# Congestion en milieu urbain
# fluidifier le trafic grâce au routage dynamique

Martin Pupat - 26609 - MPI

## Code utilisé pour le TIPE

```python
# TIPE Martin Pupat - 26609 - Code Python #
# Exécution : python3 geo.py <voitures> <ballstring> #
# <voitures>:int -> nombre de voitures #
# <ballstring>:bool -> utilisation dudit algorithme #

# IMPORTATION DES BIBLIOTHÈQUES #

# Lecture des données du fichier .geojson #

import json
import fiona
import geopandas
import matplotlib.pyplot as plt

# Calculs mathématiques #

from geographiclib.geodesic import Geodesic
import math
import numpy as np
import random
import time
from statistics import mean

# Lecture / écriture de fichiers externes #

import sys

# Représentation visuelle du graphe routier #

from pylab import Line2D, gca

# VARIABLES GLOBALES #

rnd = 6 # Arrondissement des coordoonnées des noeuds

tick = float(1) # Valeur d'une « seconde »

car_number = int(sys.argv[1]) # Nombre de voitures

bs = False # Utilisation de l'algotihme BallString
if sys.argv[2] == "True":
  bs = True
```

```
43
44  tps = float(0) # Valeur initiale du temps
45
46  end_time = 7200 # Temps maximum de départ des voitures
47
48  random.seed(1) # Graine pour recréer les résultats
49  np.random.seed(1)
50
51  # CLASSES D'OBJETS #
52
53  # Car : voiture dans le graphe #
54  class Car:
55    def __init__(self, city, start, end): # Initialisation de la voiture et du chemin parcouru
56      self.start_time = float(min(end_time, max(0, np.random.normal(end_time / 2, end_time / 6)))) #
          Temps de départ suivant une gaussienne
57      self.times_stuck = 0 # Nombre de fois où la voiture a été coincée (déboguage)
58      self.optimal_time = astar(city, start, end, False)[1] # Temps que mettrait la voiture sur un
          graphe routier vide
59      self.done = False # Vrai si la voiture est arrivée à destination
60      self.current_edge = None # Arête sur laquelle la voiture est
61      self.edge_duration = 0 # Temps pour lequel la voiture va rester sur l'arête
62      self.start = start # Noeud de départ
63      self.end = end # Noeud d'arrivée
64      self.ongoing_path = astar_queue(city, start, end, False) # Chemin que la voiture prévoit de
          suivre
65      del(self.ongoing_path[0])
66      self.elapsed_path = [start] # Chemin que la voiture a déjà parcouru
67      self.trip_time:float = 0 # Temps que le trajet a duré
68      if start == end:
69        self.done = True
70      else:
71        self.current_edge = city.edge_to(start, self.ongoing_path[0])
72        self.edge_duration = city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].time
73        city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].add_car() # Indiquer que la
            voiture est sur l'arête
74
75    def stuck_ahead(self, city): # Renvoie vrai ssi aucune des arêtes du chemin prévu n'est saturée
76      res = []
77      for i in range(len(self.ongoing_path)-1):
78        if not(city.nodes[self.ongoing_path[i]].edges_out[city.edge_to(self.ongoing_path[i], self.
            ongoing_path[i+1])].available()):
79          res.append(i)
80      return res
81
82    def update_ballstring(self, city): # Mise à jour de la voiture selon l'algorithme BallString
83      if len(self.ongoing_path) > len(city.nodes):
84        raise Exception("Overflow!")
