Congestion en milieu urbain

Fluidifier le trafic grâce au routage dynamique

Martin Pupat - 26609 - MPI

Embouteillages, Grenoble

Comment router efficacement de nombreux véhicules en minimisant les calculs et la congestion ?

- Extraction de données et implémentation des algorithmes
- Modélisation mathématique des paramètres et étude de complexité temporelle
- Validation expérimentale des résultats théoriques

Mise en forme du problème

Données :

- Un graphe routier (connexe, quasi-planaire)
  - n sommets
  - m arêtes ($n - 1 \leqslant m \leqslant 3n - 6$ donc $m = \mathcal{O}(n)$)
- V voitures
  - d'un point de départ à un point d'arrivée (aléatoires)
  - en heure de pointe (instant de départ suivant une loi de Gauss)

Points de recherche :

- Conduire les voitures à leur destination le plus vite possible
- Effectuer le moins de calculs possible
- Créer le moins de bouchons possible

Algorithmes étudiés : A^* itéré et BallString simplifié

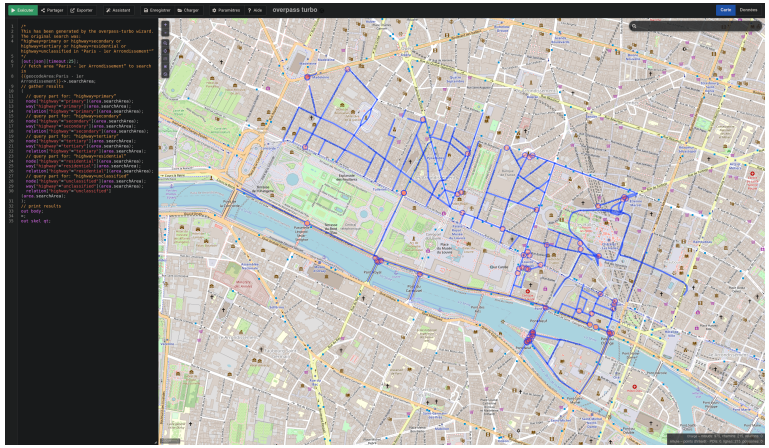
Représentation du graphe routier

Données :

- Jeux de données : API de `openstreetmap.org` → Fichiers `.geojson`
- Graphe en Python : `nodes`, `edges`, `cars`

Observations (en moyenne) :

- Coût de contournement : $e \approx 2,72\ldots$
- Capacité d'une arête : $q \gg 1$
- Capacité du graphe : $Q = qm$ donc $V < Q$ et $V = \mathcal{O}(n)$
- Distance entre deux points quelconques : $L = \mathcal{O}(\sqrt{n})$ en pratique

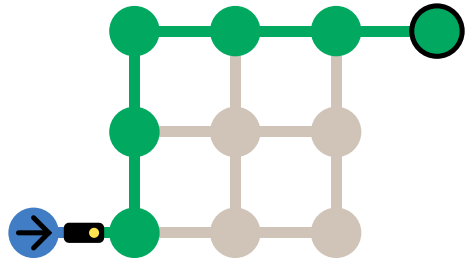


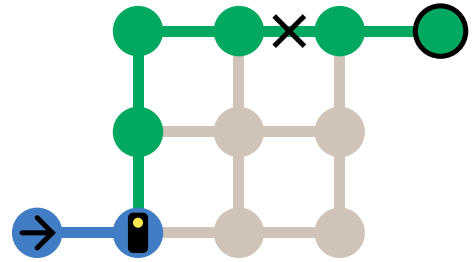
Graphe routier de Paris I
(Site : overpass-turbo.eu)

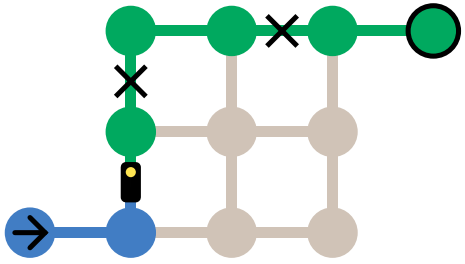
Algorithme A^* itéré (naïf)

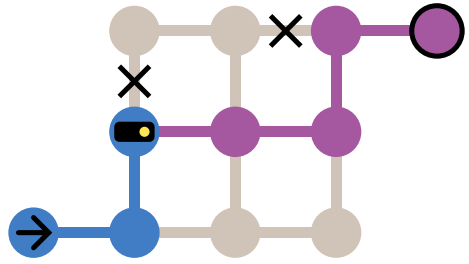
Idée :

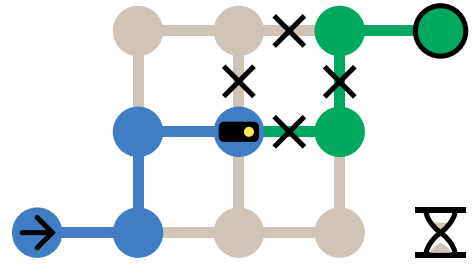
- Suivre un parcours A^*
- Recalculer *tout le trajet prévu* si bloquage *devant soi* sur le chemin







