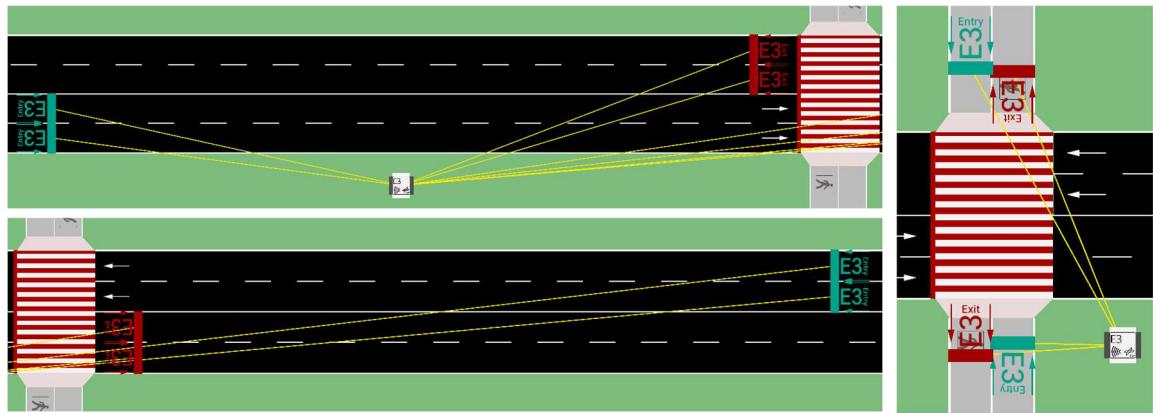
A lo largo de las diferentes etapas del trabajo se modificará la probabilidad de aparición de vehículos y personas para estudiar el comportamiento de los semáforos inteligentes.



Imágenes 6 y 7. Definición de flujos de vehículos y personas.

- Por último, queda añadir los detectores de vehículos/peatones. La posibilidad de configurar el límite de velocidad, el tiempo de detección o el tipo de vehículo, hace que sea posible programar estos elementos como detectores de atascos o simplemente como detectores de presencia. De este modo son tremadamente importantes para el proyecto ya que actúan como los sensores del sistema, proporcionando al controlador los valores de entrada necesarios para funcionar. Existen varios modelos seleccionables, pero en este caso se han utilizado los detectores tipo entrada/salida, que son capaces de determinar el número de agentes que han entrado o salido de una región concreta, como por ejemplo, el número de peatones esperando antes de un cruce. Por otra parte, estos detectores generan durante la simulación un archivo de tipo xml, en el que van registrando el número de usuarios contenidos en su área de acción, así como los tiempos de espera.

Para el mapa en cuestión, se han añadido dos sensores de este tipo, uno de ellos para los peatones y otro para los vehículos, tal y como se aprecia en las imágenes 8 y 9, con la idea de contabilizar el número de agentes que se acercan y que salen de la intersección en ambos sentidos.



Imágenes 8 y 9. Detector de acercamiento y salida de vehículos del cruce. Detector de entrada y salida de peatones del cruce.



Imágenes 10, 11, 12, 13 y 14. Iconos para la creación de elementos en el mapa.

## 3.2. DISEÑO E IMPLEMENTACIÓN DEL CONTROLADOR BORROSO

### 3.2.1. DEFINICIÓN DE ENTRADAS Y SALIDAS

El primer paso a la hora de crear cualquier tipo de controlador es definir las entradas y salidas. En este caso, el único sistema a controlar es el semáforo que regula el tráfico en la vía implementada, por lo que la salida es trivial: la orden de cambio de rojo a verde y viceversa. No es necesario definir una salida extra para el semáforo de los peatones, ya que el color de éste siempre es el contrario al que tiene el de los vehículos, por lo que si uno se pone en verde el otro debe cambiar a rojo y lo mismo al contrario.

Sin embargo, para definir las entradas es necesario analizar todas las situaciones posibles por las que dicho semáforo debería cambiar de color:

- Número de vehículos circulando por el tramo en cuestión.
- Número de peatones esperando para cruzar la calle controlada por el semáforo en cuestión.
- Tiempo de espera de los peatones.

Lo lógico sería añadir también como entrada el tiempo de espera de los vehículos, para evitar demoras prolongadas. Sin embargo, este hecho solo se da en simulaciones con un flujo constante exageradamente grande de peatones. Al ser una situación poco realista se ha decidido no incluir como entrada, ya que supone un aumento de la dificultad y la magnitud del conjunto de reglas a desarrollar.

### 3.2.2. DISEÑO DE LAS VARIABLES Y VALORES LINGÜÍSTICOS, CONJUNTOS Y REGLAS FUZZY

El desarrollo de controlador borroso se ha llevado a cabo en un entorno Python, gracias al uso de la librería “Scikit-fuzzy”, la cual incluye todas las herramientas y funciones necesarias para la implementación de este tipo de control.

En primer lugar se definen los conjuntos borrosos y las funciones de membresía. Los valores numéricos se han seleccionado de forma arbitraria.

```

peatones = ctrl.Antecedent(np.arange(0, 50, 0.1), 'peatones')
vehiculos = ctrl.Antecedent(np.arange(0, 30, 0.1), 'vehiculos')
tiempo_espera_peatones = ctrl.Antecedent(np.arange(0, 460, 1), 'tiempo_espera_peatones')
semaforo = ctrl.Consequent(np.arange(0, 1.1, 0.1), 'semaforo')

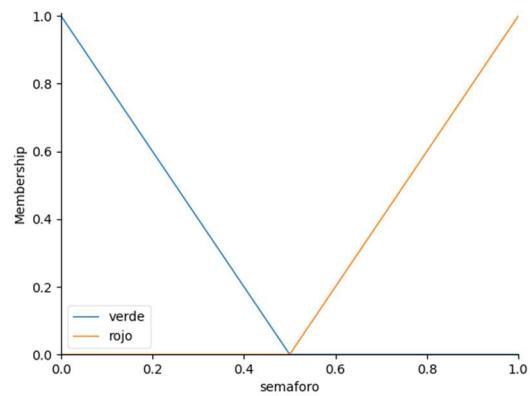
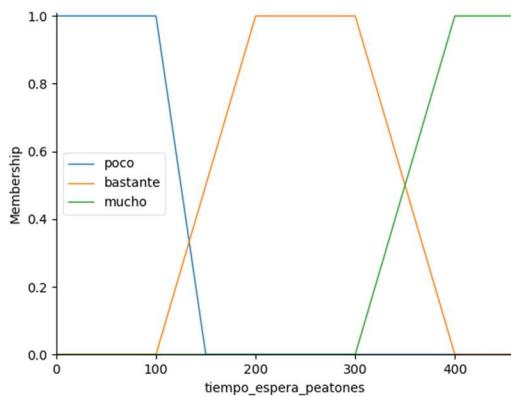
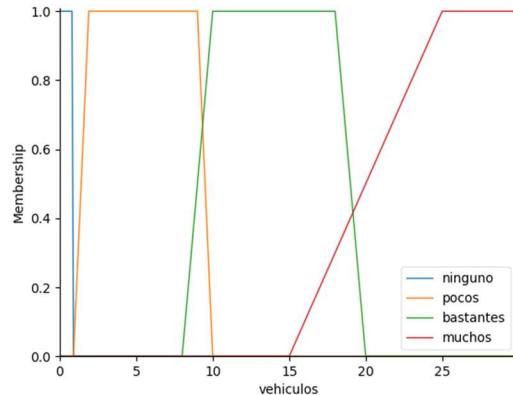
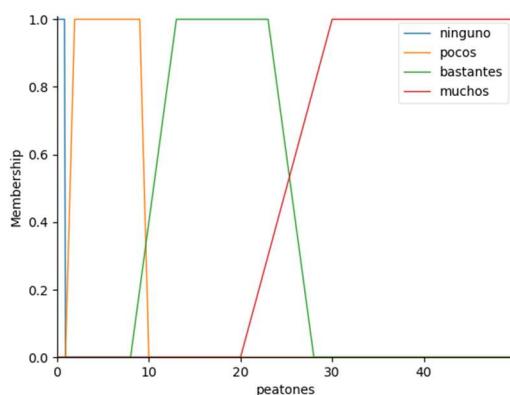
semaforo['verde'] = fuzz.trimf(semaforo.universe, [0, 0, 0.5])
semaforo['rojo'] = fuzz.trimf(semaforo.universe, [0.5, 1, 1])
semaforo.view()

peatones['ninguno'] = fuzz.trapmf(peatones.universe, [-1, 0, 0.8, 0.9])
peatones['pocos'] = fuzz.trapmf(peatones.universe, [0.9, 1.9, 9, 10])
peatones['bastantes'] = fuzz.trapmf(peatones.universe, [8, 13, 23, 28])
peatones['muchos'] = fuzz.trapmf(peatones.universe, [20, 30, 51, 55])
peatones.view()

vehiculos['ninguno'] = fuzz.trapmf(vehiculos.universe, [-1, 0, 0.8, 0.9])
vehiculos['pocos'] = fuzz.trapmf(vehiculos.universe, [0.9, 1.9, 9, 10])
vehiculos['bastantes'] = fuzz.trapmf(vehiculos.universe, [8, 10, 18, 20])
vehiculos['muchos'] = fuzz.trapmf(vehiculos.universe, [15, 25, 30, 35])
vehiculos.view()

tiempo_espera_peatones['poco'] = fuzz.trapmf(tiempo_espera_peatones.universe, [-1, 0, 100, 150])
tiempo_espera_peatones['bastante'] = fuzz.trapmf(tiempo_espera_peatones.universe, [100, 200, 300, 400])
tiempo_espera_peatones['mucho'] = fuzz.trapmf(tiempo_espera_peatones.universe, [300, 400, 500, 550])
tiempo_espera_peatones.view()

```



Imágenes 15, 16, 17, 18 y 19. Funciones de membresía.

El siguiente paso es implementar las reglas que establecen la relación entre las entradas y las salidas del sistema.

```

rule1a = ctrl.Rule(peatones['ninguno'] & ~vehiculos['ninguno'], semaforo['verde'])
rule1b = ctrl.Rule(vehiculos['ninguno'] & ~peatones['ninguno'], semaforo['rojo'])
rule1c = ctrl.Rule(vehiculos['ninguno'] & peatones['ninguno'], semaforo['verde'])

rule2a = ctrl.Rule(peatones['pocos'] & vehiculos['pocos'] & tiempo_espera_peatones['poco'] , semaforo['verde'])
rule2b = ctrl.Rule(peatones['pocos'] & vehiculos['pocos'] & ~tiempo_espera_peatones['poco'] , semaforo['rojo'])
rule2ca = ctrl.Rule(peatones['pocos'] & vehiculos['bastantes'] & ~tiempo_espera_peatones['mucho'] , semaforo['verde'])
rule2cb = ctrl.Rule(peatones['pocos'] & vehiculos['muchos'] & ~tiempo_espera_peatones['mucho'] , semaforo['verde'])
rule2da = ctrl.Rule(peatones['pocos'] & vehiculos['bastantes'] & tiempo_espera_peatones['mucho'] , semaforo['rojo'])
rule2db = ctrl.Rule(peatones['pocos'] & vehiculos['muchos'] & tiempo_espera_peatones['mucho'] , semaforo['rojo'])

rule3a = ctrl.Rule(peatones['bastantes'] & vehiculos['pocos'], semaforo['rojo'])
rule3ba = ctrl.Rule(peatones['bastantes'] & vehiculos['bastantes'] & ~tiempo_espera_peatones['mucho'], semaforo['verde'])
rule3bb = ctrl.Rule(peatones['bastantes'] & vehiculos['muchos'] & ~tiempo_espera_peatones['mucho'], semaforo['verde'])
rule3ca = ctrl.Rule(peatones['bastantes'] & vehiculos['bastantes'] & tiempo_espera_peatones['mucho'], semaforo['rojo'])
rule3cb = ctrl.Rule(peatones['bastantes'] & vehiculos['muchos'] & tiempo_espera_peatones['mucho'], semaforo['rojo'])

rule4a = ctrl.Rule(peatones['muchos'] & vehiculos['pocos'] , semaforo['rojo'])
rule4b = ctrl.Rule(peatones['muchos'] & vehiculos['bastantes'] & tiempo_espera_peatones['poco'] , semaforo['verde'])
rule4c = ctrl.Rule(peatones['muchos'] & vehiculos['bastantes'] & ~tiempo_espera_peatones['poco'] , semaforo['rojo'])
rule4d = ctrl.Rule(peatones['muchos'] & vehiculos['muchos'] & ~tiempo_espera_peatones['mucho'] , semaforo['verde'])
rule4e = ctrl.Rule(peatones['muchos'] & vehiculos['muchos'] & tiempo_espera_peatones['mucho'] , semaforo['rojo'])

```

Imagen 20. Reglas fuzzy para el paso de cebra simple.