85      if self.start_time < tps: # Pas encore l'heure de commencer
86        pass
87      if self.done: # Exécution finie
88        pass
89      elif self.edge_duration > tick: # La voiture continue sur l'arête
90        self.trip_time += tick
91        self.edge_duration -= tick
```

```
    elif len(self.ongoing_path) == 1: # La voiture est sur la fin de sa dernière arête
      self.trip_time += self.edge_duration
      tplus = tick - self.edge_duration
      self.edge_duration = 0
      self.ongoing_path = []
      city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].remove_car()
      self.elapsed_path.append(self.end)
      self.current_edge = None
      self.done = True
    else: # La voiture fait un changement d'arête
      nxt = city.edge_to(self.ongoing_path[0], self.ongoing_path[1])
      blocked = self.stuck_ahead(city)
      if blocked == []: # Aucun problème devant
        self.trip_time += tick
        tplus = tick - self.edge_duration
        city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].remove_car()
        self.current_edge = nxt
        self.elapsed_path.append(self.ongoing_path.pop(0))
        city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].add_car()
        self.edge_duration = city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].time -
            tplus
      else: # Un problème devant
        tmp = self.ongoing_path
        try:
          greffon = astar_queue(city, self.ongoing_path[blocked[0]], self.ongoing_path[blocked[-1]],
              True)
        except: # Impossible de trouver un autre chemin
          self.times_stuck += 1
          self.ongoing_path = tmp
          nxt = city.edge_to(self.ongoing_path[0], self.ongoing_path[1])
          if city.nodes[self.ongoing_path[0]].edges_out[nxt].available():
            self.trip_time += tick
            tplus = tick - self.edge_duration
            city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].remove_car()
            self.current_edge = nxt
            self.elapsed_path.append(self.ongoing_path.pop(0))
            city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].add_car()
            self.edge_duration = city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].time
                - tplus
          else:
            self.trip_time += tick
        else: # Possible de trouver un autre chemin
            self.ongoing_path = greffe(tmp, greffon, city)
            self.trip_time += tick
            tplus = tick - self.edge_duration
            city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].remove_car()
            self.current_edge = nxt
            self.elapsed_path.append(self.ongoing_path.pop(0))
            city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].add_car()
            self.edge_duration = city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].time
                - tplus

```

```
  def update_naive(self, city): # Mise à jour de la voiture selon l'algorithme A* itéré
    if self.start_time < tps:
      pass
    if self.done:
      pass
    elif self.edge_duration > tick:
      self.trip_time += tick
      self.edge_duration -= tick
    elif len(self.ongoing_path) == 1:
      self.trip_time += self.edge_duration
      self.edge_duration = 0
      self.ongoing_path = []
      city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].remove_car()
      self.elapsed_path.append(self.end)
      self.current_edge = None
      self.done = True
    else:
      nxt = city.edge_to(self.ongoing_path[0], self.ongoing_path[1])
      if city.nodes[self.ongoing_path[0]].edges_out[nxt].available():
        self.trip_time += tick
        tplus = tick - self.edge_duration
        city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].remove_car()
        self.current_edge = nxt
        self.elapsed_path.append(self.ongoing_path.pop(0))
        city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].add_car()
        self.edge_duration = city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].time -
            tplus
      else:
        tmp = self.ongoing_path
        try:
          self.ongoing_path = astar_queue(city, self.ongoing_path[0], end, True)
        except:
          self.times_stuck += 1
          self.trip_time += tick
          self.ongoing_path = tmp
        else:
          self.trip_time += tick
          tplus = tick - self.edge_duration
          city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].remove_car()
          self.current_edge = nxt
          self.elapsed_path.append(self.ongoing_path.pop(0))
          city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].add_car()
          self.edge_duration = city.nodes[self.elapsed_path[-1]].edges_out[self.current_edge].time -
               tplus

```

```
# Edge : arête reliant deux sommets #

class Edge:
 # Pas besoin d'enregistrer le noeud de départ puisqu'une arête est dans un tableau depuis le noeud de départ
  def __init__(self, start_node, end_node, speed, lanes): # Initialisation de l'arête
    self.end_node:int = end_node # Noeud d'arrivée
    self.speed = speed # Vitesse max
    self.lanes = lanes # Nombre de voies
    self.length = Geodesic.WGS84.Inverse(ville.nodes[start_node].lat, ville.nodes[start_node].lon, ville.nodes[end_node].lat, ville.nodes[end_node].lon, 1025)['s12'] # Distance de la rue, calculée avec la bibliothèque géodésique
    self.time = float(self.length)/float(self.speed) * 3.6 # Temps de parcours de la rue
    self.capacity = math.ceil((float(self.length)/float(self.speed)) * 2 * lanes) # Capacité de la rue (en tenant compte des distances de sécurité)
    self.occupied = 0 # Nombre de voiitures actuellement sur l'arête
    self.min_available = self.capacity-self.occupied # Place libre minimum dont a disposé l'arête (déboguage)

  def available(self): # Renvoie vrai si et seulement si l'arête n'est pas saturée
    return (self.capacity - self.occupied > 0)

  def add_car(self): # Applique l'ajout d'une voiture sur l'arête
    self.occupied += 1
    if self.min_available > self.capacity-self.occupied:
      self.min_available = self.capacity-self.occupied

  def remove_car(self): # Applique le retrait d'une voiture sur l'arête
    self.occupied -= 1

# Node : sommet du graphe #

class Node:
  def __init__(self, lat, lon): # Initialisation du sommet
    self.lat:int = lat # Latitude
    self.lon:int = lon # Longitude
    self.edges_out = [] # Tableau d'arêtes sortantes
    self.edges_in = [] # Tableau d'arêtes entrantes

# Road_graph : le graphe routier en question #

class Road_graph:
  def __init__(self): # Initialisation de la ville
    self.nodes = [] # Tableau de sommets

  def edge_to(self, n, end): # Renvoie le numéro de l'arête depuis n qui relie end
    for i in range(len(self.nodes[n].edges_out)):
      if self.nodes[n].edges_out[i].end_node == end:
        return i
    raise Exception("Invalid edge !")

  def neighbors(self, n): # Renvoie la liste des sommets adjacents à n
    res = []
    for i in range(self.outgoing_edges(n)):
      res.append([self.nodes[n].edges_out[i].end_node, self.nodes[n].edges_out[i].length])
    return res
```

```
234
235   def heuristic(self, i, j): # Renvoie le carré de la distance angulaire à vol d'oiseau entre les
          sommets i et j pour servir d'heuristique à A*
236     return (self.nodes[i].lat - self.nodes[j].lat) ** 2 + (self.nodes[i].lon - self.nodes[j].lon)
            ** 2
237
238   def coordinates(self, i): # Renvoie les coordonnées du noeud i
239     return [self.nodes[i].lat, self.nodes[i].lon]
240
241   def find_node(self, lat, lon): # Renvoie l'index du sommet correspondant ou -1 en cas d'échec
242     for i in range(0, len(self.nodes)):
243       if abs(self.nodes[i].lat - lat) < 10**(-rnd):
244         if abs(self.nodes[i].lon - lon) < 10**(-rnd):
245           return i
246     return -1
247
248   def neighbor(self, start, end): # Vérifie si la fin est voisine du début
249     for i in range(len(self.nodes[start].edges_out)):
250       if (self.nodes[start].edges_out[i].end_node == end):
251         return True
252     return False
253