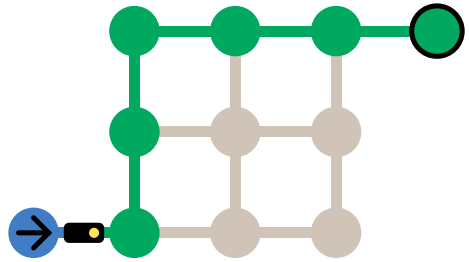


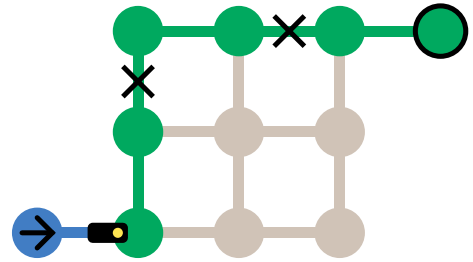


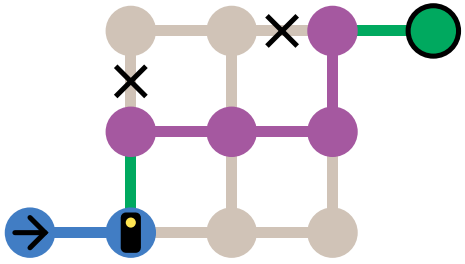
Algorithme BallString simplifié

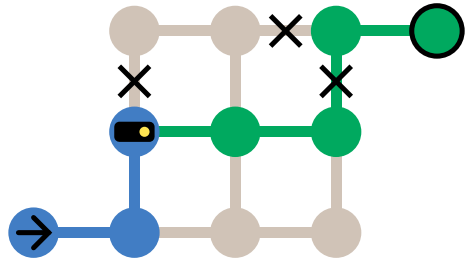
Idée :

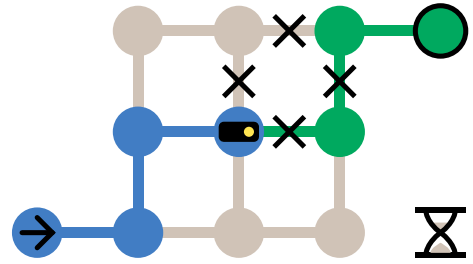
- Suivre un parcours A^*
- Recalculer *entre les sommets saturés* si bloquage *plus loin* sur le chemin











Étude de complexité - algorithme A^* itéré

Probabilité de saturation d'une arête (équirépartition) : $p = \frac{V}{Q}$

Nombre de chemins sortants par intersection : x

Indice de la première route sur laquelle la voiture bloque : $\Sigma \rightsquigarrow \mathcal{G}(p)$
$\rightarrow \mathbb{E}(\Sigma) = 1/p$.

Nombre de recalculs à faire à chaque bloquage : $B \rightsquigarrow \mathcal{G}((1-p)^x)$
$\rightarrow \mathbb{E}(B) = 1/(1-p)^x$.

Distance gagnée en moyenne à chaque recalcul : $\delta = e - \mathbb{E}(\Sigma)$

Complexité de l'algorithme A^* itéré

Pour $p < 1/e$ fixé,

$$\begin{aligned}\widehat{C} &\underset{n\to+\infty}{\sim} V \times A \times \frac{L}{\delta} \times \mathbb{E}(B)\\ &\underset{n\to+\infty}{\sim} \frac{VAL}{(1-pe)(1-p^x)}\\ &\underset{n\to+\infty}{=} \mathcal{O}(n^2\sqrt{n}\log n)\end{aligned}$$

Pour n fixé assez grand,

$$\widehat{C}_p \underset{p\to(1/e)_-}{=} \mathcal{O}(1/(e-p))$$

Étude de complexité - algorithme BallString

En notant $\sigma_i(j)$ « la $i$ème arête est saturée quand la voiture est à la $j$ème intersection »,

Probabilité de saturation d'une arête en aval à la $i$ème intersection :

$$\mathbb{P}\left(\bigcup_{k=i}^{n}\sigma_i(j)\right) = 1-(1-p)^{n-i}$$

Probabilité qu'aucune arête n'ait déjà été saturée en aval :

$$\mathbb{P}\left(\bigcap_{k=0}^{i-1}\bigcap_{l=k}^{n}\sigma_k(l)\right) = \prod_{k=0}^{i-1}(1-p)^{n-k+1} = (1-p)^{-\frac{i^2}{2}+i(n+\frac{3}{2})}$$

Donc

$$\mathbb{P}(\Sigma) = \left[1-(1-p)^{n-i}\right]\left[(1-p)^{-\frac{i^2}{2}+i(n+\frac{3}{2})}\right]$$

$\rightarrow$ trop difficile à analyser.

