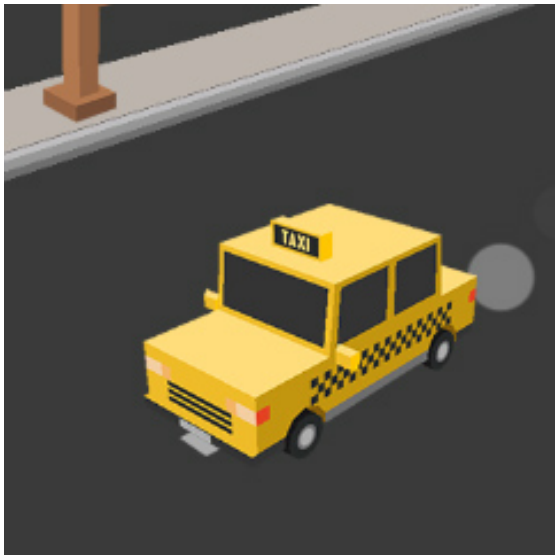
Arquitecturas Avanzadas de Computadores
Práctica 4

El juego del taxista



Autor:
Mikayel Mardanyan Petrosyan

27 de noviembre de 2019

Grado en Ingeniería Informática. Curso 2019/2020

Índice

1 Problema

1.1 Objetivo

El objetivo de la práctica es replicar el *juego del taxista*, en el que los taxis compiten por conseguir clientes a los que llevar a su destino. El primer taxista que recoja y lleve a su destino a 10 clientes gana el juego.

1.2 Restricciones

El juego tiene las siguientes restricciones:

- Se debe representar el entorno mediante una matriz de 50×50 que contenga a los clientes y los taxis.
- El número de clientes y taxis debe ser configurable al comenzar el juego. El número de taxis y clientes será siempre constante.
- Los taxis y clientes se crean en posiciones aleatorias al comenzar el juego.
- Solo puede haber un taxi en cada celda, los clientes sí pueden solaparse.
- Los clientes van andando, por lo que deben ser mucho más lentos que los taxistas en sus movimientos.
- Cada vez que un cliente llegue a su destino, ya sea llevado por un taxi o por su propio pie, este debe desaparecer del entorno y que sea creado uno nuevo en una posición aleatoria.
- Un taxi no puede llevar más de un cliente cada vez.
- Los movimientos permitidos son a las celdas adyacentes (también se puede mover en diagonal) incluida la casilla actual, respetando las restricciones anteriores.
- Se debe mostrar por pantalla cuando un cliente solicita un taxi, el movimiento del cliente y cuando un taxi recoge y deja a un cliente, identificando siempre al cliente y al taxi.
- Se deben utilizar procesos o hilos para simular los taxis y clientes.
- El acceso al entorno de juego por parte de los participantes debe ser en exclusión mutua.

2 Descripción de la implementación

2.1 Fundamentos teóricos

2.1.1 ¿Procesos o hilos?

Una de las decisiones más importantes que se debe tomar es cómo simular los participantes: mediante procesos o mediante hilos. Las librerías de procesos e hilos en Python son *multiprocessing* [1] y *threading* [2], respectivemente.

A la hora de decidir se deben tener en cuenta las siguientes características:

1. Los participantes comparten la matriz.

2. Los participantes son independientes y luchan por los recursos.

3. La matriz a compartir tiene un tamaño bastante grande.

Teniendo en cuenta lo anterior, lo más adecuado sería usar *threads* debido a que estos comparten memoria entre ellos, se pueden paralelizar y además son más ligeros que los procesos.

Lo que en otros lenguajes sería la solución perfecta, en Python va a ser un problema. Debido al *GIL* (*Global Interpreter Lock*, o *Mutex Global del Intérprete*) de Python, no se pueden realizar ejecuciones *multi-thread* aunque nuestra arquitectura sea compatible y nuestro software lo implemente [1] [3].