Las reglas se han desarrollado de forma que se prioriza la fluidez del tráfico frente al avance de los peatones. Como se puede observar, en situaciones en las que hay una cantidad similar de vehículos y peatones, el semáforo se mantiene en verde para los vehículos hasta que el tiempo de espera de los peatones aumenta demasiado.

En cuanto al mecanismo de Inferencia y la defusificación, se utiliza el método de Mamdani, en el que se calcula el centroide del área bajo la curva de la función de membresía correspondiente, obteniéndose la salida final del controlador borroso. Somos conscientes de que el método de Sugeno habría encajado mejor en este sistema, puesto que es un mecanismo que para la salida usa barras o “singletons” que valen 1 en un punto y 0 en el resto, siendo la salida de las reglas directamente un valor “crisp” (encaja perfectamente el modelo del semáforo, puesto que la salida debe ser totalmente verde o totalmente rojo, y no un valor de pertenencia intermedio entre ambos colores). Sin embargo, el método de Mamdani es el más conocido y resultaba más sencillo de implementar, y además la defusificación mediante el centroide se encarga igualmente de proporcionar un valor discreto de salida.

```

control_semaforizacion = ctrl.ControlSystem([rule1a, rule1b, rule1c, rule2a, rule2b, rule2ca...])
semaforizacion = ctrl.ControlSystemSimulation(control_semaforizacion)

```

Imagen 21. Función para completar la defusificación

### 3.2.3. DESARROLLO DEL SCRIPT DE CONTROL

Una vez creado el controlador fuzzy es necesario desarrollar el código que permite gestionar los datos generados en la simulación para proporcionárselos al controlador, y del mismo modo enviar la salida obtenida al semáforo simulado.

El control del semáforo se ha realizado mediante la librería TraCI, que es una API entre la simulación y el script de python. En cada paso de la simulación se consulta qué vehículos y peatones están dentro de los sensores, se verifica su estado (esperando o velocidad de movimiento). Una vez identificadas las entidades presentes se evalúan sus tiempos de espera. Finalmente se introducen estos datos como inputs del controlador difuso y se obtiene una resultado para el semáforo.

```
def calcular_tráfico_en_cruce(cruce_id):
    """
    Calcula el tráfico esperando en un cruce en el simulador SUMO.
    Parameters:
    - cruce_id: Identificador del cruce.
    Returns:
    - trafico_esperando: Número de vehículos esperando en el cruce.
    """
    # Obtener la lista de vehículos en el cruce
    vehiculos_en_cruce = traci.junction.getIDList()

    # Contar el número de vehículos esperando en el cruce
    trafico_esperando = 0
    for vehiculo_id in vehiculos_en_cruce:
        if traci.vehicle.isStopped(vehiculo_id):
            trafico_esperando += 1

    return trafico_esperando
```

Imagen 22. Función para calcular tráfico en el cruce.

El output del controlador fuzzy oscila varía entre 0 y 1. EL rango de 0 a 0.5 es verde para los vehículos y de 0.51 a 1 verde para peatones. Cuanto más se aleja el output de 0.5, mayor es la cantidad de vehículos esperando.

```
if __name__ == "__main__":
    sumoBinary = "C:/Program Files (x86)/Eclipse/Sumo/bin/sumo-gui.exe"
    sumoConfig = "C:/Users/User/Master/1er Cuatri/Inteligencia Artificial/Trabajo IA/SumoNetedit/Avenida/Avenida.sumocfg"
    sumoCmd = [sumoBinary, '-c', sumoConfig, "--start"]
    print("here")
    traci.start(sumoCmd)

    semaforizacion=fuzzy_setup()
    # Definir la duración del ciclo de semáforo en segundos
    # Bucle principal
    tiempo_espera_peatones=0
    while traci.simulation.getMinExpectedNumber() > 0:
        # Obtener el tiempo actual de la simulación
        simulation_time = traci.simulation.getTime()

        #print(current_state)
        peatones = len(traci.multientryexit.getLastStepVehicleIDs('SensorPersonas'))
        stopped_pedestrians_ids = traci.multientryexit.getLastStepVehicleIDs('SensorPersonas')
        vehiculos = len(traci.multientryexit.getLastStepVehicleIDs('SensorCoches'))

        for pedestrian_id in stopped_pedestrians_ids:
            waiting_time = traci.person.getWaitingTime(pedestrian_id)
            if waiting_time > tiempo_espera_peatones:
                tiempo_espera_peatones=waiting_time #El tiempo de espera es el tiempo que lleva esperando el primer
                                                #peatón que llegó al cruce
```

```

# Aplicar el nuevo estado del semáforo
print("peatones: ", peatones)
print("vehiculos: ", vehiculos)
print("tiempo_espera_peatones: ", tiempo_espera_peatones)

output=get_fuzzy_output(semaforizacion, peatones, vehiculos, tiempo_espera_peatones)
state=traci.trafficlight.getRedYellowGreenState("J1")
print(output)
if output <= 0.5 and state == "rrrrG":
    traci.trafficlight.setRedYellowGreenState("J1", "rrrrr")
    for i in range(10):
        traci.simulationStep()
    traci.trafficlight.setRedYellowGreenState("J1", "GGGGr")
    for i in range(10):
        traci.simulationStep()
elif output > 0.5 and state == "GGGGr":
    traci.trafficlight.setRedYellowGreenState("J1", "yyyyr")
    for i in range(5):
        traci.simulationStep()
    traci.trafficlight.setRedYellowGreenState("J1", "rrrrG")
    tiempo_espera_peatones=0
    for i in range(10):
        traci.simulationStep()

# Avanzar la simulación
traci.simulationStep()

# Detener la simulación y cerrar la conexión TraCI
traci.close()

```

Imágenes 23 y 24. Función principal con la gestión de las reglas y las fases del semáforo.

Como se puede observar, la fase intermedia entre el paso de verde a rojo (el color ámbar) se define en esta parte del código, no dentro de las reglas fuzzy, puesto que el sistema solo se encarga de gestionar cuando corta o abre el tráfico.

### 3.3. RESULTADOS Y CONCLUSIONES. COMPARATIVA CON MÓDULO CLÁSICO CON SEMÁFOROS TEMPORIZADOS

Lo primero que se quiere recalcar es la forma de las funciones de membresía. Usar trapecios con pendientes grandes no es la mejor alternativa, puesto que se desnaturaliza el concepto de control borroso, cuya idea es representar los intervalos solapados unos con otros. Al hacerlo así se definen tramos poco borrosos, y el resultado final es que la salida salta de un estado a otro de forma muy brusca. Esto no tiene consecuencias importantes, ya que una salida numérica de 0.13 tiene exactamente el mismo resultado que una salida de 0.25 (semáforo para vehículos en verde y para peatones en rojo). No obstante, sí que afecta al tiempo en el que el semáforo se mantiene abierto. A más fuerte sea el grado de pertenencia de la salida, más alto será el tiempo de apertura.

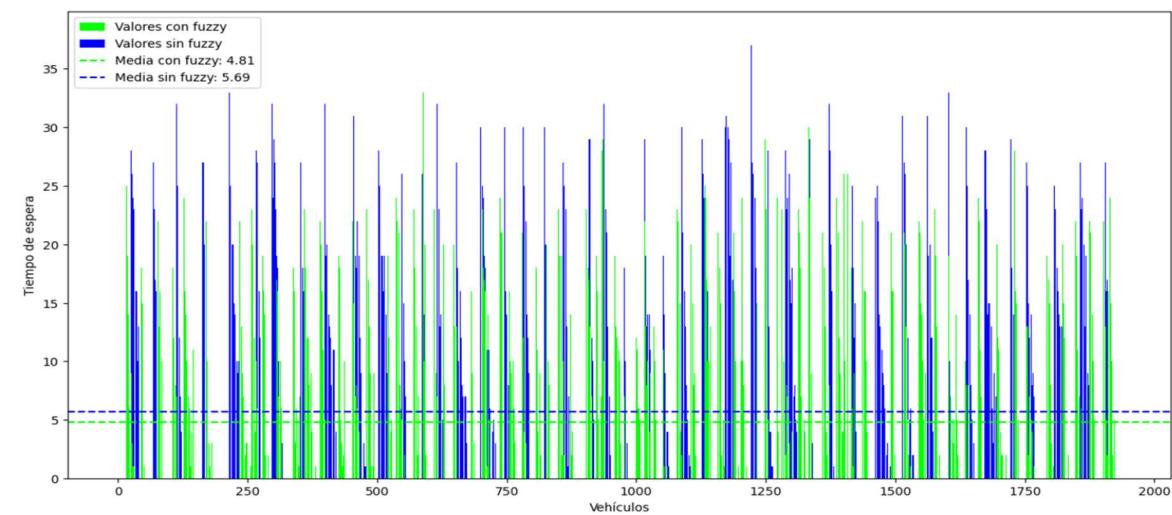
Aun así, somos conscientes que un diseño con funciones de membresía triangulares sería mucho más adecuado. Se diseñó con forma trapezoidal y funcionó realmente bien, luego quedaría como posible prueba comprobar la diferencia entre las dos filosofías.

El siguiente paso es hacer una comparación entre el modelo clásico y el modelo borroso para valorar los resultados de la utilización de este tipo de controlador.

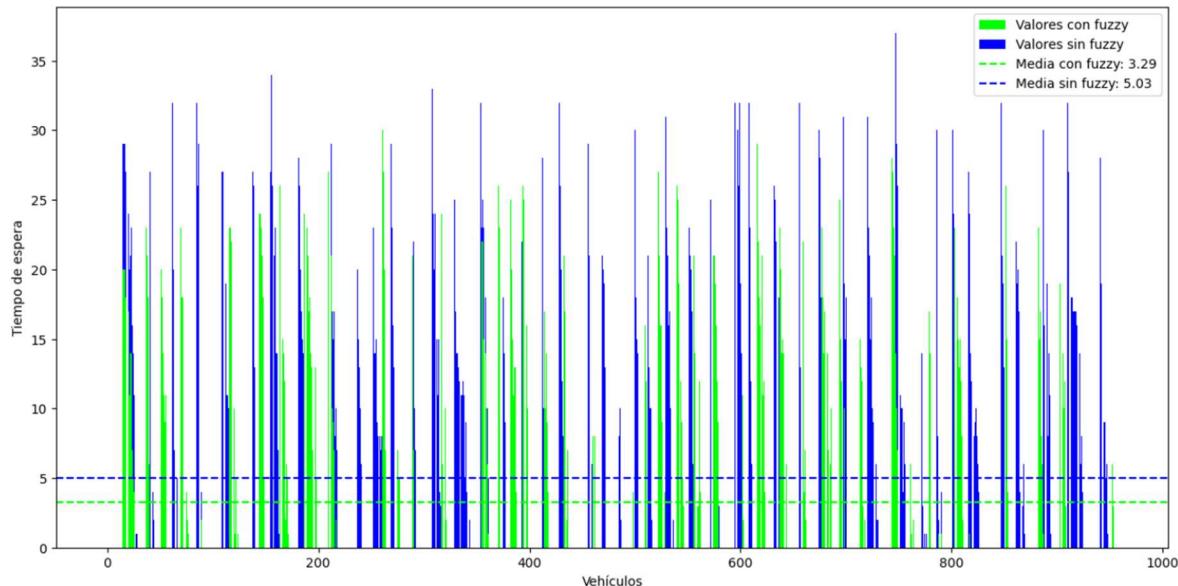
Para ello simplemente se ejecuta la simulación dos veces. En la primera se mantiene el código intacto, obteniéndose los valores de tiempo de espera individual y medio de cada vehículo y peatón generado en la simulación. En la segunda ejecución se deberá de eliminar del código del script aquellas funciones que se encargan de aplicar las reglas fuzzy y controlar las fases del semáforo, de forma que éstos pasen a funcionar de la forma tradicional, de forma temporizada sin tener en cuenta número de agentes o tiempos de espera.