254   def capacity(self): # Renvoie la capacité totale du graphe (déboguage)
255     res = 0
256     for i in range(len(self.nodes)):
257       for j in range(len(self.nodes[i].edges_out)):
258         res += self.nodes[i].edges_out[j].capacity
259     return res
260
261   def add_node(self, lat, lon): # Rajoute un sommet s'il n'existe pas déjà
262     if self.find_node(lat, lon) == -1:
263       self.nodes.append(Node(lat, lon))
264
265   def add_oneway_edge(self, speed, lanes, coordinates): # Ajoute une arête entre deux coordonnées
266     nodes = []
267     for i in range(len(coordinates)):
268       nodes.append(self.find_node(coordinates[i][1], coordinates[i][0]))
269     for j in range(0, len(nodes)-1):
270       if not (self.neighbor(nodes[j], nodes[j+1])):
271         self.nodes[nodes[j]].edges_out.append(Edge(nodes[j], nodes[j+1], speed, lanes))
272         self.nodes[nodes[j+1]].edges_in.append(nodes[j])
273
274   def add_twoway_edge(self, speed, lanes_backward, lanes_forward, coordinates): # Ajoute une arête
        dans les deux sens
275     self.add_oneway_edge(speed, lanes_forward, coordinates)
276     coordinates.reverse()
277     self.add_oneway_edge(speed, lanes_backward, coordinates)
278
```

```
279   def add_edge(self, coordinates, data): # Créée une arête avec les données reçues
280     speed = int(data.get("maxspeed", 30))
281     lanes = int(data.get("lanes", 2))
282     lanes_backward = int(data.get("lanes_backward", lanes/2))
283     lanes_forward = int(data.get("lanes_forward", lanes/2))
284     oneway = (data.get("oneway", "no") == "yes")
285     if oneway:
286       self.add_oneway_edge(speed, lanes, coordinates)
287     else:
288       self.add_twoway_edge(speed, lanes_backward, lanes_forward, coordinates)
289 
290   def outgoing_edges(self, i): # Renvoie le nombre d'arêtes sortantes d'un sommet
291     return len(self.nodes[i].edges_out)
292 
293   def incoming_edges(self, i): # Renvoie le nombre d'arêtes entrantes d'un sommet
294     return len(self.nodes[i].edges_in)
295 
296   def connexes(self): # Tableau des composantes
297     queue = []
298     res = [-1]*len(self.nodes)
299     cc = -1
300     for start in range(len(self.nodes)):
301       if (res[start] < 0):
302         cc += 1
303         queue.append(start)
304         while (len(queue) > 0):
305           active = queue[0]
306           del queue[0]
307           res[active] = cc
308           for i in range(len(self.nodes[active].edges_out)):
309             if res[self.nodes[active].edges_out[i].end_node] < 0:
310               queue.append(self.nodes[active].edges_out[i].end_node)
311     return res
312 
313   def remove_node(self, n): # Supprime un noeud du graphe et met à jour les indices et les arêtes
314     del(self.nodes[n])
315     for i in range(len(self.nodes)):
316       l = len(self.nodes[i].edges_out)
317       j = 0
318       while j < l:
319         if self.nodes[i].edges_out[j].end_node == n:
320           del (self.nodes[i].edges_out[j])
321           l -= 1
322         elif self.nodes[i].edges_out[j].end_node > n:
323           self.nodes[i].edges_out[j].end_node -= 1
324           j += 1
325         else:
326           j += 1
327 
328   def single_connex(self): # Élague le graphe de toutes ses composantes connexes sauf la plus
       grande
329     connex = self.connexes()
330     maj = max(set(connex), key=connex.count)
331     for i in range(len(connex)-1, -1, -1):
332       if connex[i] != maj:
333         self.remove_node(i)
```

```
334
335 # Proximity_queue : file d'attente pour un parcours de graphe #
336
337 class proximity_queue:
338   def __init__(self, size): # Créer le tas
339     self.size = size # Nombre de noeuds au maximum
340     self.heap = [] # Le tas en lui-même
341     self.fathers = [-1]*self.size # Les pères de chacun des noeuds dans le parcours
342     self.index = [math.inf]*self.size # La distance de chacun des noeuds
343
344   def parent(self, key): # Renvoie le parent de l'élément dans la file
345     return (key-1)//2
346
347   def swap(self, i, j): # Échange deux éléments de la file