Las consecuencias son que al ejecutar un programa con n threads, aunque nuestra CPU tenga $n/2$ cores y pueda ejecutar n hilos, todos se van a ejecutar en un mismo core [4]. Por tanto, el rendimiento de un programa de un thread y de uno con varios thread será el mismo [5]. Este matiz puede llevar a pensar que los procesos encajan mejor para nuestro problema, ya que sí se pueden ejecutar multiples procesos al mismo tiempo. Esto sería cierto si no fuera porque la matriz a compartir es grande y compleja. Para compartir memoria entre procesos, lo cual no se recomienda, se debe implementar esta compartición mediante directivas específicas de la librería multiprocessing [6]. Sin embargo, solo se permiten compartir variables de tipos primitivos o vectores simples de tipos primitivos. Para compartir tipos complejos son necesarias las librerías *multiprocessing.shared_memory* [7] y *ctypes* [8]. Tras contemplar esta opción, se vio que la compartición de una matriz tan grande y compleja era inviable y la aceleración que se obtenía al ejecutar los procesos en distintos cores se veía reducida a la nada al compartir una estructura como nuestra matriz. Por ello, se tomó la decisión de usar *threading*.

2.1.2 Exclusión mutua

A la hora de implementar procesos o hilos debemos tener en cuenta que estos comparten la porción de memoria que almacena la matriz, y que pueden acceder a los mismos valores y que se produzcan condiciones de carrera. Para evitarlo usamos objetos mutex, implementados mediante Lock [9]. Los participantes competirán por obtener este mutex y solo cuando lo obtengan podrán realizar cambios en la matriz.

Para que la implementación sea lo más óptima posible, los hilos solamente bloquearán las celdas a las que sea posible moverse. Por tanto, durante la creación bloquearán todo el entorno usando un mutex global, ya que la generación de la posición inicial es aleatoria y el participante puede aparecer en cualquier punto; mientras que para el desplazamiento, bloquearán únicamente las celdas a las que puedan moverse. Estas son, en el caso de los clientes, las 8 adyacentes y la actual y, en el caso de los taxis, la actual y las adyacentes que no contengan ya un taxi. De este modo, no se producirá un bloqueo global y la ejecución será más rápida.

[1] Es por ello que la librería se llama *threading* y no *multithreading*.

2.2 Entorno

Para representar la matriz de 50×50 se ha creado una lista bidimensional. Cada posición de las listas anidadas se inicializa con una instancia de la clase *Celda*. Se ha creado esta clase para almacenar el contenido de cada posición.

En la inicialización del programa creamos un objeto *Lock* global que se empleará únicamente en la creación de taxis y clientes. A continuación, inicializamos la matriz con el tamaño pedido. Hay una variable *game_end* que controla el fin del juego. Esta variable vale 1 cuando se termina el juego, 0 en caso contrario. También hay un par de listas, *lista_taxis* y *lista_clientes*, que contendrán los procesos taxi y cliente, respectivamente.

2.3 Número de participantes

El número de clientes y taxis se pide por teclado y solo acepta números enteros positivos. Si se introduce cualquier otra cosa, se informa de ello y se vuelve a pedir.

2.4 Velocidad de los participantes

Para que los clientes se muevan más despacio se debe hacer que estos duerman más tiempo que los taxis. Para saber cuánto más deben dormir los clientes, y a modo de anécdota, se ha consultado la velocidad media de los taxis y peatones en la ciudad de Madrid. La velocidad media de los taxis es de 30km/h [10] y la velocidad media de los peatones es de 5km/h [11], es decir, los taxis van 6 veces más rápido, por lo que los clientes dormirán 6 veces más tiempo que los taxis.

2.5 Ejecución por línea de mandatos

La forma más cómoda de ejecutar el programa es usando la consola. Se debe ejecutar el siguiente mandato:

```
>$ python3.X Taxis.py [args]
```

donde los argumentos son opcionales y son los siguientes:

- **`--verbose`** o **`-v`**: Se muestran por pantalla todos los sucesos.
- **`--simple`** o **`-s`**: Se muestra por pantalla únicamente cuando un taxi recoge un cliente y le deja en su destino.

Solo se puede ejecutar con uno de estos argumentos. Si se ejecuta sin argumentos, además de imprimir los sucesos que se muestran en **`--simple`**, se muestran las posiciones de los clientes creados.