Complexité de l'algorithme BallString

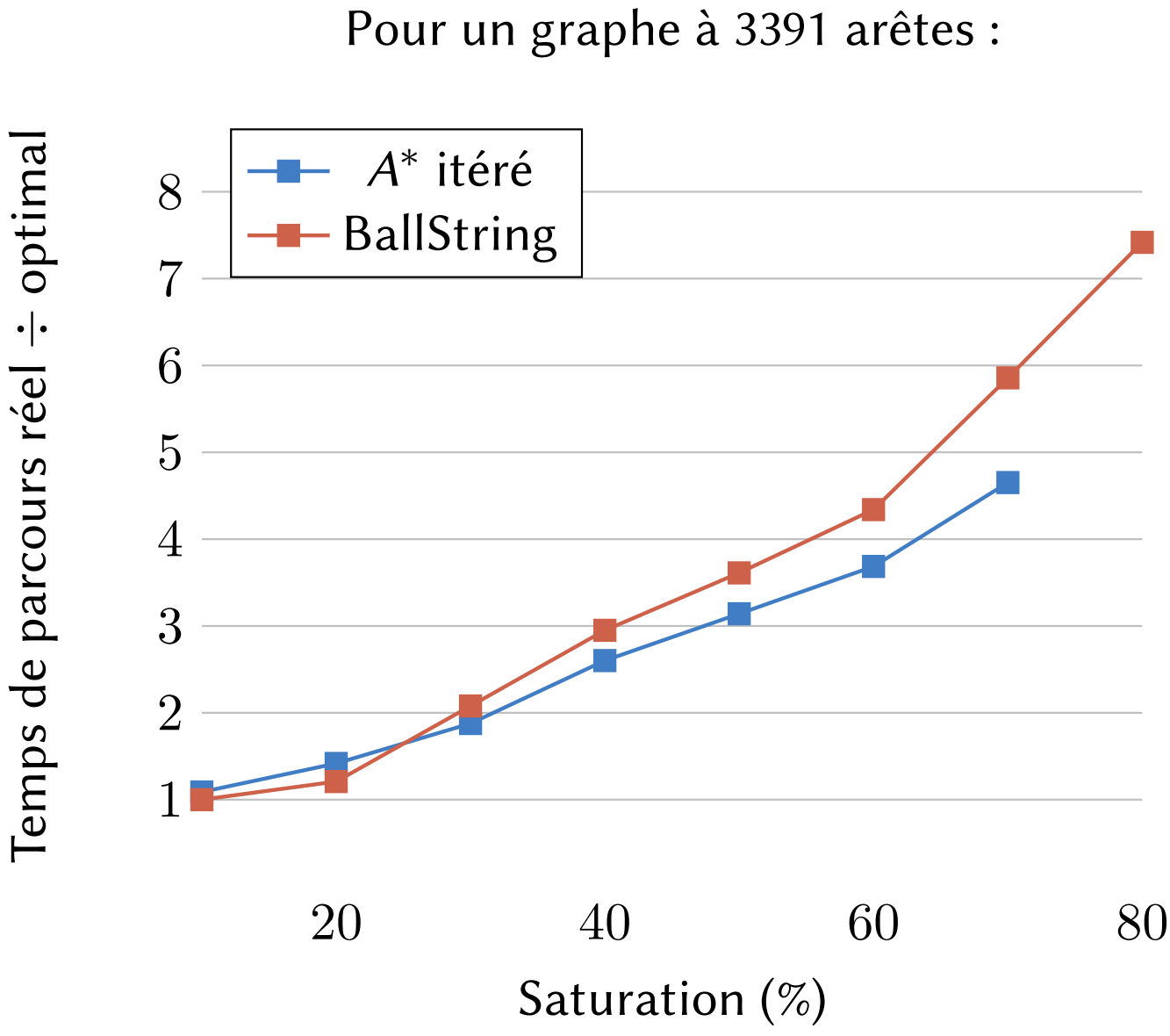
D'après [5], en pratique, pour une voiture :

