Análisis y Reflexión.

Diego Herrera Olmos - A01652570

Modelación de sistemas multiagentes con gráficas computacionales

Profesores:
Dr. Sergio Ruiz Loza
Dr. David Christopher Balderas Silva

Fecha de entrega: 1 de diciembre del 2021

1. Análisis

1.1 Propuesta de solución

El problema que decidimos atacar fue el de los coches inteligentes, que tomaran la ruta menos congestionada, sin importar la distancia. Esto lo hacen obteniendo un destino al momento de ser inicializados, al llegar a él, se les da un nuevo destino y repite el comportamiento.

Nuestra solución contempla dos tipos de agentes que residirán en el ambiente. El primero es el de Coche, el principal. Este agente será inicializado al principio de la ejecución. Este pedirá datos al ambiente y con base en este decidirá su movimiento. El segundo agente, el TrafficIntersection, tendrá un par de semáforos, los cuales controlarán el flujo hacia un nodo. Finalmente, el ambiente controla el step de los agentes y hace la comunicación entre estos.

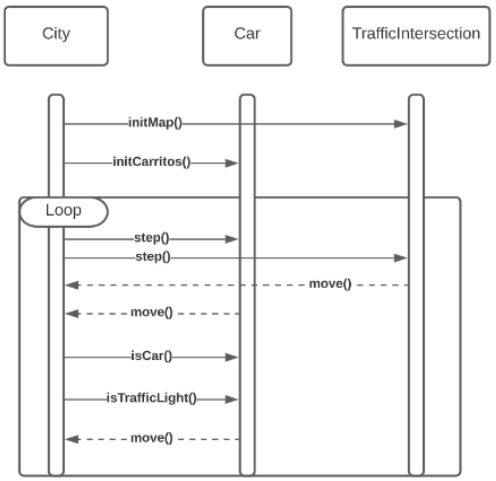

Fig 1. Diagrama de protocolos de interacción de la solución.

Se utilizó este modelo multiagentes debido a que se necesitaban de dos tipos de agentes, uno que se va moviendo y uno de control. Debido a que la posición de los agentes móviles, así como el estado de los agentes estáticos son importantes para determinar el movimiento de los coches, ambos agentes están suscritos al ambiente, y este sirve como puente para comunicar al agente con la información a su alrededor, sin hacer que dos agentes se comuniquen directamente y no haya un control agente-agente.

Decidimos modelar el ambiente como un grafo dirigido, el cual representa en escala 1 a 1 las calles de la mini ciudad. Cada nodo es un segmento de la calle por donde se pueden mover los coches. La dirección de las aristas indican el sentido de la calle.

Las variables que se tomaron en cuenta fueron la distancia entre el nodo actual y el destino, la posición y estado del semáforo y la posición de los coches para evitar colisiones. Estas variables son las necesarias para configurar el movimiento del coche utilizando una función heurística con A*.

las ventajas de haberlo hecho de esta forma, era que la navegación de los agentes era bastante sencilla, solamente se corría el algoritmo A* cuando el agente se encontraba con una decisión y gracias al modelado del grafo la complejidad no aumentaba mucho para correr el algoritmo. Además de que el envío de datos de python a Unity era transparente, no se debe hacer ningún post procesamiento al recibirlos y podían ser utilizados tal y como estaban.
Como desventajas, se podría incluir que no se toman en cuenta intersecciones más complejas, con diagonales o rotondas, debido a que se hace con grafos. Además que el mapa es fijo, definido por un archivo de texto. También, los semáforos al tener un elemento de aleatoriedad, afectan la eficiencia de la ruta de los coches.

a) Posibles mejoras
   El equipo al inicio optó por realizar un generador de mapas de manera procedural, que entregara un mapa distinto cada vez que se corriera el modelo, además de hacer todas las calles bidireccionales. Pero esto presentaba varios problemas con el diseño del grafo, por lo que se descartó por temas de tiempo. Nos gustaría volver a ese diseño y corregirlo para que la simulación fuera mucho más interesante.