#### - ANÁLISIS GLOBAL DEL TIEMPO DE ESPERA DE LOS VEHÍCULOS

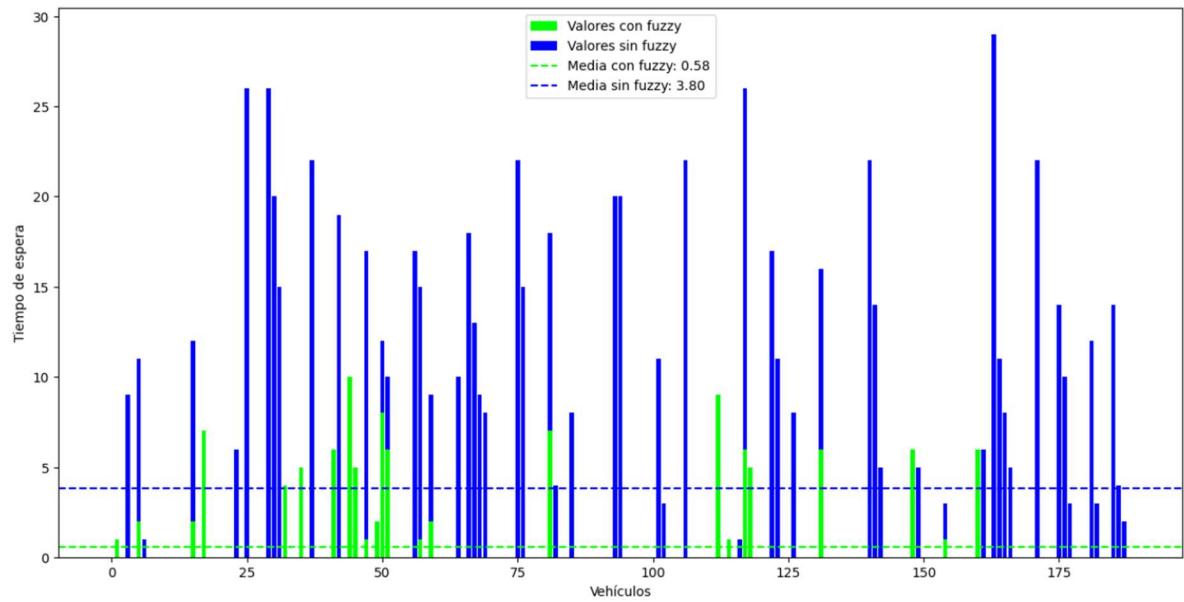
DENSIDAD ALTA



DENSIDAD MEDIA



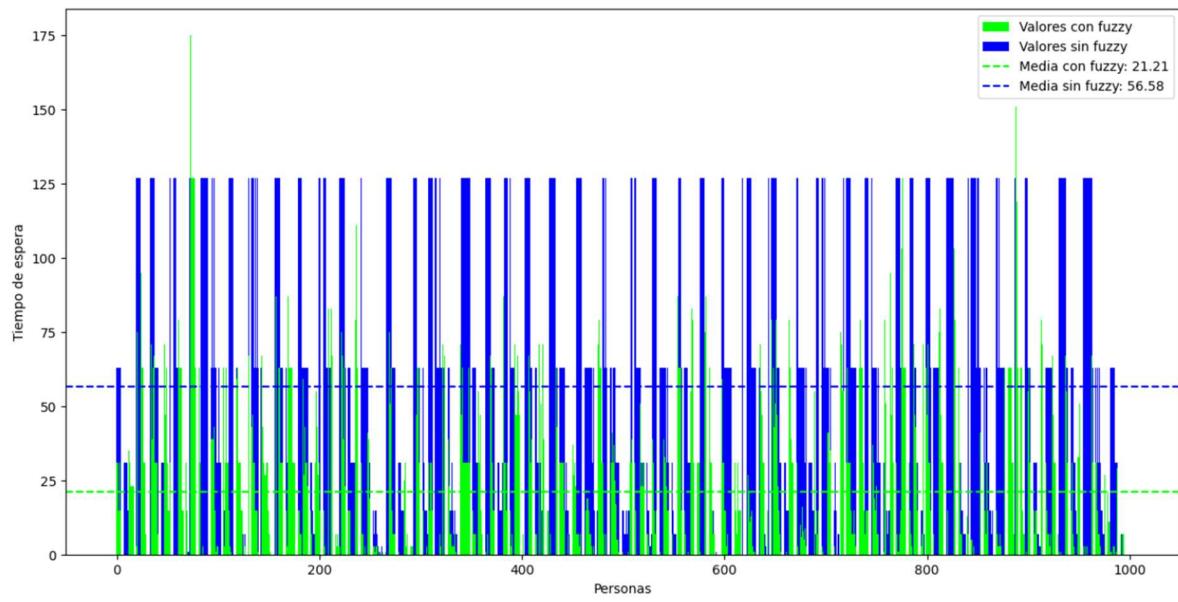
## DENSIDAD BAJA



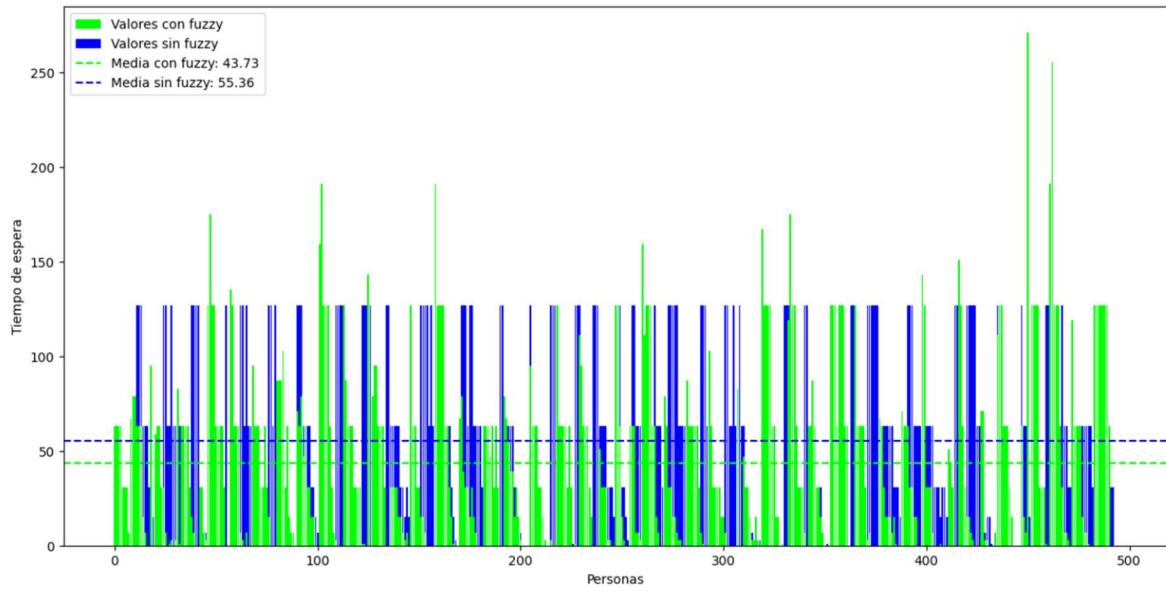
Según se reduce la densidad del tráfico se aprecia más la diferencia en cuanto a tiempo de espera entre los dos controladores.

- ANÁLISIS GLOBAL DEL TIEMPO DE ESPERA DE LOS PEATONES

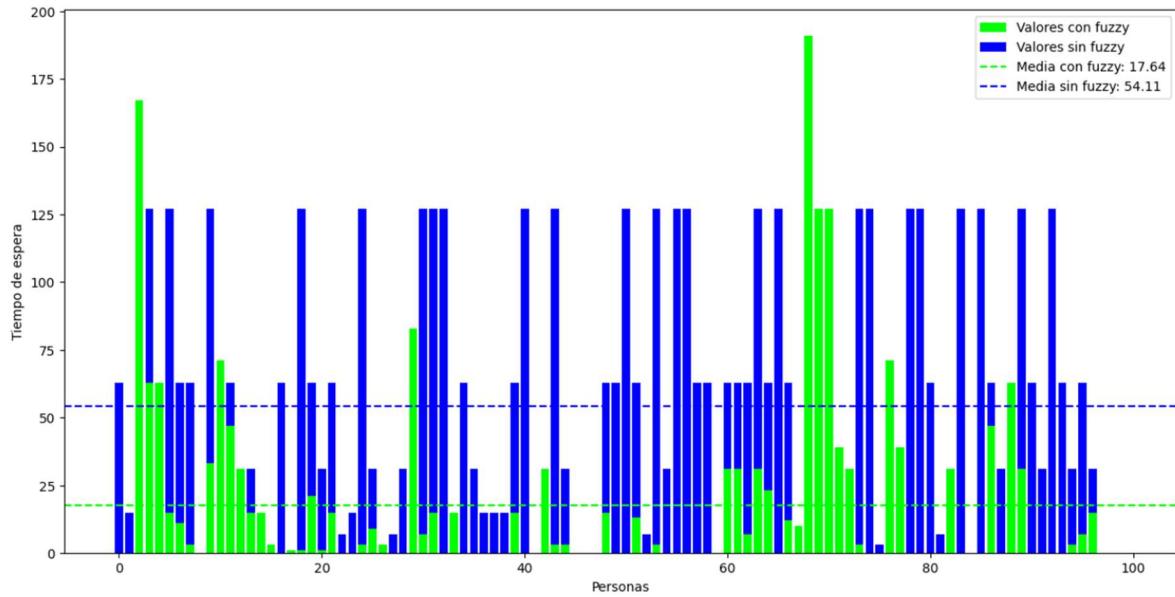
## DENSIDAD ALTA



## DENSIDAD MEDIA



## DENSIDAD BAJA



El resultado obtenido es llamativo, puesto que el controlador difuso favorece más el paso de vehículos frente al paso de peatones, pero a la vez tiene mucho sentido. Los tiempos del semáforo temporizado no favorecen mucho el cruce de los peatones(de cada 100 segundos de simulación solo 15 corresponden al paso de los mismos). Esta situación da lugar a que sea poco probable que puedan pasar justo cuando lleguen al cruce, y por tanto lo normal es que tengan que esperar en la mayoría de ocasiones. Sin embargo, en el modelo de control borroso pasa justo lo contrario, ya que al haber muy pocos vehículos circulando los peatones casi siempre cruzan en cuanto llegan al área de espera.

Se puede afirmar que los resultados obtenidos tras implementar semáforos inteligentes en este tipo de cruces son positivos, pero la idea es ir un paso más allá e intentar diseñar un sistema más complejo de semáforos interconectados para controlar una red de un tamaño mayor.

## VIDEO CON EL FUNCIONAMIENTO DEL PASO DE CEBRA

<https://www.youtube.com/watch?v=5Vb39hrMYpA>

## 4. MAPA 2: RED CON MÚLTIPLES INTERSECCIONES

### 4.1. DISEÑO DEL MODELO Y CREACIÓN DEL MAPA

Para evitar la redundancia se va a prescindir de esta explicación, al ser una repetición de los pasos comentados para el modelo 1.