$$\hat{c} \underset{n \to +\infty}{=} \mathcal{O}(n \log n)$$

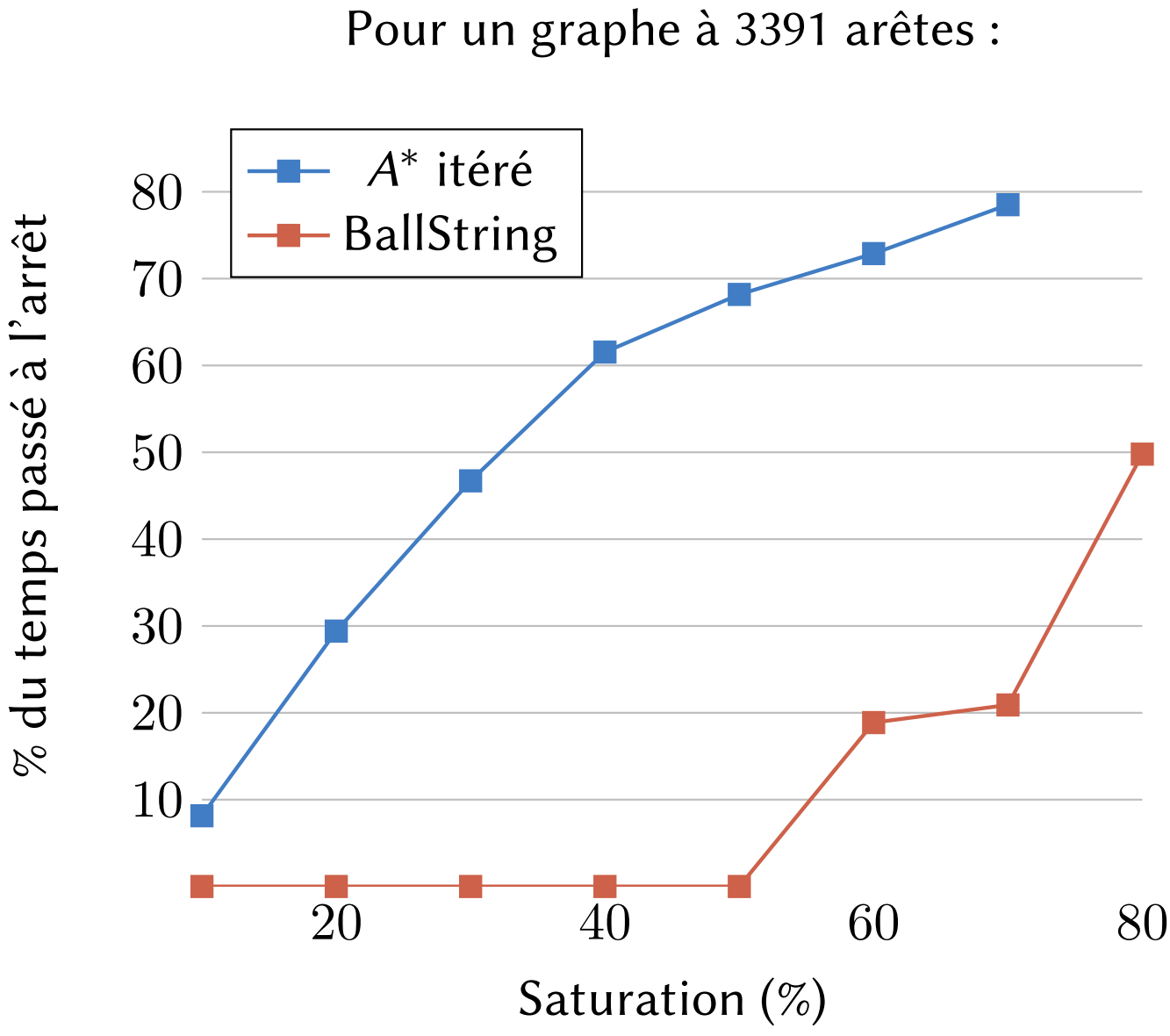
donc pour $V \in \mathcal{O}(n)$ voitures :

$$\hat{C} \underset{n \to +\infty}{=} \mathcal{O}(n^2 \log n)$$

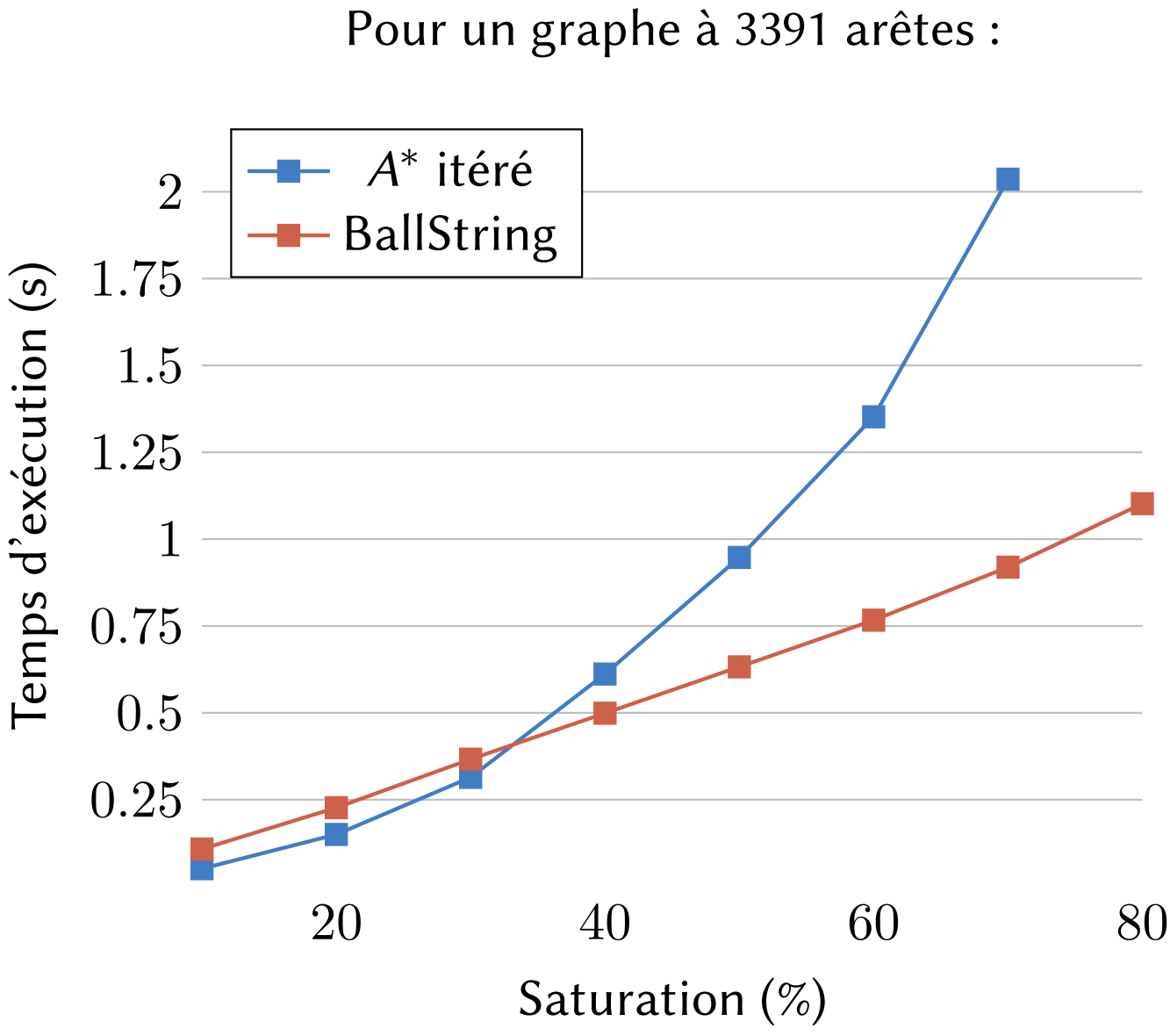
A^* itéré donne des parcours plus courts