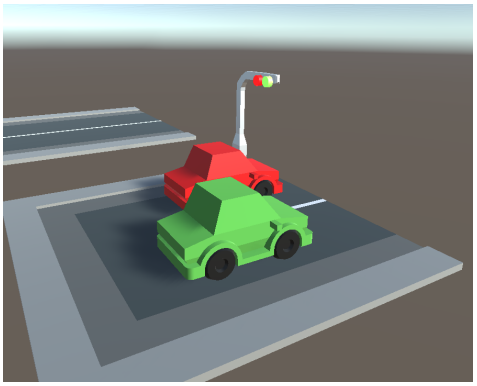

Fig 2. primera versión de los coches, calles y semáforos

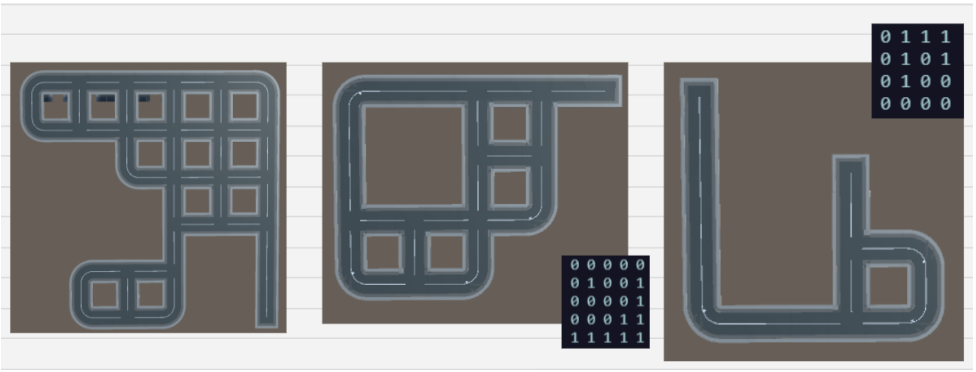

Fig 3. Mapas bidireccionales proceduralmente creados

También, se podría mejorar la implementación del algoritmo A* para que tomara en cuenta más variables y otorgara mejores decisiones, además de mejorar la complejidad temporal del algoritmo.

1.2 Modelado en python

A continuación se muestran los diagramas de clase de las clases utilizadas para los agentes. Se tiene City, que será el ambiente, Car y TrafficIntersection, los agentes, y TrafficLight, una clase auxiliar para manejar individualmente cada semáforo.

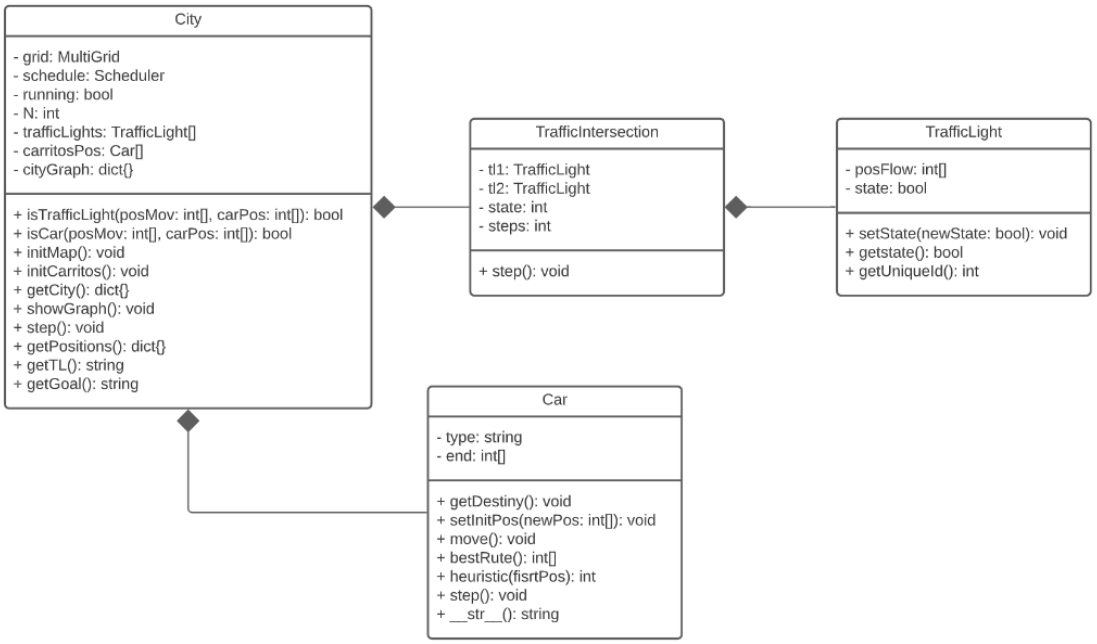

Fig 4. Diagrama de clases.