Simplemente al ser una red más grande se han tenido que añadir más carreteras, intersecciones, semáforos, flujos y sensores, quedando el mapa final tal y como se muestra en la imagen 25:

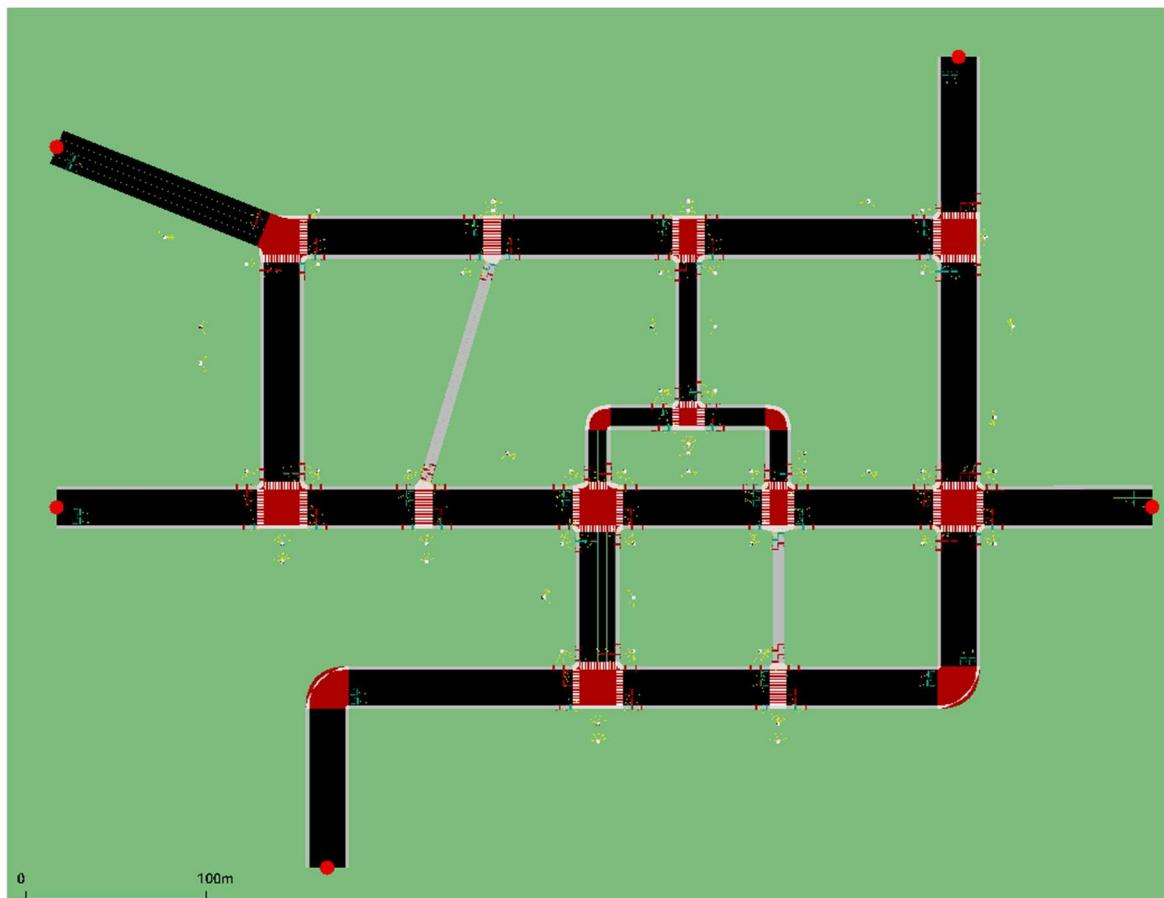


Imagen 25. Diseño final del mapa 2.

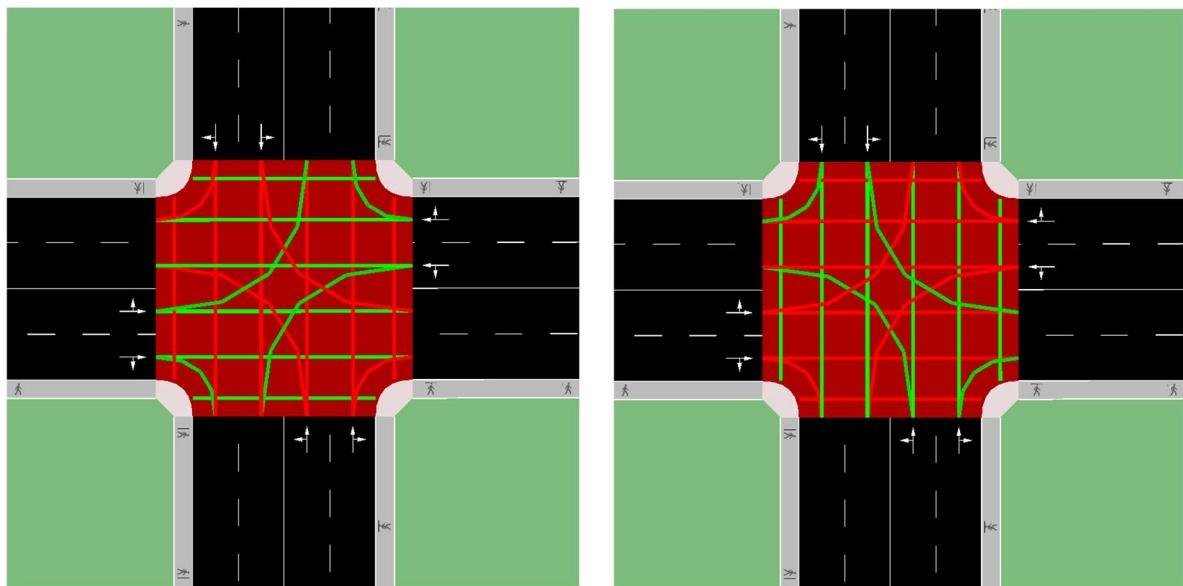
## 4.2. DISEÑO E IMPLEMENTACIÓN DEL CONTROLADOR BORROSO

### 4.2.1. DEFINICIÓN DE ENTRADAS Y SALIDAS

Al aumentar la complejidad de la red surgen intersecciones y cruces no contemplados en el modelo anterior. Esto da lugar a una serie de variaciones a la hora de implementar el controlador borroso, en función del tipo de semáforo.

Los **semáforos que regulan los pasos de cebra simples** como el mostrado en el modelo anterior no cambian, presentan las mismas entradas y salidas, por lo que las reglas utilizadas son las mismas.

En cuanto a los **semáforos que regulan el tráfico en intersecciones**, la salida del sistema de control será la orden de alternar entre las dos direcciones de circulación del flujo. En la primera fase, se permite el paso de los vehículos y peatones que circulan en dirección horizontal, mientras que en la segunda fase se permite el paso de los que circulan en dirección vertical.



Imágenes 26 y 27. Ambas fases de los semáforos situados en intersecciones.

Esta circunstancia obliga a tener en cuenta las condiciones de todos los agentes de la intersección para definir las entradas del nuevo controlador difuso:

- Número de vehículos circulando en la dirección horizontal de la intersección.
- Número de vehículos circulando en la dirección vertical de la intersección.
- Número de peatones esperando para cruzar en uno de los pasos horizontales de la intersección.
- Número de peatones esperando para cruzar en uno de los pasos verticales de la intersección.
- Tiempo de espera de cada uno de los cuatro agentes anteriores.

#### 4.2.2. DISEÑO DE LAS VARIABLES Y VALORES LINGÜÍSTICOS, CONJUNTOS Y REGLAS FUZZY

Del mismo modo que en el modelo anterior, se definen los conjuntos borrosos y las funciones de membresía, cuyos valores se han seleccionado de forma arbitraria.

```

peatones_vert = ctrl.Antecedent(np.arange(0, 50, 0.1), 'peatones_vert')
peatones_horiz = ctrl.Antecedent(np.arange(0, 50, 0.1), 'peatones_horiz')
vehiculos_vert = ctrl.Antecedent(np.arange(0, 30, 0.1), 'vehiculos_vert')
vehiculos_horiz = ctrl.Antecedent(np.arange(0, 30, 0.1), 'vehiculos_horiz')
tiempo_peatones_vert = ctrl.Antecedent(np.arange(0, 460, 1), 'tiempo_peatones_vert')
tiempo_peatones_horiz = ctrl.Antecedent(np.arange(0, 460, 1), 'tiempo_peatones_horiz')
tiempo_vehiculos_vert = ctrl.Antecedent(np.arange(0, 300, 1), 'tiempo_vehiculos_vert')
tiempo_vehiculos_horiz = ctrl.Antecedent(np.arange(0, 300, 1), 'tiempo_vehiculos_horiz')
semaforo = ctrl.Consequent(np.arange(0, 1.1, 0.1), 'semaforo')

semaforo['VerdeHoriz_RojoVert'] = fuzz.trimf(semaforo.universe, [0, 0, 0.5])
semaforo['RojoHoriz_VerdeVert'] = fuzz.trimf(semaforo.universe, [0.5, 1, 1])
semaforo.view()

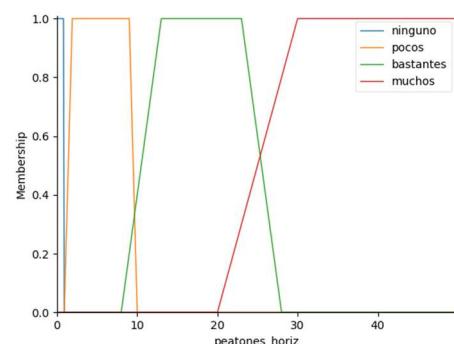
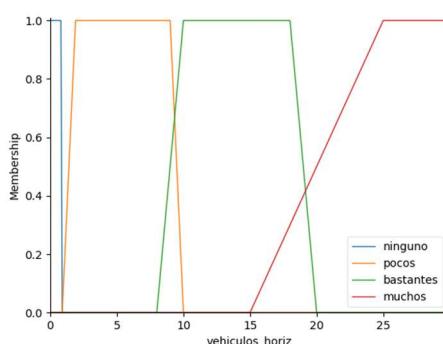
peatones_horiz['ninguno'] = fuzz.trapmf(peatones_horiz.universe, [-1, 0, 0.8, 0.9])
peatones_horiz['pocos'] = fuzz.trapmf(peatones_horiz.universe, [0.9, 1.9, 9, 10])
peatones_horiz['bastantes'] = fuzz.trapmf(peatones_horiz.universe, [8, 13, 23, 28])
peatones_horiz['muchos'] = fuzz.trapmf(peatones_horiz.universe, [20, 30, 51, 55])
peatones_horiz.view()

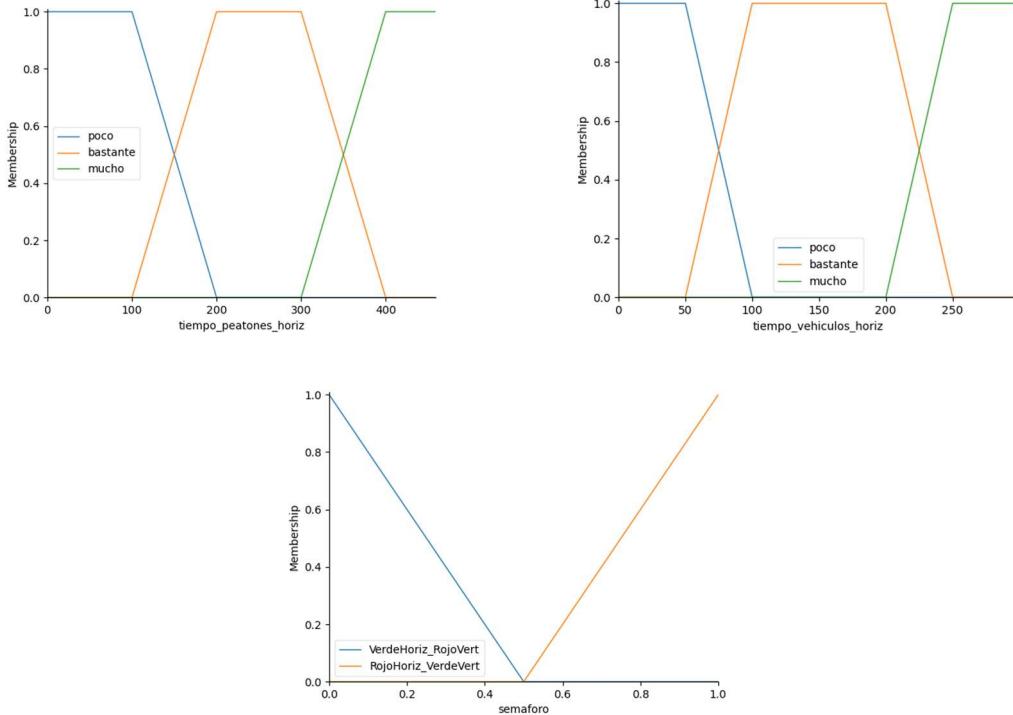
tiempo_peatones_horiz['poco'] = fuzz.trapmf(tiempo_peatones_horiz.universe, [-1, 0, 100, 200])
tiempo_peatones_horiz['bastante'] = fuzz.trapmf(tiempo_peatones_horiz.universe, [100, 200, 300, 400])
tiempo_peatones_horiz['mucho'] = fuzz.trapmf(tiempo_peatones_horiz.universe, [300, 400, 500, 550])
tiempo_peatones_horiz.view()

vehiculos_horiz['ninguno'] = fuzz.trapmf(vehiculos_horiz.universe, [-1, 0, 0.8, 0.9])
vehiculos_horiz['pocos'] = fuzz.trapmf(vehiculos_horiz.universe, [0.9, 1.9, 9, 10])
vehiculos_horiz['bastantes'] = fuzz.trapmf(vehiculos_horiz.universe, [8, 10, 18, 20])
vehiculos_horiz['muchos'] = fuzz.trapmf(vehiculos_horiz.universe, [15, 25, 30, 35])
vehiculos_horiz.view()

tiempo_vehiculos_horiz['poco'] = fuzz.trapmf(tiempo_vehiculos_horiz.universe, [-1, 0, 50, 100])
tiempo_vehiculos_horiz['bastante'] = fuzz.trapmf(tiempo_vehiculos_horiz.universe, [50, 100, 200, 250])
tiempo_vehiculos_horiz['mucho'] = fuzz.trapmf(tiempo_vehiculos_horiz.universe, [200, 250, 300, 350])
tiempo_vehiculos_horiz.view()