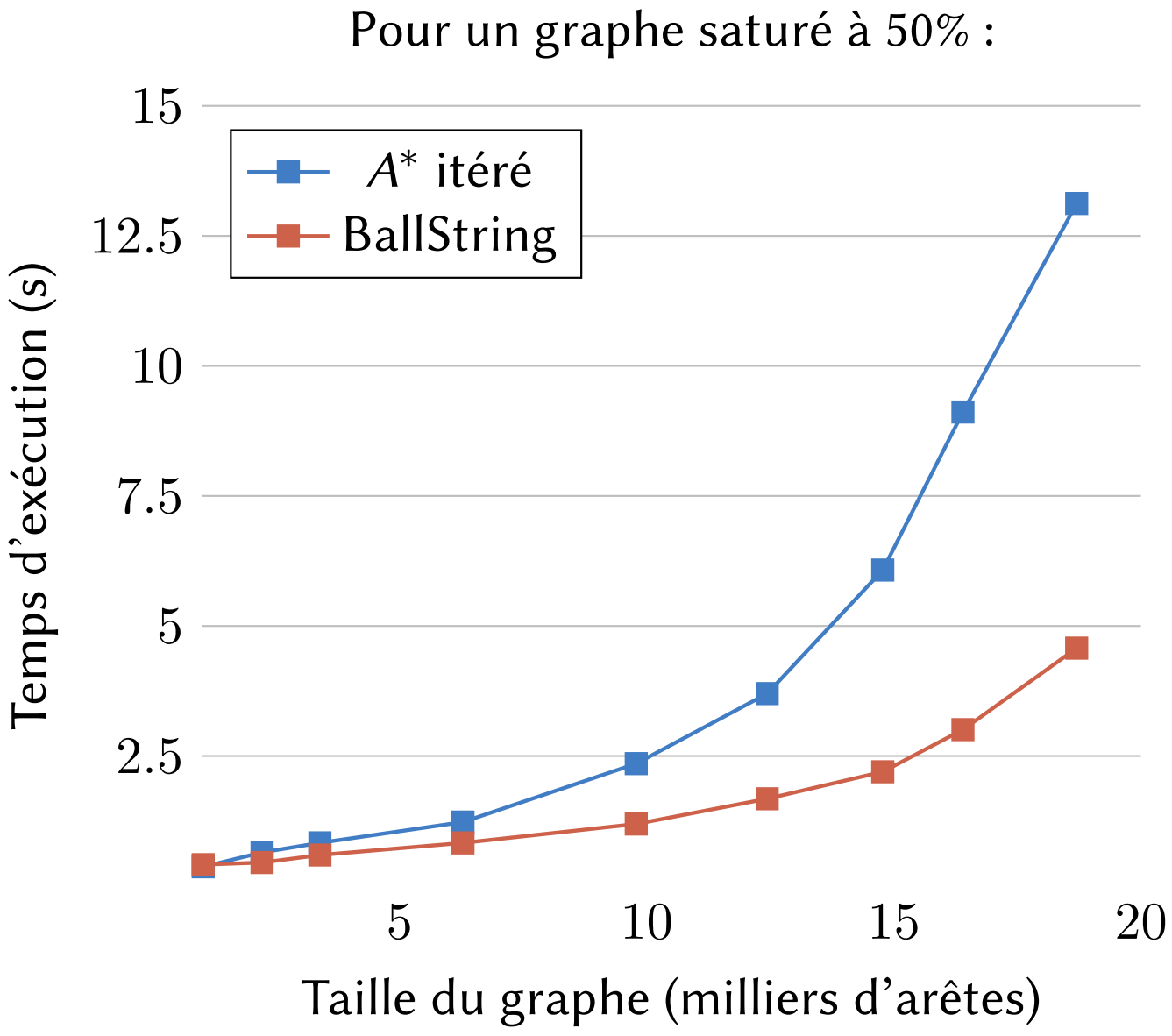
Ballstring crée moins de bouchons



BallString calcule plus vite sur des routes encombrées



BallString calcule plus vite sur des grands graphes



Comparaison des deux algorithmes

Algorithme	A^* itéré	BallString simplifié
Complexité	$\mathcal{O}(n^2\sqrt{n}\log n)$	$\mathcal{O}(n^2\log n)$
Saturation	Temps moyen de parcours réel ÷ à vide	
20%	1.2	1.4
50%	3.1	3.6
80%	$*$	7.4
Arêtes	Temps d'exécution à 50% de saturation (s)	
1000	0.10	0.14
4000	0.95	0.63
20000	13.12	4.57

Conclusion

L'algorithme A^* itéré est plus efficace :

- Pour les petits graphes
- Pour les réseaux peu saturés

L'algorithme BallString simplifié est plus efficace :

- Pour les grands graphes
- Pour les réseaux très saturés
- Pour garder les rues en mouvement

Bibliographie

[1] Department of Engineering Science. *Fast Shortest Path Algorithms for Large Road Networks*. URL : orsnz.org.nz/conf36/papers/Engineer.pdf.

[2] Dimitris J. Bertsimas et Garrett Van Ryzin. *Stochastic and Dynamic Vehicle Routing with General Demand and Interarrival Time Distributions*. DOI : doi.org/10.2307/1427801.

[3] David Larsen. *The Dynamic Vehicle Routing Problem*. URL : portal.issn.org/resource/ISSN/0909-3192.

[4] Edward P.F. Chan et Yaya Yang. *Shortest Path Tree Computation in Dynamic Graphs*. DOI : doi.org/10.1109/TC.2008.198.

[5] Giacomo Nannicini et Leo Liberti. *Shortest Paths on Dynamic Graphs*. DOI : doi.org/10.1111/j.1475-3995.2008.00649.x.

[6] Melanie Smith et Roger Mailler. *The Effect of Congestion Frequency and Saturation on Coordinated Trac Routing*. DOI : doi.org/10.1007/978-3-642-25044-6.

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

```

```
6  # IMPORTATION DES BIBLIOTHÈQUES #
7
8  # Lecture des données du fichier .geojson #
9
10 import json
11 import fiona
12 import geopandas
13 import matplotlib.pyplot as plt
14
15 # Calculs mathématiques #
16
17 from geographiclib.geodesic import Geodesic
18 import math
19 import numpy as np
20 import random
21 import time
22 from statistics import mean
23
```

```
24 # Lecture / écriture de fichiers externes #
25
26 import sys
27
28 # Représentation visuelle du graphe routier #
29
30 from pylab import Line2D, gca
31
```

```
32  # VARIABLES GLOBALES #