2.6 Clases implementadas

2.6.1 Clase Celda

La clase celda es capaz de almacenar un taxi y una lista de clientes y tiene como atributo un mutex para bloquearla.

```
class Celda:
    def __init__(self):
        # Variable que almacena el taxi
        self.taxi = None
        # Lista de clientes
        self.clientes = []
        # Mutex de la celda
        self.mutex = Lock()

    def add_taxi(self, taxi):
        """Introduce el taxi pasado por argumento en la
            celda."""
        self.taxi = taxi

    def remove_taxi(self):
        """Elimina el taxi existente en la celda."""
        self.taxi = None

    def add_cliente(self, cliente):
        """Introduce el taxi pasado por argumento en la
            celda."""
        self.clientes.append(cliente)

    def pop_cliente(self):
        """Saca uno de los clientes de la celda."""
        return self.clientes.pop(0)

    def remove_cliente(self, cliente):
        """Saca de la celda al cliente pasado como
            argumento."""
        self.clientes.remove(cliente)

    def lock_mutex(self):
        """Bloquea el mutex de la celda."""
        self.mutex.acquire()

    def release_mutex(self):
        """Libera el mutex de la celda."""
        self.mutex.release()
```

2.6.2 Clase Cliente

Constructor A continuación, observamos el constructor de la clase cliente. Los atributos de la clase son un identificador que se pasa por parámetro, un entero, que vale 0 si ha sido recogido por un cliente o 1 si sí ha sido recogido; un entero, que vale 1 si está vivo o 0 si no; y un par de tuplas origen y destino, con las coordenadas de origen y destino respectivamente, inicializadas a (0, 0) por defecto.

```
class Cliente:
    def __init__(self, id_):
        # Identificador del cliente
        self.id = id_
```

```
        # No ha sido recogido por ningun taxi
        self.taken = 0
        # El cliente esta vivo
        self.vivo = 1
        # Tupla con las coordenadas de origen
        self.origen = (0, 0)
        # Tupla con las coordenadas de destino
        self.destino = (0, 0)
```

Ciclo de vida Aquí se muestra el ciclo de vida del cliente dividido en 3 partes: lo que ocurre cuando comienza a vivir, lo que ocurre cuando está buscando un taxi y lo que ocurre cuando llega al destino a pie.

Cuando comienza a vivir obtiene posiciones aleatorias de origen y destino distintas entre sí y las almacena en sus atributos. Bloquea el mutex global y se mete en la matriz. Si no se ha ejecutado el programa en modo simple, muestra por pantalla dónde ha aparecido. Por último, libera el mutex global.

```
def vive(self):
    """Ciclo de vida del cliente."""

    # Bloquea el mutex global
    lock.acquire()
    # Genera posiciones aleatorias de origen y destino
    self.origen = random_position()
    self.destino = random_position()
    # Si el origen y el destino son iguales, vuelve a
        generar el destino hasta que sean distintos
    while self.origen == self.destino:
        self.destino = random_position()
    # Se mete en la matriz
    self.entrar()
    # Pide un taxi para que le lleve a su destino
    if verbose_mode != 2:
        self.pedir_taxi()
    # Libera el mutex global
    lock.release()
```

Mientras esté vivo y buscando un taxi, va a obtener las celdas adyacentes y bloquear los mutex de estos. Si en este tiempo no ha sido recogido, sale de la celda, elige una celda adyacente al azar y la guarda en su atributo origen. Este atributo se va a usar a partir de ahora para almacenar la posición actual del cliente. Se mete en la nueva celda y libera los mutex.

```
        # Mientras este vivo
        while self.vivo == 1:
            # Mientras no haya sido recogido ni haya llegado a su
                destino
            while self.taken == 0 and self.ha_llegado() == 0:
                sleep(0.6)

                # Obtiene las coordenadas de las celdas adyacentes
                celdas_ady = get_celdas_ady(self.origen[0],
                    self.origen[1])
```

```
# Bloquea el mutex en las celdas adyacentes
for (x, y) in celdas_ady:
   matriz[x][y].lock_mutex()

# Si no ha sido recogido por ningun taxi mientras
    bloqueaba los mutex
if self.taken == 0:
   # Sale de su celda
   self.salir()
   # Se calcula el siguiente movimiento de forma
       aleatoria
   self.origen = random_move(celdas_ady)
   # Se mueve a la nueva posicion
   self.entrar()

   # Si esta en modo verbose, imprime por pantalla
       el estado del cliente
   if verbose_mode == 1:
      self.mostrar_estado()

# Libera el mutex en las celdas adyacentes
for (x, y) in celdas_ady:
   matriz[x][y].release_mutex()
```