```





Imágenes 28 a 38. Funciones de membresía.

Se han omitido las definiciones y las gráficas de los peatones y vehículos circulando en la dirección vertical para evitar redundancia, ya que son idénticas a las mostradas anteriormente.

En cuanto a la definición de las reglas, hay un cambio importante en la forma en la que se ha enfocado su diseño. En el modelo del paso de cebra anterior, el número de peatones era un factor importante a la hora de modificar el estado del semáforo. Sin embargo, en este nuevo tipo de cruces siempre va a estar habilitada una de las dos direcciones de paso; es decir, si los pasos de cebra que regulan el flujo horizontal de peatones están en rojo, entonces los que regulan el flujo vertical estarán en verde. Lo mismo sucederá en el caso contrario. Esta situación, ligada a la alternancia entre los dos estados del semáforo provocada por el flujo relativamente uniforme de vehículos, evita que se acumule una gran cantidad de peatones en las zonas de espera. Es cierto que se pueden generar situaciones no óptimas, por ejemplo, que una masa de decenas de peatones tenga que esperar para cruzar debido a que un único vehículo impide su paso. Es un fallo asumible, ya que la prioridad del trabajo es gestionar mejor los atascos y el tráfico vehicular. Además, incluir el número de peatones como variable aumentaría bastante el número de reglas y su dificultad de forma innecesaria, puesto que son situaciones extrañas que en rara ocasión activarán dichas reglas.

A cambio, lo que si se tiene en cuenta es su tiempo de espera, de forma que, si por alguna circunstancia varios peatones llevan esperando demasiado tiempo sin poder cruzar, el semáforo les cederá el paso inmediatamente, independientemente del estado del tráfico vehicular. Esto se puede apreciar en el código de las reglas principales, de forma que todas tienen como primera condición que el tiempo de espera de los peatones no sea grande.

La conclusión a la que se llega es que, en este modelo, el tiempo de espera de los peatones es la entrada de mayor prioridad; si su valor es muy grande la mayoría de reglas se cancelan. Sin embargo, es muy complicado que se produzca esta situación, puesto que durante la mayor parte de la simulación el tiempo de espera de los peatones será bajo.

En caso de que no haya vehículos circulando, entonces sí se tendrá en cuenta el número de peatones en ambas direcciones para gestionar la prioridad de paso.

En el resto de circunstancias (que forman la mayoría de las reglas del controlador), el estado del semáforo dependerá del número de vehículos y de su tiempo de espera. En situaciones de ‘empate’, en las que estas dos entradas asociadas a los vehículos horizontales estén en igualdad de condiciones con las asociadas a los verticales, se cederá el paso al flujo horizontal. Este desempate es necesario hacerlo, y la decisión se ha tomado teniendo en cuenta que todas las vías horizontales del mapa creado son avenidas de dos carriles, mientras que algunas de las vías verticales son solo de un carril, lo que supone que una cantidad ligeramente superior de vehículos circulando horizontalmente que verticalmente.

```
#Situaciones sin vehiculos en ambas direcciones
rule1a = ctrl.Rule( V_H['ninguno'] & V_V['ninguno'] & P_H['ninguno'] & P_V['ninguno'] , S['VerdeH_RojoV'])
rule1b = ctrl.Rule( V_H['ninguno'] & V_V['ninguno'] & P_H['ninguno'] & ~P_V['ninguno'] , S['RojoH_VerdeV'])
rule1c = ctrl.Rule( V_H['ninguno'] & V_V['ninguno'] & ~P_H['ninguno'] & P_V['ninguno'] , S['VerdeH_RojoV'])
rule1d = ctrl.Rule( V_H['ninguno'] & V_V['ninguno'] & ~P_H['ninguno'] & ~P_V['ninguno'] & T_P_H['poco'] & T_P_V['poco'] , S['VerdeH_RojoV'])
rule1e = ctrl.Rule( V_H['ninguno'] & V_V['ninguno'] & ~P_H['ninguno'] & ~P_V['ninguno'] & T_P_H['poco'] & T_P_V['bastante'] , S['RojoH_VerdeV'])
rule1f = ctrl.Rule( V_H['ninguno'] & V_V['ninguno'] & ~P_H['ninguno'] & ~P_V['ninguno'] & T_P_H['bastante'] & T_P_V['poco'] , S['VerdeH_RojoV'])
rule1g = ctrl.Rule( V_H['ninguno'] & V_V['ninguno'] & ~P_H['ninguno'] & ~P_V['ninguno'] & T_P_H['bastante'] & T_P_V['bastante'] , S['VerdeH_RojoV'])

#Preferencia de paso inmediata los peatones que lleven mucho esperando, independientemente del estado del tráfico
rule1h = ctrl.Rule( ~P_H['ninguno'] & T_P_H['mucho'] , S['VerdeH_RojoV'])
rule1i = ctrl.Rule( ~P_V['ninguno'] & T_P_V['mucho'] , S['RojoH_VerdeV'])

#Situaciones con vehiculos en al menos una direccion:
rule2a = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['ninguno'] , S['RojoH_VerdeV'])

rule3a = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['ninguno'] , S['VerdeH_RojoV'])
rule3b = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['pocos'] & T_V_H['poco'] & T_V_V['poco'] , S['VerdeH_RojoV'])
rule3c = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['pocos'] & T_V_H['poco'] & ~T_V_V['poco'] , S['RojoH_VerdeV'])
rule3d = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['pocos'] & T_V_H['bastante'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule3e = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['pocos'] & T_V_H['bastante'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule3f = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['pocos'] & T_V_H['mucho'] , S['VerdeH_RojoV'])
rule3g = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['pocos'] & T_V_H['poco'] , S['RojoH_VerdeV'])
rule3h = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['bastantes'] & T_V_H['bastante'] & T_V_V['poco'] , S['VerdeH_RojoV'])
rule3i = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['bastantes'] & T_V_H['bastante'] & ~T_V_V['poco'] , S['RojoH_VerdeV'])
rule3j = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['bastantes'] & T_V_H['mucho'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule3k = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['bastantes'] & T_V_H['mucho'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule3l = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['muchos'] & T_V_H['poco'] , S['RojoH_VerdeV'])
rule3m = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['muchos'] & T_V_H['bastante'] , S['RojoH_VerdeV'])
rule3n = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['muchos'] & T_V_H['mucho'] & T_V_V['poco'] , S['VerdeH_RojoV'])
rule3o = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['pocos'] & V_V['muchos'] & T_V_H['mucho'] & ~T_V_V['poco'] , S['RojoH_VerdeV'])

rule4a = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['ninguno'] & T_V_H['poco'] & T_V_V['mucho'] , S['VerdeH_RojoV'])
rule4b = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['pocos'] & T_V_H['poco'] & T_V_V['mucho'] , S['VerdeH_RojoV'])
rule4c = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['pocos'] & T_V_H['poco'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule4d = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['pocos'] & T_V_H['bastante'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule4e = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['pocos'] & T_V_H['bastante'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule4f = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['pocos'] & T_V_H['mucho'] , S['VerdeH_RojoV'])
rule4g = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['bastantes'] & T_V_H['poco'] & T_V_V['poco'] , S['VerdeH_RojoV'])
rule4h = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['bastantes'] & T_V_H['poco'] & ~T_V_V['poco'] , S['RojoH_VerdeV'])
rule4i = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['bastantes'] & T_V_H['bastante'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule4j = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['bastantes'] & T_V_H['bastante'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule4k = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['bastantes'] & T_V_H['mucho'] , S['VerdeH_RojoV'])
rule4l = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['muchos'] & T_V_H['poco'] , S['RojoH_VerdeV'])
rule4m = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['muchos'] & T_V_H['bastante'] & T_V_V['poco'] , S['VerdeH_RojoV'])
rule4n = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['muchos'] & T_V_H['bastante'] & ~T_V_V['poco'] , S['RojoH_VerdeV'])
rule4o = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['muchos'] & T_V_H['mucho'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule4p = ctrl.Rule( ~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['bastantes'] & V_V['muchos'] & T_V_H['mucho'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
```

```

rule5a = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['ninguno'] , S['VerdeH_RojoV'])
rule5b = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['pocos'] & T_V_H['poco'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule5c = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['pocos'] & T_V_H['poco'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule5d = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['pocos'] & T_V_H['bastante'] , S['VerdeH_RojoV'])
rule5e = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['pocos'] & T_V_H['mucho'] , S['VerdeH_RojoV'])
rule5f = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['bastantes'] & T_V_H['poco'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule5g = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['bastantes'] & T_V_H['poco'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule5h = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['bastantes'] & T_V_H['bastante'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule5i = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['bastantes'] & T_V_H['bastante'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule5j = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['bastantes'] & T_V_H['mucho'] , S['VerdeH_RojoV'])
rule5k = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['muchos'] & T_V_H['poco'] & T_V_V['poco'] , S['VerdeH_RojoV'])
rule5l = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['muchos'] & T_V_H['poco'] & ~T_V_V['poco'] , S['RojoH_VerdeV'])
rule5m = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['muchos'] & T_V_H['bastante'] & ~T_V_V['mucho'] , S['VerdeH_RojoV'])
rule5n = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['muchos'] & T_V_H['bastante'] & T_V_V['mucho'] , S['RojoH_VerdeV'])
rule5o = ctrl.Rule(~T_P_H['mucho'] & ~T_P_V['mucho'] & V_H['muchos'] & V_V['muchos'] & T_V_H['mucho'] , S['VerdeH_RojoV'])

```

Imagenes 39, 40, 41, 42, 43. Reglas de control borroso en intersecciones.

El mecanismo de Inferencia y defusificación es el mismo que el del modelo anterior.

#### 4.2.3. DESARROLLO DEL SCRIPT DE CONTROL

Como este nuevo modelo incluye dos tipos de semáforos se han definido dos setup distintos, uno que incluye las reglas fuzzy para los semáforos que regulan pasos de cebra sencillos (reglas definidas en el apartado 3.3.2), y otro setup con las reglas fuzzy para el control del tráfico en intersecciones.