33
34  rnd = 6 # Arrondissement des coordoonnées des noeuds
35
36  tick = float(1) # Valeur d'une « seconde »
37
38  car_number = int(sys.argv[1]) # Nombre de voitures
39
40  bs = False # Utilisation de l'algotihme BallString
41  if sys.argv[2] == "True":
42    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
```

```
92      elif len(self.ongoing_path) == 1: # La voiture est
            sur la fin de sa dernière arête
93          self.trip_time += self.edge_duration
94          tplus = tick - self.edge_duration
95          self.edge_duration = 0
96          self.ongoing_path = []
97          city.nodes[self.elapsed_path[-1]].edges_out[self.
                current_edge].remove_car()
98          self.elapsed_path.append(self.end)
99          self.current_edge = None
100         self.done = True
```

```
101         else: # La voiture fait un changement d'arête
102             nxt = city.edge_to(self.ongoing_path[0], self.
                    ongoing_path[1])
103             blocked = self.stuck_ahead(city)
104             if blocked == []: # Aucun problème devant
105                 self.trip_time += tick
106                 tplus = tick - self.edge_duration
107                 city.nodes[self.elapsed_path[-1]].edges_out[self
                        .current_edge].remove_car()
108                 self.current_edge = nxt
109                 self.elapsed_path.append(self.ongoing_path.pop
                        (0))
110                 city.nodes[self.elapsed_path[-1]].edges_out[self
                        .current_edge].add_car()
111                 self.edge_duration = city.nodes[self.
                        elapsed_path[-1]].edges_out[self.current_edge
                        ].time - tplus
```

```
112         else: # Un problème devant
113           tmp = self.ongoing_path
114           try:
115             greffon = astar_queue(city, self.ongoing_path[
                    blocked[0]], self.ongoing_path[blocked
                    [-1]], True)
116           except: # Impossible de trouver un autre chemin
117             self.times_stuck += 1
118             self.ongoing_path = tmp
119             nxt = city.edge_to(self.ongoing_path[0], self.
                    ongoing_path[1])
```