Si el cliente llega a pie a su destino, se imprime por pantalla y se modifican sus atributos para simular que es un cliente nuevo. Se incrementa el valor de la variable global que cuenta los clientes existentes y con este valor se asigna un nuevo identificador al cliente (como si fuera uno nuevo). Toma nuevas posiciones de origen y destino, bloquea el mutex global, se mete en la matriz y libera el mutex.

```
# Si llega a pie al destino
if self.ha_llegado() == 1:
   print("Soy {0} y he llegado a pie mi destino ({1},
       {2}). ".format(str(self.id),
       str(self.destino[0]), str(self.destino[1])))
   # Bloquea el mutex global
   lock.acquire()
   # Genera nuevas posiciones aleatorias de origen y
       destino
   self.origen = random_position()
   self.destino = random_position()
   # Si el origen y el destino son iguales, vuelve a
       generar el destino hasta que sean distintos
   while self.origen == self.destino:
      self.destino = random_position()
   global num_clientes
   # Incrementa el numero de clientes
   num_clientes += 1
   self.id = "Cliente " + str(num_clientes - 1)
   # Se mete en la matriz
   self.entrar()
   if verbose_mode != 2:
```

```
            self.pedir_taxi()
        # Libera el mutex global
        lock.release()
```

Funciones auxiliares La clase dispone de varios métodos. Estos están debidamente documentados mediante *docstring* y son fáciles de entender, por lo que no se van a explicar.

2.6.3 Clase Taxi

Constructor La clase taxi tiene como atributos un identificador, un entero que vale 1 si está ocupado, o 0 en caso contrario; un cliente, que será None hasta que recoja alguno y mientras esté libre; un entero que almacena el número de clientes llevados a su destino, o lo que es lo mismo, de carreras realizadas; y un par de tuplas origen y destino, con las coordenadas de origen y destino respectivamente, inicializadas a (0, 0) por defecto.

```
class Taxi:
    def __init__(self, id_):
        self.id = id_
        self.busy = 0
        self.origen = (0, 0)
        self.destino = (0, 0)
        self.cliente = None
        self.num_clientes = 0
```

Ciclo de vida El ciclo de vida del taxi se divide en 3 partes: lo que ocurre cuando aparece en el entorno, su comportamiento cuando está libre y su comportamiento cuando está ocupado.

Cuando aparece en el entorno genera posiciones de origen hasta obtener una en la que no haya un taxi y se mete en la celda, después de haber bloqueado el mutex global. Por último, libera el mutex.

```
def empieza_servicio(self):
    """Ciclo de vida del taxi. """

    # Genera una posicion aleatoria de origen
    origen = random_position()
    # Bloquea el mutex global
    lock.acquire()
    # Vuelve a generar la posicion de origen mientras en la
        generada haya ya un taxi
    while hay_taxi(origen):
        origen = random_position()
    # Toma como origen una posicion vacia
    self.origen = origen
    # Se mete en dicha posicion
    self.add_taxi()
    # Libera el mutex global
    lock.release()
```