Del mismo modo, también se definen dos funciones que retornan las salidas de ambos tipos de semáforos

Al tratarse de un mapa más complejo y con multitud de intersecciones con características distintas se ha construido un “diccionario” donde se almacenan los sensores asociados a cada semáforo. De esta forma, se facilita el acceso a estos y la escalabilidad del programa.

Cada key del diccionario representa la ID de cada intersección, mientras que el contenido de las mismas son los id de los detectores correspondientes.

```

Diccionario = {
    "J55": ["Ped7_ArribaIzq", "Ped7_AbajoIzq", "Ped7_Arribadcha", "Ped7_Aabajodcha", "Veh7_Horiz", "Veh7_Vert"],
    "J58": ["Ped11_Aabajo", "Ped11_ArribaDcha", "Ped11_ArribaIzq", "Veh11_Vert", "Veh11_Horiz"],
    "J61": ["Ped12_Arriba", "Ped12_Abajo", "Veh12"],
    "J57": ["Ped10_ArribaIzq", "Ped10_AbajoDcha", "Ped10_AbajoIzq", "Ped10_ArribaDcha", "Veh10_Vert", "Veh10_Horiz"],
    "J15": ["Ped4_Dcha", "Ped4_AbajoIzq", "Ped4_ArribaIzq", "Veh4_Horiz", "Veh4_Vert"],
    "J14": ["Ped3_Arriba", "Ped3_AbajoIzq", "Ped3_AbajoDcha", "Veh3_Vert", "Veh3_Horiz"],
    "J13": ["Ped8_ArribaIzq", "Ped8_Abajo", "Ped8_ArribaDcha", "Veh8_Vert", "Veh8_Horiz"],
    "J4": ["Ped2_Arriba", "Ped2_Abajo", "Veh2"],
    "J19": ["Ped1_AbajoIzq", "Ped1_AbajoDcha", "Ped1_Arriba", "Veh1_Horiz", "Veh1_Vert"],
    "J54": ["Ped5_ArribaIzq", "Ped5_Abajo", "Ped5_ArribaDcha", "Veh5_Horiz", "Veh5_Vert"],
    "J3": ["Ped6_Arriba", "Ped6_Abajo", "Veh6"],
    "J56": ["Ped9_ArribaIzq", "Ped9_Abajo", "Ped9_ArribaDcha", "Veh9_Horiz", "Veh9_Vert"]
}

```

Imagen 44. Diccionario.

#### - *Process\_traffic\_data(IntersectionID)*

Esta función es la encargada de leer los sensores de cada semáforo e identificar los vehículos y personas que están cerca, así como su tiempo de espera en dicho semáforo. En cuanto al cómputo del tiempo de espera se identifican aquellas entidades cuya velocidad es nula.

Para saber si las entidades pretenden cruzar el cruce de forma horizontal u vertical, se obtiene la dirección de las personas y vehículos detectados por los sensores. Una vez calculada la cantidad de entidades y sus tiempos de espera dentro de un intersección, se usan dichos como inputs para el controlador difuso.

- ***Manage\_traffic\_lights(IntersectionID, margen)***

Esta función es la función principal ejecutada en cada bucle, una vez por intersección existente. Se encarga de llamar a “process\_traffic\_data” para evaluar el estado del tráfico dentro de cada cruce y obtener el output del controlador. Con este output se decide si abrir el semáforo en sentido vertical u horizontal. Si el valor obtenido es  $\leq 0.5$ , circula el flujo horizontal, mientras que para valores  $< 0.5$ , se da prioridad al flujo vertical.

A diferencia del cruce simple, en este caso se ha implementado un tiempo de apertura relacionado con la salida del controlador difuso. Si el controlador detecta que hay muchos coches esperando para cruzar, el tiempo de apertura será mayor a que si fueran pocos. Este tiempo de apertura se traduce dentro del código como la variable margen. De tal forma que no se puede modificar el semáforo hasta que haya transcurrido dicho margen de espera.

- ***if \_\_name\_\_ == "\_\_main\_\_"***

En esta sección es donde se realiza el bucle de la simulación y donde se llama a tantas “manage\_traffic\_lights” como intersecciones haya.

```
def process_traffic_data(IntersectionID):
    detectores_id = Diccionario[IntersectionID]
    all_ids = []
    vehicle_ids = traci.vehicle.getIDList()
    for detector_id in detectores_id:
        all_ids.extend(traci.multientryexit.getLastStepVehicleIDs(detector_id))

    peatones_ids = [id for id in all_ids if id not in vehicle_ids]
    vehiculos_ids = [id for id in all_ids if id in vehicle_ids]
    tiempo_espera_peatones_vertical = 0
    tiempo_espera_peatones_horizontal = 0
    tiempo_espera_coches_vertical = 0
    tiempo_espera_coches_horizontal = 0
    num_peatones_vertical = 0
    num_peatones_horizontal = 0
    num_coches_vertical = 0
    num_coches_horizontal = 0
    for peaton_id in peatones_ids:
        if(traci.person.getSpeed(peaton_id) <= 0.1):
            angle = traci.person.getAngle(peaton_id)
            if 355 < angle < 360:
                angle = 0
            if angle<=5 or 175<=angle<=185:
                waiting_time = traci.person.getWaitingTime(peaton_id)
                tiempo_espera_peatones_vertical += waiting_time
                num_peatones_vertical += 1
            elif 85<=angle<=95 or 265<=angle<=275:
                waiting_time = traci.person.getWaitingTime(peaton_id)
                tiempo_espera_peatones_horizontal += waiting_time
                num_peatones_horizontal += 1

    for coche_id in vehiculos_ids:
        angle = traci.vehicle.getAngle(coche_id)
        if 355 < angle < 360:
            angle = 0
        print('angle car:',angle)
        if angle<=5 or 175<=angle<=185:
            waiting_time = traci.vehicle.getWaitingTime(coche_id)
            tiempo_espera_coches_vertical += waiting_time
            num_coches_vertical += 1
        elif 85<=angle<=95 or 265<=angle<=275:
            waiting_time = traci.vehicle.getWaitingTime(coche_id)
            tiempo_espera_coches_horizontal += waiting_time
            num_coches_horizontal += 1
```

```
if __name__ == "__main__":
    sumoBinary = "C:/Program Files (x86)/Eclipse/Sumo"
    sumoConfig = "C:/Users/Usuario/Documents/Master/I"
    sumoCmd = [sumoBinary, '-c', sumoConfig, "--start"]
    print("here")
    traci.start(sumoCmd)
    tiempo_espera_peatones=0
    vehicle_ids = traci.vehicle.getIDList()
    T_J55 = 0
    T_J58 = 0
    T_J61 = 0
    T_J57 = 0
    T_J15 = 0
    T_J14 = 0
    T_J13 = 0
    T_J4 = 0
    T_J19 = 0
    T_J54 = 0
    T_J3 = 0
    T_J56 = 0
    while traci.simulation.getTime() <= 3600:
        T_J55 = manage_traffic_lights('J55',T_J55)
        T_J58 = manage_traffic_lights('J58',T_J58)
        T_J61 = manage_traffic_lights('J61',T_J61)
        T_J57 = manage_traffic_lights('J57',T_J57)
        T_J15 = manage_traffic_lights('J15',T_J15)
        T_J14 = manage_traffic_lights('J14',T_J14)
        T_J13 = manage_traffic_lights('J13',T_J13)
        T_J4 = manage_traffic_lights('J4',T_J4)
        T_J19 = manage_traffic_lights('J19',T_J19)
        T_J54 = manage_traffic_lights('J54',T_J54)
        T_J3 = manage_traffic_lights('J3',T_J3)
        T_J56 = manage_traffic_lights('J56',T_J56)
    traci.simulationStep()
```

```

def manage_traffic_lights(IntersectionID, margen):
    time=traci.simulation.getTime()
    if time >= margen:
        output=process_traffic_data(IntersectionID)
        #print(output)
        pre_state = traci.trafficlight.getPhase(IntersectionID)
        if output <= 0.5:
            state = traci.trafficlight.setPhase(IntersectionID, 0)
            if state != pre_state:
                gain=int(abs(output)*20)
                margen=gain+time
        elif output > 0.5:
            state = traci.trafficlight.setPhase(IntersectionID, 3)
            if state != pre_state:
                gain=int(abs(output-0.5)*20)
                margen=gain+time
    return margen

```

Imágenes 45, 46 y 47.: Funciones del script de control.

### 4.3. RESULTADOS Y CONCLUSIONES. COMPARATIVA CON MÓDELO CLÁSICO CON SEMÁFOROS TEMPORIZADOS

El primer detalle importante es que el tiempo de apertura de estos semáforos es proporcional a lo agresiva que es la salida del controlador difuso. Es decir, si hay muchas entidades esperando, la duración del semáforo a su favor será mayor que si fueran pocas.

Igual que en el mapa anterior, para poder hacer una comparación entre el modelo clásico y el modelo borroso es necesario ejecutar la simulación dos veces con la misma probabilidad de generación de peatones y vehículos, una para cada tipo de control.

La comparación se llevará a cabo en tres situaciones distintas, modificando las probabilidades de aparición de individuos:

- Densidad alta: gran cantidad de vehículos, que provocan retenciones en los cruces y que se asemejan a los atascos producidos en las grandes ciudades en horas punta.
- Densidad media: flujo normal de vehículos y peatones, sin grandes acumulaciones, pero con buena frecuencia de entrada y salida de vehículos en las intersecciones.
- Densidad baja: pocos vehículos y peatones presentes, lo que se asemeja a una situación de tráfico nocturno.

Además, se va a analizar cómo afecta el control borroso a los tiempos de espera de vehículos y peatones de forma individual, así como un estudio del tiempo en cada semáforo por separado.

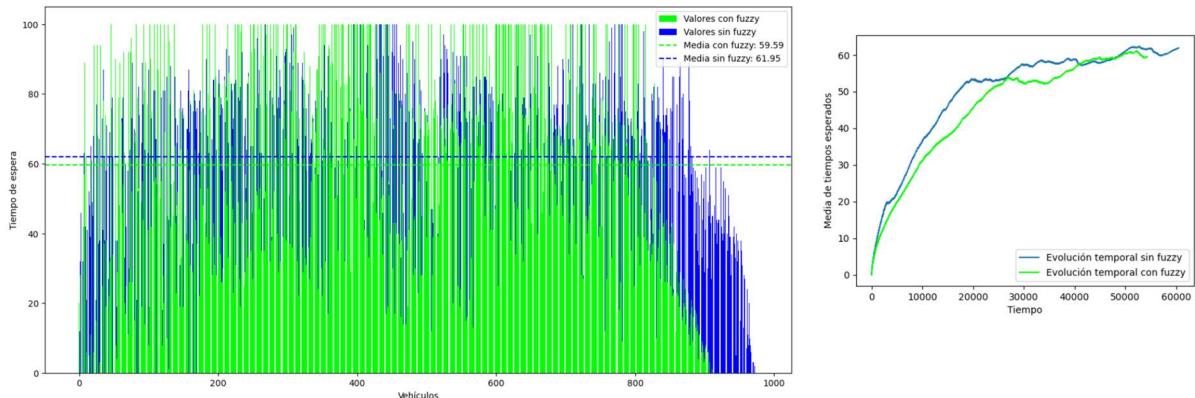
#### - ANÁLISIS GLOBAL DEL TIEMPO DE ESPERA DE LOS VEHÍCULOS

Como aclaración para las tres gráficas que se muestran a continuación, el eje de abscisas representa el ID de cada vehículo que se ha generado en la simulación. Es decir, el valor 400 no indica el tiempo global de espera con 400 vehículos en la red, sino el tiempo de espera individual del vehículo con id 400. Esto dependerá de la situación específica en la que se haya visto involucrado dicho vehículo durante la simulación. Como la generación de aparición se realiza en base a una probabilidad, el tiempo de espera de cada vehículo concreto es aleatorio, cambiando en cada test que se realice. Sin embargo, tras realizar varias pruebas se puede

confirmar que el tiempo promedio de espera se mantiene siempre en torno al mismo valor real, por lo que se tomará como dato relevante para el siguiente estudio.