Se utilizaron las clases Agent y Model de la librería mesa. Esto, con el fin de tener acceso al scheduler y a las propiedades y métodos de estas clases, las cuales hacían más sencillo el trabajo de implementar el modelo.
Solamente Car y TrafficLight recibían la herencia de Agent, mientras que City heredaba de la clase Model.

Para la parte de la conexión, se cambió el archivo **hello.py** para agregar funcionalidad. Tenemos diferentes direcciones, las cuales mandan distintos tipos de datos. Sirven para inicializar el mapa, inicializar los agentes, recibir los datos de semáforos y recibir posiciones de los coches en cada step.

```python
from flask import Flask, render_template, request, jsonify
import json, logging, os, atexit
import PyModel


app = Flask(__name__, static_url_path='')
port = int(os.getenv('PORT', 8000))


@app.route('/main/<int:N>', methods=['GET', 'POST'])
def main(N):
    global city
    city = PyModel.City(N,10,10)
    city.showGraph()
    return "Ciudad creada"

@app.route('/position', methods=['GET', 'POST'])
def initialPos():
    result = city.getPositions()
    return jsonify(result)

@app.route('/step', methods=['GET', 'POST'])
def movement():
    result = city.step()
    return jsonify(result)

@app.route('/tl', methods=['GET', 'POST'])
def traffic():
    result = city.getTL()
    return result
```

Fig 5. Captura del script hello.py

En el step() el scheduler corre en orden aleatorio, los agentes suscritos a él. TrafficIntersection cada cierto número de steps (en este caso 4), elige de manera aleatoria un estado, así que cada cuatro steps tiene 50% de probabilidad de cambiar de estado.

El coche cada step va a revisar si llega al destino. En caso de que sí, se le asigna otro nodo destino de manera aleatoria, en caso de que no, avanza según las opciones que se tienen en su posición actual. Para esto, el agente cuando encuentra una intersección, corre una función que toma en cuenta dos cosas: g(x) y h(x). g(x) es la cantidad de coches que se encuentran entre su posición y la siguiente intersección según cada opción que puede tomar en ese momento. h(x) es la heurística, la cual toma la distancia pitagórica entre las opciones que puede tomar y el destino a alcanzar. Sumando el resultado de estas dos funciones, nos dan el costo de tomar cada nodo disponible. Se elige el de menor costo y a continuación, se revisa si existe un auto para evitar colisiones, además del estado del semáforo que controla el flujo al nodo con el costo menor, en caso de existir.

1.3 Gráficas computacionales en Unity

Unity se utilizó únicamente como herramienta para visualizar los datos que salían del modelo, además de una herramienta de apoyo para decorar el ambiente. Un script hace requests al servidor en IBM Cloud y recibe las posiciones de los agentes y el estado de los semáforos.
Se bajaron todos los modelos de internet, específicamente fueron los paquetes de modelos City Kit (Commercial) y City Kit (Roads) de Kenney.
https://www.kenney.nl/assets?q=3d
Elegimos estos modelos debido a que contaban con bajo conteo de polígonos, además de compartir un estilo gráfico y porque nos parecieron bonitos.

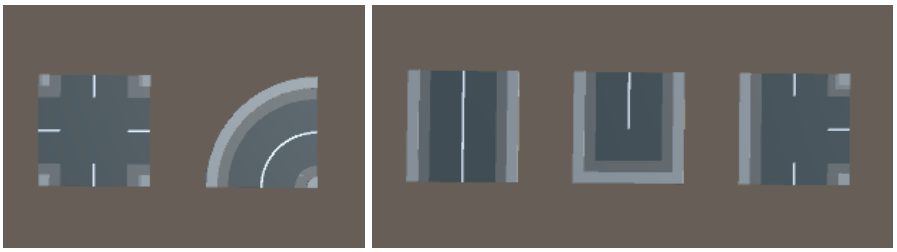

Fig 6. Modelos originales de las calles

Para los agentes, se instancia un coche y se le asigna un material con un color aleatorio. Se instancia en la coordenada (0,0). El movimiento de estos agentes es únicamente del mesh, por lo que el gameObject que los contiene no se mueve, permanece en todo momento en (0,0). Este movimiento se hace con matrices de transformación, interpolando la posición actual con la recibida y sincronizándolo con el tiempo de actualización de la posición para que se hiciera fluido y sin cortes ni teletransportaciones en el movimiento.
Decidimos cambiar los materiales de los edificios y calles, para darle una estética neon. Esto, debido a que queríamos hacerlo de noche, para que se notaran más las luces de los semáforos. El resultado fue una colección de colores brillantes y llamativos, combinados con el negro en el cuerpo de los edificios, suelo y skybox.