A continuación, el taxi va a obtener las posiciones de las celdas adyacentes disponibles y las va a bloquear para ver si hay un cliente en la celda en la que se encuentra. Si lo hay, lo recoge y pasa a estar ocupado. Si no, sigue libre. Mientras esté libre va a bloquear las celdas adyacentes disponibles y va a comprobar si hay un taxi en alguna de estas celdas. Si es así, se va a mover a dicha celda y si no, se va a mover a una posición aleatoria. Si en la nueva posición hay un taxi, lo recoge. Por último, libera los mutex.

```
global game_end
# Mientras no se haya terminado el juego
while game_end == 0:

   # Obtiene las posiciones de las celdas adyacentes
       disponibles
   pos_coord = self.get_pos_coord()
   # Bloquea el mutex de dichas celdas adyacentes
   for (x, y) in pos_coord:
      matriz[x][y].lock_mutex()
   # Si hay un cliente en su celda, lo recoge
   if self.hay_cliente():
      self.coger_cliente()
   # Libera los mutex
   for (x, y) in pos_coord:
      matriz[x][y].release_mutex()

   # Mientras esta libre y el juego no ha terminado
   while self.busy == 0 and game_end == 0:
      sleep(0.1)
      # Obtiene las posiciones de las celdas adyacentes
          disponibles
      pos_coord = self.get_pos_coord()
      # Bloquea el mutex de dichas celdas adyacentes
      for (x, y) in pos_coord:
         matriz[x][y].lock_mutex()
      # Obtiene las coordenadas de uno de los clientes
          adyacentes, (-1, -1) si no hay
      adj_client = is_adjacent_client(pos_coord)
      # Si hay algun cliente en las adyacentes, toma la
          posicion de este para moverse a su celda
      if adj_client != (-1, -1):
         new_x = adj_client[0]
         new_y = adj_client[1]
      # Si no, se dirige a una al azar de las disponibles
      else:
         new = random.choice(pos_coord)
         new_x = new[0]
         new_y = new[1]
      # Sale de la celda
      self.remove_taxi()
      # Guarda la posicion de la celda a la que se va a
          mover
      self.origen = (new_x, new_y)
      # Se mete en la celda
      self.add_taxi()
```

```
# Si hay un cliente en la celda, lo recoge
if self.hay_cliente():
   self.coger_cliente()
else:
   if verbose_mode == 1:
      self.mostrar_estado()
# Libera los mutex
for (x, y) in pos_coord:
   matriz[x][y].release_mutex()
```

Mientras esté ocupado va a moverse hacia el destino del taxi hasta llegar a él. Cuando llegue se marcará libre y mostrará que ha llevado a un cliente a su destino. Después, incrementará su contador de clientes y si ha llegado a 10, ganará el juego. Si no ha llegado a 10, se creará otro cliente.

```
# Mientras esta ocupado
while self.busy == 1 and game_end == 0:
   sleep(0.1)
   # Mientras no haya llegado al origen
   while self.origen[0] != self.destino[0] or
       self.origen[1] != self.destino[1] and game_end
       == 0:
      # Obtiene las posiciones de las celdas
          adyacentes disponibles
      next_coord = self.get_pos_coord()
      # Bloquea el mutex de dichas celdas adyacentes
      for (x, y) in next_coord:
         matriz[x][y].lock_mutex()
      # Ordenamos las coordenadas por cercania al
          destino
      next_move = sorted(next_coord,
          key=self.euclidean_distance)[0]
      # Sale de la celda
      self.remove_taxi()
      # Guarda la posicion de la celda a la que se va
          a mover
      self.origen = (next_move[0], next_move[1])
      # Se mete en la celda
      self.add_taxi()

      if verbose_mode == 1:
         self.mostrar_estado_trayecto()
      # Libera los mutex
      for (x, y) in next_coord:
         matriz[x][y].release_mutex()

   # Si llega al destino
   if self.origen[0] == self.destino[0] and
       self.origen[1] == self.destino[1]:
      # Vuelve a estar libre
      self.busy = 0
      # Se suma un cliente al contador
      self.num_clientes += 1
```

```
lock.acquire()
# Muestra por pantalla que ha dejado al taxi y
    el numero de carreras que lleva realizadas
out = "Soy " + str(self.id) + " y dejo a " +
    str(self.cliente.id) + " en (" +
    str(self.origen[0]) \
     + ", " + str(self.origen[1]) + "), he
         realizado "
if self.num_clientes != 1:
   out += str(self.num_clientes) + " carreras. "
else:
   out += " 1 carrera."
# No lleva ningun cliente
self.cliente = None
print(out)

# Si ha conseguido 10 clientes
if self.num_clientes == 10:
   # Informa de que ha ganado
   print("SE ACABO EL JUEGO. EL GANADOR ES {0}.
       ".format(str(self.id)))
   # Se acaba el juego
   game_end = 1
# Si ha dejado a un cliente pero aun no ha
    ganado, crea uno nuevo
else:
   crear_cliente()
lock.release()
```

Funciones auxiliares La clase dispone de muchas funciones auxiliares, vamos a comentar las dos más importantes.