Otro detalle importante a tener en cuenta es que las gráficas con la evolución temporal (derecha) se han escalado automáticamente según el tiempo que tarda el controlador en terminar de gestionar todo los vehículos que se generan.

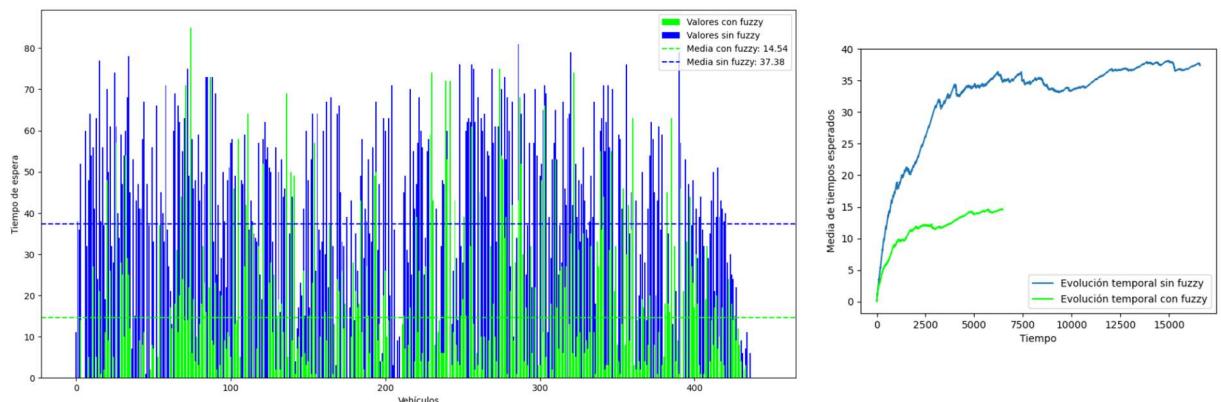
### DENSIDAD ALTA



Como se puede observar, en situaciones con una elevada cantidad de vehículos los tiempos de espera de ambos modelos son muy similares. Este comportamiento era predecible, pues es evidente que en situaciones con mucho tráfico es casi imposible gestionar el paso sin evitar que nadie tenga que esperar. A pesar de ello, se aprecia una pequeña mejora de un par de segundos de media.

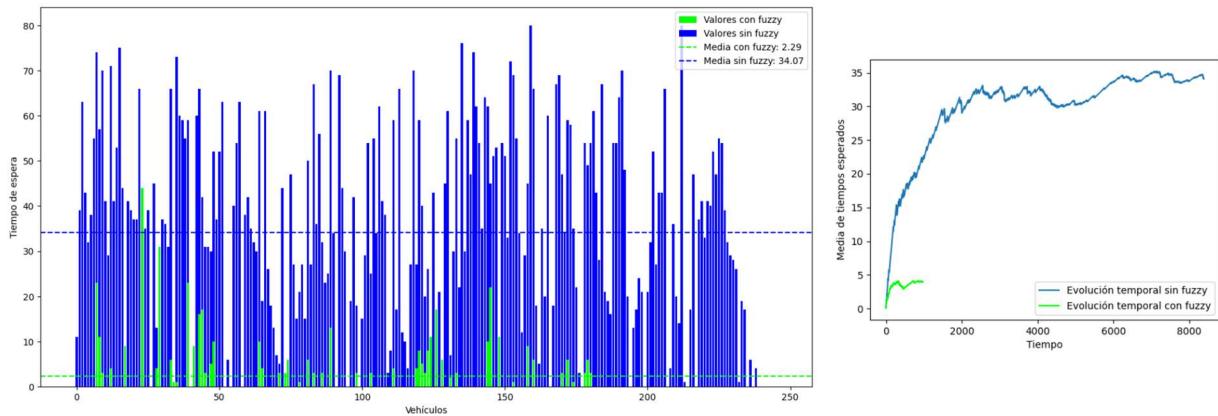
Es destacable el comportamiento de ambos modelos al llegar al final de la simulación, que es la fase en la que los vehículos dejan de generarse y los atascos empiezan a reducirse. Se puede apreciar como la pendiente de bajada es mayor al usar un controlador borroso que al usar un controlador tradicional. La conclusión a la que se llega es que el proceso de descongestión de una red de este tipo es más efectivo utilizando un controlador borroso.

### DENSIDAD MEDIA



Para este segundo caso, con un nivel de tráfico normal, la diferencia es muy destacable, reduciendo a la mitad los tiempos de espera medios. Según se reduce la cantidad de vehículos se observa como el controlador inteligente es capaz de gestionar mejor los tiempos de espera y los flujos entrantes.

## DENSIDAD BAJA



Simplemente con los resultados obtenidos en la situación anterior el uso de un controlador fuzzy ya quedaba totalmente justificado, pero en este tercer caso la diferencia es abrumadora. Con tan poco tráfico el controlador difuso es capaz de ceder el paso de forma inmediata a los vehículos que entran en las intersecciones, y en caso de tener que esperar el tiempo será ínfimo.

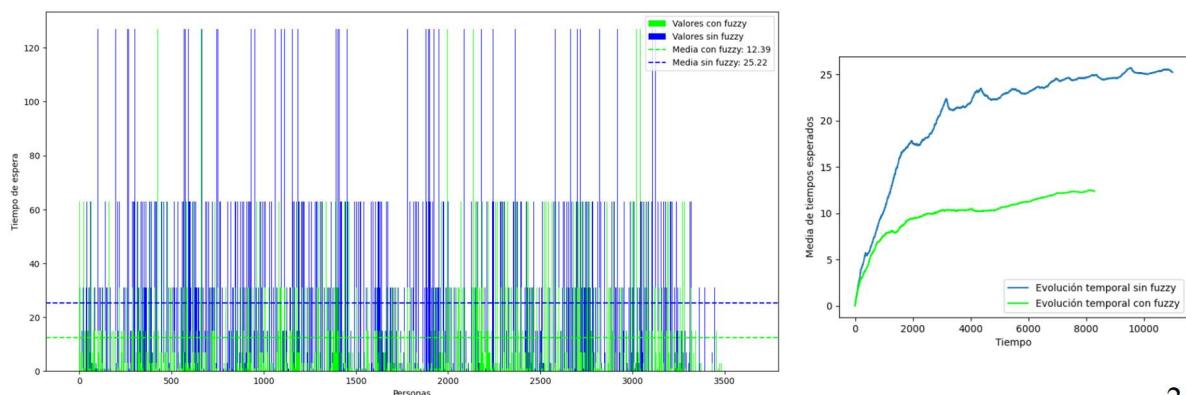
Sin embargo, los semáforos tradicionales no tienen esa inteligencia para controlar el paso, de forma que tener el hecho de tener que esperar o no al llegar a un cruce recae en el azar y la aleatoriedad. De hecho, se puede apreciar como el tiempo de espera promedio utilizando semáforos temporizados es prácticamente el mismo con una densidad de tráfico media y baja (37.38 frente a 34.07).

La conclusión que se obtiene es evidente: el control borroso evita las esperas innecesarias que se dan al utilizar semáforos temporizados, como por ejemplo que un vehículo tenga que esperar cuando no hay ningún vehículo ni peatón cruzando en la intersección.

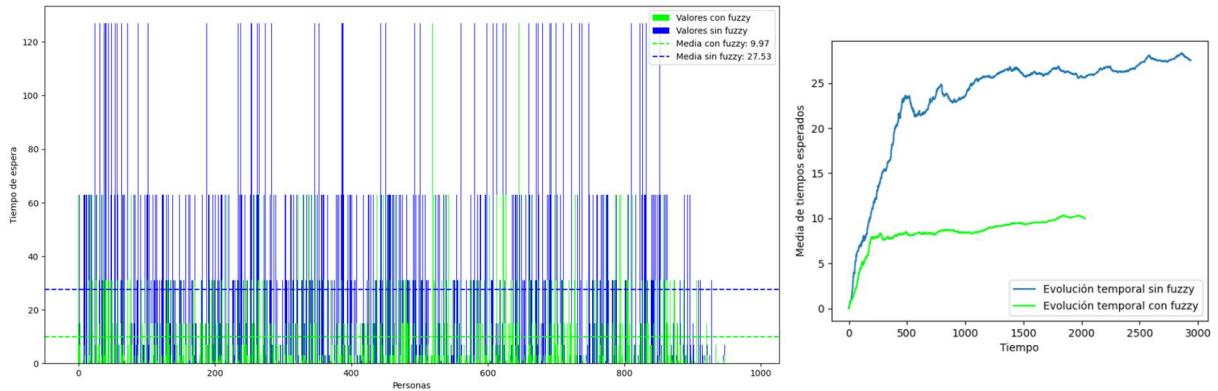
Otra conclusión importante a la que se llega al comparar esta gráfica con la obtenida en el mapa con un solo paso de cebra es que, cuanto más grande sea la red mejores beneficios se observan en el tiempo de espera global. En un solo cruce, puede que el controlador difuso solo mejore un par de segundos el tiempo de espera medio, pero esa diferencia multiplicada por todas las intersecciones de la red genera una reducción de los tiempos muy importante.

### - ANÁLISIS GLOBAL DEL TIEMPO DE ESPERA DE LOS PEATONES

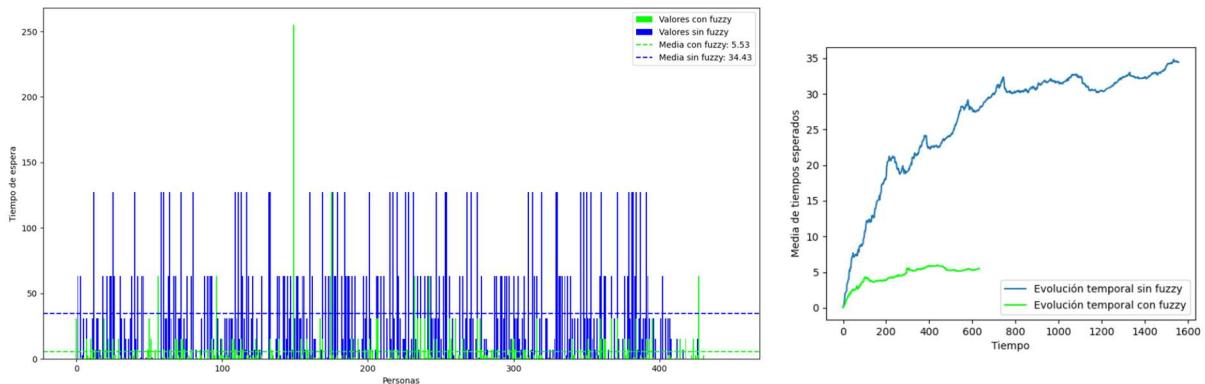
## DENSIDAD ALTA



## DENSIDAD MEDIA



## DENSIDAD BAJA



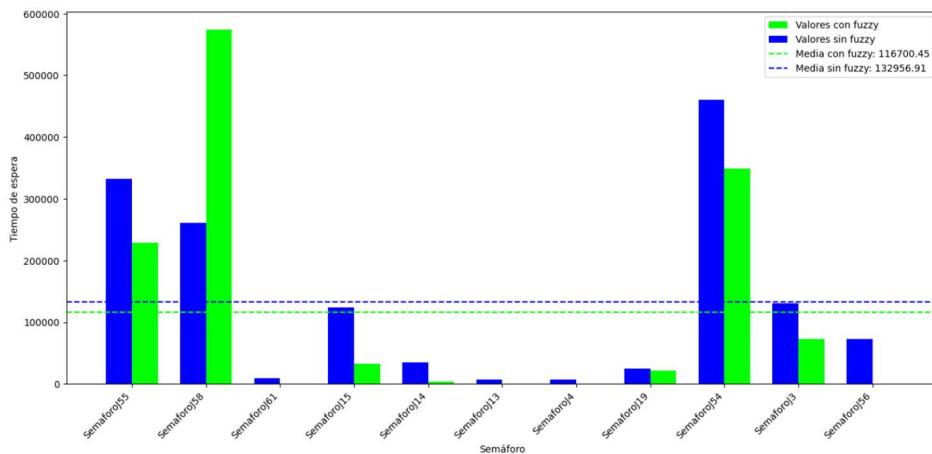
Sacar conclusiones de los tiempos de espera promedio de los peatones es más complicado, puesto que, como se comentaba en apartados anteriores, en las intersecciones siempre está en abierto el paso de peatones en una de las direcciones, lo que evita que se generen tiempos de espera muy elevados.