348     self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
349
350   def is_before(self, v, g, city): # Teste si self.heap[v] < self.heap[g]
351     i = self.heap[v]
352     j = self.heap[g]
353     return (self.get_index(i) + city.heuristic(i, end) < self.get_index(j) + city.heuristic(j, end
          ))
354
355   def set_father(self, key, father): # Change le père d'un élément
356     self.fathers[key] = father
357
358   def get_father(self, key): # Renvoie le père d'un élément
359     return self.fathers[key]
360
361   def set_index(self, key, val): # Initialise la distance d'un sommet
362     self.index[key] = val
363
364   def get_index(self, key): # Renvoie la distance d'un sommet
365     return self.index[key]
366
367   def is_empty(self): # Vérifie si la file est vide
368     return (len(self.heap) == 0)
369
370   def descent(self, i): # Ajoute un élément à la fin de la file
371     l = 2 * i + 1
372     r = 2 * i + 2
373     first = i
374     if (l < len(self.heap) and self.is_before(l, i)):
375       first = l
376     if (r < len(self.heap) and self.is_before(r, first)):
377       first = r
378     if (first != i):
379       self.swap(first, i)
380       self.descent(first)
381
```

```
  def insert(self, key, val, father, city): # Insère un élément dans la file
    if(self.get_index(key) > val):
      self.set_index(key, val)
      self.set_father(key, father)
      if key in self.heap:
        self.heap.remove(e)
      i = len(self.heap)
      self.heap.append(key)
      while (i != 0):
        p = self.parent(i)
        if self.is_before(p, i):
          self.swap(p, i, city)
        i = p

  def extract(self): # Renvoie l'élément en tête de file
    if len(self.heap) == 0:
      return None
    r = self.heap[0]
    self.heap[0] = self.heap[-1]
    del self.heap[-1]
    self.descent(0)
    return r

  def ancestry(self, key): # Renvoie le chemin pour aller jusqu'à l'élément
    r = [key]
    while (self.get_father(key) != key):
      r.append(self.get_father(key))
      key = self.get_father(key)
      return r.reverse()

# ALGORITHME DE PARCOURS #

def astar_queue(city, start, end, av): # Algorithme A* de start à end, en utilisant une file de
    priorité. Si av=True, ne cherche que parmi les arêtes non saturées.
  queue = proximity_queue(len(ville.nodes))
  queue.insert(start, 0, start)
  while (not(queue.is_empty())):
    current = queue.extract()
    so_far = queue.get_index(current)
    for i in range(len(city.nodes[current].edges_out)):
      if (city.nodes[current].edges_out[i].available) or (not av):
        d = queue.get_index(current) + city.nodes[current].edges_out[i].length
        e = city.nodes[current].edges_out[i].end_node
        queue.insert(e, d, current, city)
  if queue.get_index(end) == math.inf:
    raise Exception("No path found !")
  res = ancestry(end)
  return (res, math.inf)
```

```
430 def astar(city, start, end, av): # Algorithme A* de start à end qui renvoie le chemin et le temps
        de parcours prévu. Si av=True, ne cherche que parmi les arêtes non saturées.
431   queue = [start]
432   fathers = [-1]*len(city.nodes)
433   fathers[start] = start
434   dist = [math.inf]*len(city.nodes)
435   dist[start] = 0
436   def swap(i):
437     if i > 0:
438       if dist[queue[i]] + city.heuristic(queue[i], end) < dist[queue[i-1]] + city.heuristic(queue[i
            -1], end):
439         queue[i], queue[i-1] = queue[i-1], queue[i]
440         swap(i-1)
441   while queue != []:
442     current = queue.pop(0)
443     for i in range(len(city.nodes[current].edges_out)):
444       if (city.nodes[current].edges_out[i].available) or (not av):
445         d = dist[current] + city.nodes[current].edges_out[i].length
446         e = city.nodes[current].edges_out[i].end_node
447         if (dist[e] > d):
448           fathers[e] = current
449           dist[e] = d
450           if e in queue:
451             queue.remove(e)
452           queue.append(e)
453           swap(len(queue)-1)
454   if dist[end] == math.inf:
455     raise Exception("No path found !")