La primera función nos devuelve una lista de tuplas con las coordenadas de las celdas adyacentes a las que se puede mover el taxi. Estas son las celdas que no contienen ningún taxi y la celda en la que se encuentra actualmente. La función obtiene primero todas las celdas adyacentes y después filtra en el bucle las que cumplen las condiciones anteriores.

```
def get_pos_coord(self):
   """Devuelve las celdas adyacentes a las que sea posible
       moverse."""
   # Obtiene todas las celdas adyacentes
   celdas_ady = get_celdas_ady(self.origen[0],
       self.origen[1])
   ret = []
   # Guarda en la lista aquellas que no tienen un taxi y la
       celda actual
   for (x, y) in celdas_ady:
      if matriz[x][y].taxi is None or matriz[x][y].taxi ==
          self:
         ret.append((x, y))
   return ret
```

El segundo método es el encargado de recoger a un cliente. Marca al cliente

como recogido y como no vivo, de este modo no cumplirá las condiciones de su ciclo de vida necesarias para seguir viviendo y morirá. El taxi pasa a estar ocupado y almacena al cliente en uno de sus atributos. Adquiere el destino del cliente para empezar a moverse a él e informa del suceso por pantalla.

```
def coger_cliente(self):
    """Recoge a un cliente de su celda, sacando a este de su
        celda y marcandolo como no vivo. El taxi pasa a estar
        ocupado y toma como destino el destino del cliente.
        Imprime por pantalla que ha recogido al cliente."""
    # Esta ocupado
    self.busy = 1
    # Saca al cliente de la celda
    cl = self.remove_cliente()
    # El cliente ha sido recogido
    cl.taken = 1
    # El cliente deja de estar vivo
    cl.vivo = 0
    # Guarda al cliente
    self.cliente = cl
    # Adquiere el destino del cliente
    self.destino = (cl.destino[0], cl.destino[1])
    # Informa por pantalla
    print("Soy {0} y cogi a {5} en ({1}, {2}), le llevo a
        ({3}, {4})".format(str(self.id), str(self.origen[0]),
        (self.origen[1]), (self.destino[0]),
        (self.destino[1]), str(cl.id)))
```