Es probable que la diferencia que se observa en el modelo de alta densidad se deba al control inteligente en los pasos de cebra simples, en cuyas reglas se da más prioridad al paso de los peatones que en las reglas de las intersecciones.

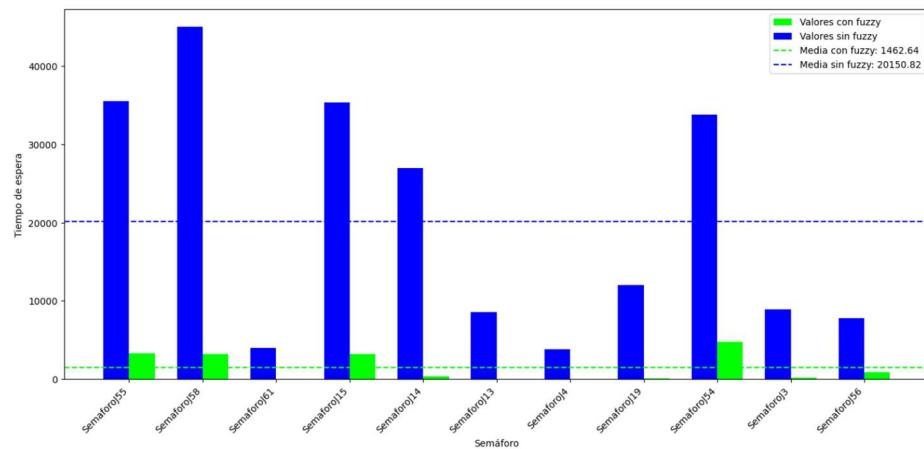
Del mismo modo que ocurría con los vehículos, según disminuye la cantidad de agentes en la red menor es el tiempo de espera promedio, al estar los cruces despejados la mayor parte del tiempo.

- ANÁLISIS INDIVIDUAL EN CADA SEMÁFORO DEL TIEMPO DE ESPERA TOTAL

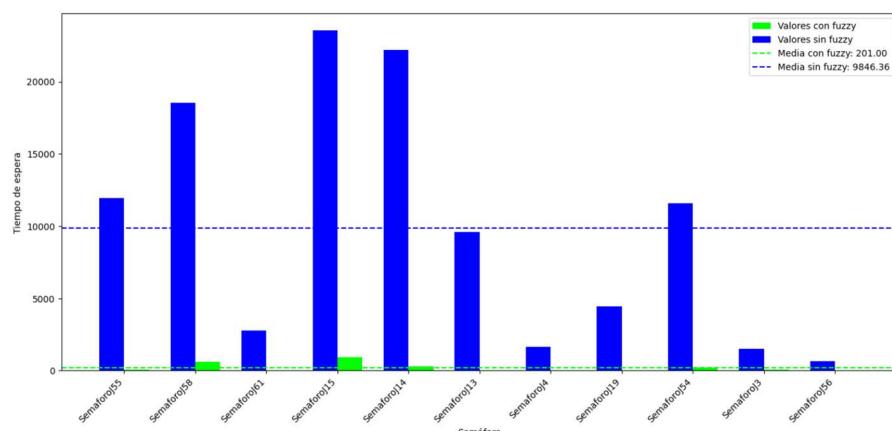
DENSIDAD ALTA



DENSIDAD MEDIA



DENSIDAD BAJA



El desempeño del controlador se ilustra bastante bien estas gráficas. Cada columna representa la suma total de los tiempos de espera de las entidades durante la simulación. Como se puede observar, el controlador difuso gestiona mucho mejor el flujo, reduciendo muy notoriamente los tiempos de esperas. Llama la atención como en el caso de densidad alta, la intersección 58

está más saturada que cualquiera del resto. Sus intersecciones vecinas son las 55, 61 y punto de entrada de vehículos. Este resultado se debe a que es un punto donde convergen muchas rutas y la correcta gestión del resto de cruces derivada en un cuello de botella en esta intersección concreta.

## VIDEO CON EL FUNCIONAMIENTO DE LOS SEMÁFOROS EN EL MAPA 2

[https://www.youtube.com/watch?v=x\\_m-OXR5des](https://www.youtube.com/watch?v=x_m-OXR5des)

<https://www.youtube.com/watch?v=bYRgrewkHxE>

## 5. COMPARATIVA DEL CONTROLADOR FUZZY CON ALGORITMO GENÉTICO

Debido al buen rendimiento del controlador difuso, se decidió compararlo con otras técnicas de optimización, como es el caso de los algoritmos genéticos. Dado el tiempo limitado del que se disponía se optó por usar un diseño de algoritmo genético ya hecho para el framework Sumo. En la bibliografía se puede observar el repositorio de donde se extrajo el código.

### 5.1. DISEÑO DEL ALGORITMO GENÉTICO

**Diseño del cromosoma:** Se trata de un array que contiene las duraciones y estados (rojo, verde, ámbar) asociados a cada duración de todos los semáforos del mapa.

**Función Fitness:** Evalúa el rendimiento de cada cromosoma, en función del tiempo de espera medio de los vehículos y sus emisiones de CO<sub>2</sub>.

**Población:** Se tiene una población de 12 miembros.

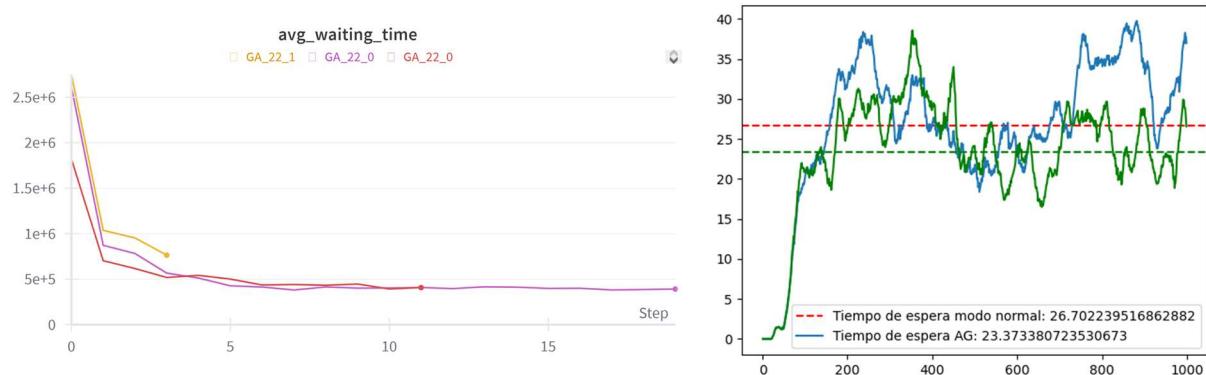
**Crossover:** Cruce aleatorio uniforme de dos padres. Se seleccionan dos padres de forma aleatoria dando mayor probabilidad a los padres con mejor desempeño según la función fitness. Cada gen del cromosoma del hijo puede ser del padre o de la madre, con una probabilidad del 50%. Se generan tantos hijos como el tamaño de la población, 12 en este caso.

**Mutación:** Una vez generados los nuevos hijos, se mutan sus cromosomas. Las duraciones se modifican añadiendo números aleatorios de una distribución normal con media=0 y desviación 8. Los estados de los semáforos se mutan sustituyendo los valores de un estado por otro estado aleatorio. El criterio de selección de genes a mutar es aleatorio para cada gen con una probabilidad fija del 20%.

**Selección de supervivientes:** La nueva generación sustituye por completo a la anterior.

## 5.2. ESTADÍSTICAS DEL ENTRENAMIENTO Y DESEMPEÑO

La duración del entrenamiento fue de 20 generaciones, pues a partir de la quinta generación la mejora por generación bajaba significativamente.



Como se puede observar, el desempeño del algoritmo genético mejora significativamente respecto al modelo inicial. No obstante, comparado con el controlador difuso, tiene un rendimiento mucho más pobre. Claramente esta diferencia se debe a la naturaleza de los controladores. Mientras que el genético ajusta los parámetros para que sean estáticos, el control difuso tiene una capacidad adaptativa que le permite adaptarse al flujo.

## 6. DISTRIBUCIÓN DEL TRABAJO Y REFERENCIAS

	Investigación de frameworks disponibles y su validación para el proyecto	Construcción de los mapas de prueba	Desarrollo de las reglas Fuzzy	Implementación del algoritmo de control en el entorno	Construcción de las pruebas de validación	Algoritmo genético	Escritura de la memoria
Pablo		X	X				X
David	X	X		X		X	X
Shota	X				X		X
Carlos		X	X				X

[1] SUMO User Documentation:  
<https://sumo.dlr.de/docs/index.html>

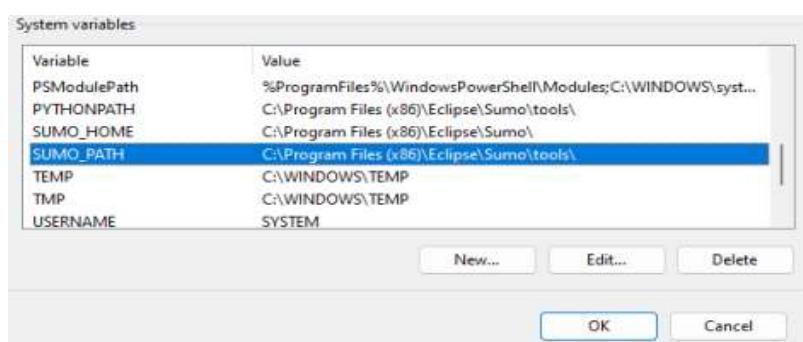
[2] Librería Scikit-fuzzy:  
<https://github.com/scikit-fuzzy/scikit-fuzzy>

[3] Libreria TraCi:  
<https://github.com/eclipse-sumo/sumo/blob/main/docs/web/docs/TraCI.md>

- [4] BilHim - Traffic Simulator:  
<https://github.com/BilHim/trafficSimulator>
  
- [5] Sumo-Riger-AG  
<https://github.com/geronimocharlie/SumoRinger/tree/master>
  
- [6] Dharma-AG  
<https://github.com/dharma9696/Traffic-Lights-Genetic-Algorithm>

## 7. ANEXO. INSTRUCCIONES DE INSTALACIÓN Y USO DEL PROGRAMA

La instalación del simulador se realiza desde la página oficial de Sumo, donde se instala el ejecutable. Es aconsejable seguir la guía oficial. Una vez instalados los programas, se configuran las rutas del entorno como en la guía o la siguiente imagen. Ha de configurarse la ruta SUMO\_PATH y SUMO\_HOME.



Una vez configuradas las rutas, se puede ejecutar el script de python que ejecutará el resto de los programas. Es importante asegurarse de que todas las librerías están correctamente instaladas. Durante el proyecto algunos entornos no detectaban la librería scikit-fuzzy. La solución es recurrir al entorno de anaconda, instalar las librerías en anaconda y seleccionar el kernel de python propuesto por anaconda. Llegado este punto, todo debería funcionar correctamente y se deberían ejecutar los archivos sin problema.

Dentro del directorio, existen 3 subdirectorios, uno para el cruce sencillo, otro para el mapa complejo y otro para el algoritmo genético. La forma de ejecutar el simulador consiste en ejecutar los archivos ControladorPasoCebra.py y ControladorIntersecciones.py.

Dentro de los scripts de python, es necesario ajustar la ruta de búsqueda del mapa, en función de la carpeta donde se descargue el archivo comprimido.

```
sumoBinary = "C:/Program Files (x86)/Eclipse/Sumo/bin/sumo-gui.exe"
sumoConfig = "C:/Users/User/Downloads/Code/Mapa Grande/tutorial.sumocfg"
```