Fig 8. Vista aérea del mapa.

Se agregaron reflexiones de las luces en el asfalto, además de una textura normal en los postes, para hacer uso de las técnicas vistas en la clase de gráficas computacionales.

Fig 8. Muestra de mapeo UV y utilización de normal maps en los postes

Finalmente, como extra, se incluyeron diferentes cámaras y un pequeño script para cambiar entre estas y mostrar diferentes vistas del mapa, esto con el objetivo de apreciar bien cada rincón y poder ver más de cerca el comportamiento de los coches en diferentes puntos de la ciudad. Además, de agregó una acción para ocultar los postes de luz, decoraciones y edificios, dejando únicamente coches y calles, para que se pudiera apreciar solamente el funcionamiento del modelo, ignorando las decoraciones

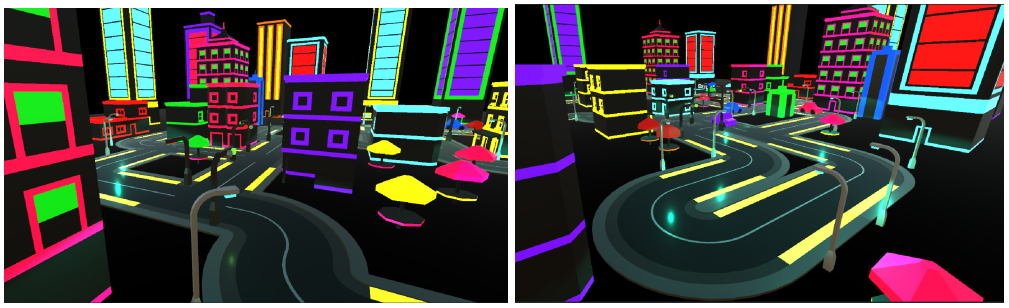

Fig 9. Muestra de distintas cámaras.

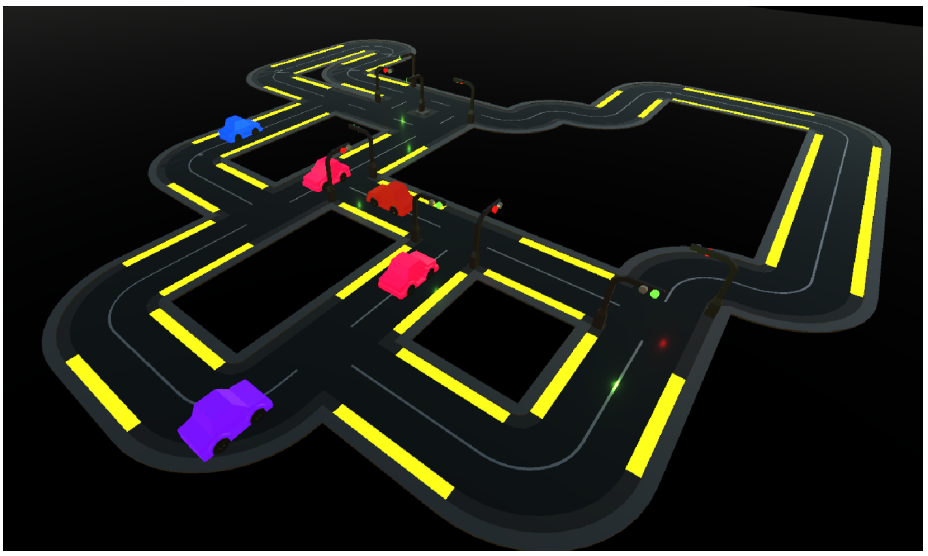

Fig 10. Muestra del ambiente en unity sin decoraciones

2. Reflexión

En realidad ha sido uno de los retos más divertidos que me ha tocado implementar. Teníamos ideas interesantes y aspiramos a hacer más, pero el tiempo no nos lo permitió. Mis expectativas estaban altas, pero creo que fueron superadas. Vimos mucho contenido, todo de valor e interesante, aunque lamentablemente no todo se pudo aplicar al reto. La situación que se ataca con el reto es interesante y existen muchas otras más que se podrían aplicar.

Finalmente, considero que por la cantidad de temas y la naturaleza de la parte de gráficas computacionales estaría excelente que durara más de 5 semanas.Incluso una colaboración con la carrera de LAD habría sido interesante, para que destacaran aún más los proyectos en la parte visual. Podría servir a ambas carreras para ver aplicado su campo.