2.7 Creación de clientes y taxis

En el *main* del programa se inicializa el mutex global, la matriz, se crean listas para almacenar los hilos taxi y cliente. Se controlan los argumentos de la función cuando es ejecutada por línea de comandos y se pide el número de taxis y clientes por teclado. Una vez tenemos estos datos, creamos un hilo para cada marticipante mediante las funciones *constructor* definidas y etiquetamos los hilos como *daemon*, para que mueran cuando lo haga el programa padre. A continuación, se da vida a los hilos y el programa se queda en un bucle infinito hasta que se modifique la variable de fin de juego.

```
def constructor_cliente(id_):
    """Crea un cliente con el id pasado por parametro y le da
        vida"""
    c_ = Cliente(id_)
    c_.vive()


def constructor_taxi(id_):
    """Crea un taxi con el id pasado por parametro y lo pone a
        trabajar"""
    t_ = Taxi(id_)
    t_.empieza_servicio()
```

```
if __name__ == '__main__':

    # Crea el mutex
    lock = Lock()
    # Dimension de la matriz
    m = 50
    # Inicializa la matriz
    matriz = [[Celda() for x in range(m)] for y in range(m)]
    # Variable que controla el fin del juego
    game_end = 0
    # Lista que contiene los taxis
    lista_taxis = []
    # Lista que contiene los clientes
    lista_clientes = []

    # Modo verbose, que permite ver por pantalla cualquier cosa
        que suceda en el entorno
    verbose_mode = 0
    # Se activa si ejecutamos el programa con un argumento
        especifico
    if len(argv) > 1:
       if argv[1] == "--verbose" or argv[1] == "-v":
          verbose_mode = 1
       elif argv[1] == "--simple" or argv[1] == "-s":
          verbose_mode = 2

    print("\n###### EL JUEGO DEL TAXISTA ######\n")
    # Pide el numero de taxis por pantalla
    num_taxis = input('Cuantos taxis quieres que haya? \n')
    # Mientras no se introduzca un digito
    while not num_taxis.isdigit():
       # se informa del error
       print("El caracter introducido no es un numero entero
           positivo, vuelve a intentarlo. ")
       # y se vuelve a pedir
       num_taxis = input('Cuantos taxis quieres que haya? \n')
    num_taxis = int(num_taxis)
    # Pide el numero de clientes por pantalla
    num_clientes = input('Cuantos clientes quieres que haya?
        \n')
    while not num_clientes.isdigit():
       print("El caracter introducido no es un numero entero
           positivo, vuelve a intentarlo. ")
       num_clientes = input('Cuantos clientes quieres que haya?
           \n')
    num_clientes = int(num_clientes)
    print()

    # Crea los procesos Cliente
    for c in range(num_clientes):
       id_c = "Cliente " + str(c)
       # Crea el hilo con el cliente
       lista_clientes.append(Thread(target=constructor_cliente,
           args=(id_c,)))
```

```
# Crea los procesos Taxi
for t in range(num_taxis):
   id_t = "Taxi " + str(t)
   # Crea el hilo con el taxi
   lista_taxis.append(Thread(target=constructor_taxi,
       args=(id_t,)))

for clien in lista_clientes:
   # Declaramos los hilos como daemon
   clien.daemon = True
   # y les damos vida
   clien.start()

for tax in lista_taxis:
   # Declaramos los hilos como daemon
   tax.daemon = True
   # y les damos vida
   tax.start()

# El programa sigue vivo mientras no se cambie el valor de
    la variable
while game_end == 0:
   pass
```

3 Conclusión

Los hilos de Python nos permiten crear agentes independientes que compitan por los recursos y, aunque estos no se puedan paralelizar, son una buena aproximación. La prática ha sido entretenida y ha aportado mucho.

References

[1] Documentacion de Python, *Multiprocessing*.
Fuente: https://docs.python.org/3/library/multiprocessing.html

[2] Documentacion de Python, *Threading*.
Fuente: https://docs.python.org/3/library/threading.html

[3] Abhinav Ajitsaria *What is the Python Global Interpreter Lock (GIL)?. Real Python*. Fuente: https://realpython.com/python-gil/

[4] EngineerMan, *Threading vs Multiprocessing in Python*. Fuente: https://www.youtube.com/watch?v=ecKWiaHCEKs

[5] Jitender Kumar *What is the Python Global Interpreter Lock (GIL). GeeksforGeeks*. Fuente: https://www.geeksforgeeks.org/what-is-the-python-global-interpreter-lock-gil/

[6] Documentacion de Python, *Multiprocessing Shared Memory*.
Fuente: https://docs.python.org/3/library/multiprocessing.shared_memory.html

[7] Documentacion de Python, *Shared ctypes*.
Fuente: https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.sharedctypes

[8] Documentacion de Python, *ctypes*.
Fuente: https://docs.python.org/3/library/ctypes.html

[9] Documentacion de Python, *Lock Object*.
Fuente: https://docs.python.org/3/library/threading.html#lock-objects

[10] D. G. de Gestión y Vigilancia de la Circulación - S. G. de Regulación de la Circulación y del Servicio del Taxi (2017) *Estudio del servicio del taxi. Ayuntamiento de Madrid*. Fuente: https://www.madrid.es/UnidadesDescentralizadas/UDCMovilidadTransportes/TAXI/Ficheros/Otros/Ayto%20Madrid%20%C2%B7%20Estudio%20del%20Servicio%20del%20Taxi.pdf

[11] Muévete por Madrid. (2013) *Velocidad desplazamiento a pie. EMT Madrid*.
Fuente: http://www.muevetepormadrid.es/peaton.html

[12] Documentacion de Python, `sys.argv`.
Fuente: https://docs.python.org/3/library/sys.html#sys.argv