```
120             if city.nodes[self.ongoing_path[0]].edges_out[
                    nxt].available():
121                 self.trip_time += tick
122                 tplus = tick - self.edge_duration
123                 city.nodes[self.elapsed_path[-1]].edges_out[
                        self.current_edge].remove_car()
124                 self.current_edge = nxt
125                 self.elapsed_path.append(self.ongoing_path.
                        pop(0))
126                 city.nodes[self.elapsed_path[-1]].edges_out[
                        self.current_edge].add_car()
127                 self.edge_duration = city.nodes[self.
                        elapsed_path[-1]].edges_out[self.
                        current_edge].time - tplus
128             else:
129                 self.trip_time += tick
```

```
130         else: # Possible de trouver un autre chemin
131             self.ongoing_path = greffe(tmp, greffon, city
                    )
132             self.trip_time += tick
133             tplus = tick - self.edge_duration
134             city.nodes[self.elapsed_path[-1]].edges_out[
                    self.current_edge].remove_car()
135             self.current_edge = nxt
136             self.elapsed_path.append(self.ongoing_path.
                    pop(0))
137             city.nodes[self.elapsed_path[-1]].edges_out[
                    self.current_edge].add_car()
138             self.edge_duration = city.nodes[self.
                    elapsed_path[-1]].edges_out[self.
                    current_edge].time - tplus
139
```

```
140     def update_naive(self, city): # Mise à jour de la
            voiture selon l'algorithme A* itéré
141       if self.start_time < tps:
142         pass
143       if self.done:
144         pass
145       elif self.edge_duration > tick:
146         self.trip_time += tick
147         self.edge_duration -= tick
148       elif len(self.ongoing_path) == 1:
149         self.trip_time += self.edge_duration
150         self.edge_duration = 0
151         self.ongoing_path = []
152         city.nodes[self.elapsed_path[-1]].edges_out[self.
                current_edge].remove_car()
153         self.elapsed_path.append(self.end)
154         self.current_edge = None
155         self.done = True
```

```
156         else:
157             nxt = city.edge_to(self.ongoing_path[0], self.
                    ongoing_path[1])
158             if city.nodes[self.ongoing_path[0]].edges_out[nxt
                    ].available():
159                 self.trip_time += tick
160                 tplus = tick - self.edge_duration
161                 city.nodes[self.elapsed_path[-1]].edges_out[self
                        .current_edge].remove_car()
162                 self.current_edge = nxt
163                 self.elapsed_path.append(self.ongoing_path.pop
                        (0))
164                 city.nodes[self.elapsed_path[-1]].edges_out[self
                        .current_edge].add_car()
165                 self.edge_duration = city.nodes[self.
                        elapsed_path[-1]].edges_out[self.current_edge
                        ].time - tplus
```

```
166         else:
167           tmp = self.ongoing_path
168           try:
169             self.ongoing_path = astar_queue(city, self.
                    ongoing_path[0], end, True)
170           except:
171             self.times_stuck += 1
172             self.trip_time += tick
173             self.ongoing_path = tmp
```

```
174             else:
175                 self.trip_time += tick
176                 tplus = tick - self.edge_duration
177                 city.nodes[self.elapsed_path[-1]].edges_out[
                        self.current_edge].remove_car()
178                 self.current_edge = nxt
179                 self.elapsed_path.append(self.ongoing_path.pop
                        (0))
180                 city.nodes[self.elapsed_path[-1]].edges_out[
                        self.current_edge].add_car()
181                 self.edge_duration = city.nodes[self.
                        elapsed_path[-1]].edges_out[self.
                        current_edge].time - tplus
182
```

```
183 # Edge : arête reliant deux sommets #
184
185 class Edge:
186  # Pas besoin d'enregistrer le noeud de départ puisqu'
       une arête est dans un tableau depuis le noeud de
       départ
187   def __init__(self, start_node, end_node, speed, lanes
        ): # Initialisation de l'arête
188     self.end_node:int = end_node # Noeud d'arrivée
189     self.speed = speed # Vitesse max
190     self.lanes = lanes # Nombre de voies
191     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
192     self.time = float(self.length)/float(self.speed) *
          3.6 # Temps de parcours de la rue
```

```
193     self.capacity = math.ceil((float(self.length)/float
            (self.speed)) * 2 * lanes) # Capacité de la rue
            (en tenant compte des distances de sécurité)
194     self.occupied = 0 # Nombre de voiitures
            actuellement sur l'arête
195     self.min_available = self.capacity-self.occupied #
            Place libre minimum dont a disposé l'arête (
            déboguage)
196
```

```
197     def available(self): # Renvoie vrai si et seulement
            si l'arête n'est pas saturée
198       return (self.capacity - self.occupied > 0)
199
200     def add_car(self): # Applique l'ajout d'une voiture
            sur l'arête
201       self.occupied += 1
202       if self.min_available > self.capacity-self.occupied
              :
203         self.min_available = self.capacity-self.occupied
204
205     def remove_car(self): # Applique le retrait d'une
            voiture sur l'arête
206       self.occupied -= 1
207
```

```
208 # Node : sommet du graphe #
209
210 class Node:
211   def __init__(self, lat, lon): # Initialisation du
        sommet
212     self.lat:int = lat # Latitude
213     self.lon:int = lon # Longitude
214     self.edges_out = [] # Tableau d'arêtes sortantes
215     self.edges_in = [] # Tableau d'arêtes entrantes
216
```