456   res = [end]
457   t = 0
458   current = end
459   while current != start:
460     res.append(fathers[current])
461     t += city.nodes[fathers[current]].edges_out[city.edge_to(fathers[current], current)].time
462     current = fathers[current]
463   res.reverse()
464   return (res, t)
465
466 def ended_cars(cars): # Vérifie si toutes les voitures ont toutes fini leur trajet
467   for i in range(len(cars)):
468     if not cars[i].done:
469       return False
470   return True
471
```

```
472 def greffe(base, bouture, city): # Greffe une liste dans une autre à l'endroit du début de la
        bouture
473  # Coût : O(e*m) avec e la distance de contournement
474   racine = bouture[0]
475   cime = bouture[-1]
476   longueur = 0
477   initial = 0
478   del bouture[0]
479   while base[initial] != racine:
480    initial += 1
481   final = initial
482   while base[final] != cime:
483    final += 1
484   if (base[initial] != racine) or (base[final] != cime):
485    raise Exception("Error!")
486   if bouture != []:
487    del bouture[-1]
488    longueur = len(bouture)
489    base = base[0:initial+1] + bouture + base[final:]
490   if (base[initial] != racine) or (base[final] != cime):
491    raise Exception("Bad greffe!")
492   return base
493 
494 # FONCTIONS DE DÉBOGUAGE #
495 
496 def print_city(city): # Affichage graphique du réseau routier une fois importé (déboguage)
497   edges = []
498   x = []
499   y = []
500   for i in range(len(city.nodes)):
501    y.append(city.nodes[i].lat)
502    x.append(city.nodes[i].lon)
503    for j in range(len(city.nodes[i].edges_out)):
504     edges.append((i, city.nodes[i].edges_out[j].end_node))
505   plt.figure()
506   ax = gca()
507   for edge in edges:
508    ax.add_line(Line2D([x[edge[0]], x[edge[1]]], [y[edge[0]], y[edge[1]]], color='#000000'))
509   ax.plot(x, y, 'ro')
510   plt.show()
511 
```

```python
# CODE PRINCIPAL #

ville = Road_graph() # Initialisation du graphe routier

if bs: # Vérifie si on utilise l'algorithme naïf ou BallString et nomme le fichier de données en conséquent
  out_file = "data/Ville_" + str(car_number) + "_bs.txt"
else:
  out_file = "data/Ville_" + str(car_number) + "_astar.txt"

sys.stdout = open(out_file,'wt') # Indique au compilateur d'écrire dans ledit fichier

with open('city.geojson', 'r') as ville_file: # Lit les données JSON du fichier en question et les importe
  ville_data = json.load(ville_file)
  for i in range(len(ville_data["features"])):
    coordinates = ville_data["features"][i]["geometry"]["coordinates"]
    coordinates = [[np.round(float(i), rnd) for i in nested] for nested in coordinates]
    data = ville_data["features"][i]["properties"]
    for j in range(0, len(coordinates)):
      ville.add_node(coordinates[j][1], coordinates[j][0])
    ville.add_edge(coordinates, data)

  ville.single_connex() # Rend le graphe connexe pour éviter les erreurs de sommets inaccessibles

  size = len(ville.nodes)

  cars = [] # Crée le tableau de voitures
  for i in range(car_number):
    cars.append(Car(ville, random.randint(0, size-1), random.randint(0, size-1)))

  start = time.time() # Démarre un chronomètre pour mesurer le temps d'exécution

  while not ended_cars(cars): # Fait « vivre » les voitures
    tps += tick
    for i in range(len(cars)):
      if bs:
        cars[i].update_ballstring(ville)
      else:
        cars[i].update_naive(ville)

  end = time.time() # Arrête le chronomètre

  trip_times = []
  opti_times = []
  stucks = 0
  for i in range(len(cars)): # Enregistrement des temps optimaux et réels de trajet des voitures
    trip_times.append(cars[i].trip_time)
    opti_times.append(cars[i].optimal_time)
    stucks += cars[i].times_stuck
  print(end-start) # Impression des métriques intéressantes dans le fichier de sortie
  print(mean(trip_times))
  print(mean(opti_times))
  print(stucks)

# FIN DU CODE #
```