```
217 # Road_graph : le graphe routier en question #
218
219 class Road_graph:
220   def __init__(self): # Initialisation de la ville
221     self.nodes = [] # Tableau de sommets
222
223   def edge_to(self, n, end): # Renvoie le numéro de l'
        arête depuis n qui relie end
224     for i in range(len(self.nodes[n].edges_out)):
225       if self.nodes[n].edges_out[i].end_node == end:
226         return i
227     raise Exception("Invalid edge !")
228
```

```
229     def neighbors(self, n): # Renvoie la liste des
            sommets adjacents à n
230       res = []
231       for i in range(self.outgoing_edges(n)):
232         res.append([self.nodes[n].edges_out[i].end_node,
              self.nodes[n].edges_out[i].length])
233       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
```

```
382     def insert(self, key, val, father, city): # Insère un
            élément dans la file
383       if(self.get_index(key) > val):
384         self.set_index(key, val)
385         self.set_father(key, father)
386         if key in self.heap:
387           self.heap.remove(e)
388         i = len(self.heap)
389         self.heap.append(key)
390         while (i != 0):
391           p = self.parent(i)
392           if self.is_before(p, i):
393             self.swap(p, i, city)
394           i = p
395
```

```
396     def extract(self): # Renvoie l'élément en tête de
            file
397       if len(self.heap) == 0:
398         return None
399       r = self.heap[0]
400       self.heap[0] = self.heap[-1]
401       del self.heap[-1]
402       self.descent(0)
403       return r
404
405     def ancestry(self, key): # Renvoie le chemin pour
            aller jusqu'à l'élément
406       r = [key]
407       while (self.get_father(key) != key):
408         r.append(self.get_father(key))
409         key = self.get_father(key)
410         return r.reverse()
411
```

```
412 # ALGORITHME DE PARCOURS #
413
414 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.
415   queue = proximity_queue(len(ville.nodes))
416   queue.insert(start, 0, start)
417   while (not(queue.is_empty())):
418     current = queue.extract()
419     so_far = queue.get_index(current)
420     for i in range(len(city.nodes[current].edges_out)):
421       if (city.nodes[current].edges_out[i].available) or
              (not av):
422         d = queue.get_index(current) + city.nodes[
              current].edges_out[i].length
423         e = city.nodes[current].edges_out[i].end_node
424         queue.insert(e, d, current, city)
```

```
425     if queue.get_index(end) == math.inf:
426       raise Exception("No path found !")
427     res = ancestry(end)
428     return (res, math.inf)
429
```

```
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
```

```
512 # CODE PRINCIPAL #
513
514 ville = Road_graph() # Initialisation du graphe
        routier
515
516 if bs: # Vérifie si on utilise l'algorithme naïf ou
        BallString et nomme le fichier de données en
        conséquent
517   out_file = "data/Ville_" + str(car_number) + "_bs.txt
          "
518 else:
519   out_file = "data/Ville_" + str(car_number) + "_astar.
          txt"
520
521 sys.stdout = open(out_file,'wt') # Indique au
        compilateur d'écrire dans ledit fichier
522
```

```
523 with open('city.geojson', 'r') as ville_file: # Lit
      les données JSON du fichier en question et les
      importe
524   ville_data = json.load(ville_file)
525   for i in range(len(ville_data["features"])):
526     coordinates = ville_data["features"][i]["geometry"
          ]["coordinates"]
527     coordinates = [[np.round(float(i), rnd) for i in
          nested] for nested in coordinates]
528     data = ville_data["features"][i]["properties"]
529     for j in range(0, len(coordinates)):
530       ville.add_node(coordinates[j][1], coordinates[j
            ][0])
531     ville.add_edge(coordinates, data)
532
533   ville.single_connex() # Rend le graphe connexe pour
      éviter les erreurs de sommets inaccessibles
534
```

```
535     size = len(ville.nodes)
536
537     cars = [] # Crée le tableau de voitures
538     for i in range(car_number):
539       cars.append(Car(ville, random.randint(0, size-1),
              random.randint(0, size-1)))
540
541     start = time.time() # Démarre un chronomètre pour
            mesurer le temps d'exécution
542
543     while not ended_cars(cars): # Fait « vivre » les
            voitures
544       tps += tick
545       for i in range(len(cars)):
546         if bs:
547           cars[i].update_ballstring(ville)
548         else:
549           cars[i].update_naive(ville)
```

```
550
551     end = time.time() # Arrête le chronomètre
552
553     trip_times = []
554     opti_times = []
555     stucks = 0
556     for i in range(len(cars)): # Enregistrement des temps
            optimaux et réels de trajet des voitures
557       trip_times.append(cars[i].trip_time)
558       opti_times.append(cars[i].optimal_time)
559       stucks += cars[i].times_stuck
560     print(end-start) # Impression des métriques
            intéressantes dans le fichier de sortie
561     print(mean(trip_times))
562     print(mean(opti_times))
563     print(stucks)
564
565   # FIN DU CODE #
```