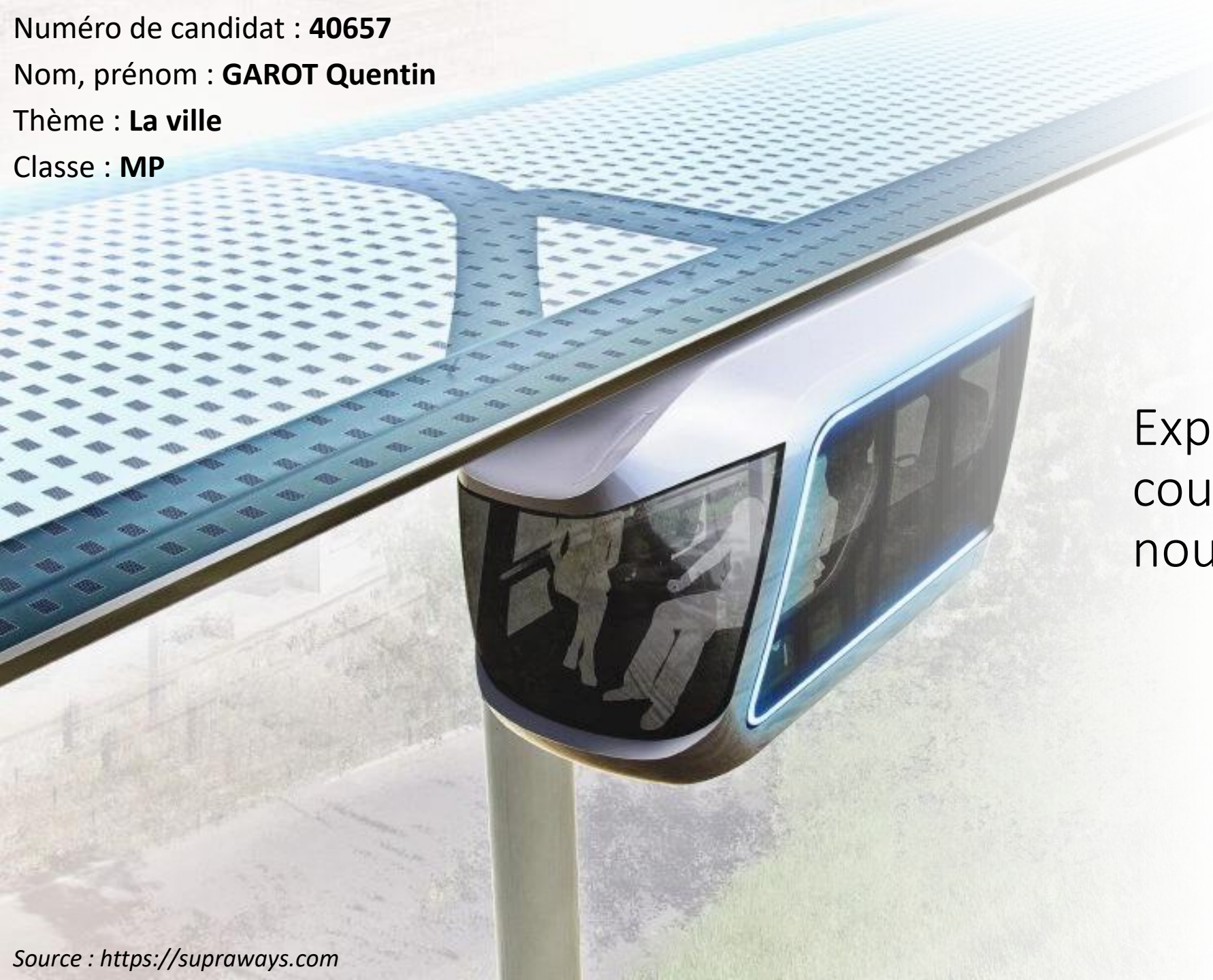


Numéro de candidat : **40657**

Nom, prénom : **GAROT Quentin**

Thème : **La ville**

Classe : **MP**



Exploitation optimale des
couloirs aériens pour une
nouvelle mobilité urbaine

Problématique retenue

Comment peut-on implémenter des algorithmes permettant d'une part de représenter le fonctionnement d'un nouveau moyen de transport urbain et d'autre part de s'approcher d'une fluidité optimale du trafic ?

Sommaire

1. Principe du nouveau moyen de transport *Supraways* et implémentation informatique possible
2. Etude de l'algorithme A* pour optimiser les trajets
3. Répartition des cabines en fonction des demandes
4. Annexe (code et démonstrations)

I. Principe et implémentation possible

I.1. Généralités

- Installation de stations
- Rails aériens
- Représentation : graphe
- Programme : choix de l'emplacement des stations



I.1. Généralités

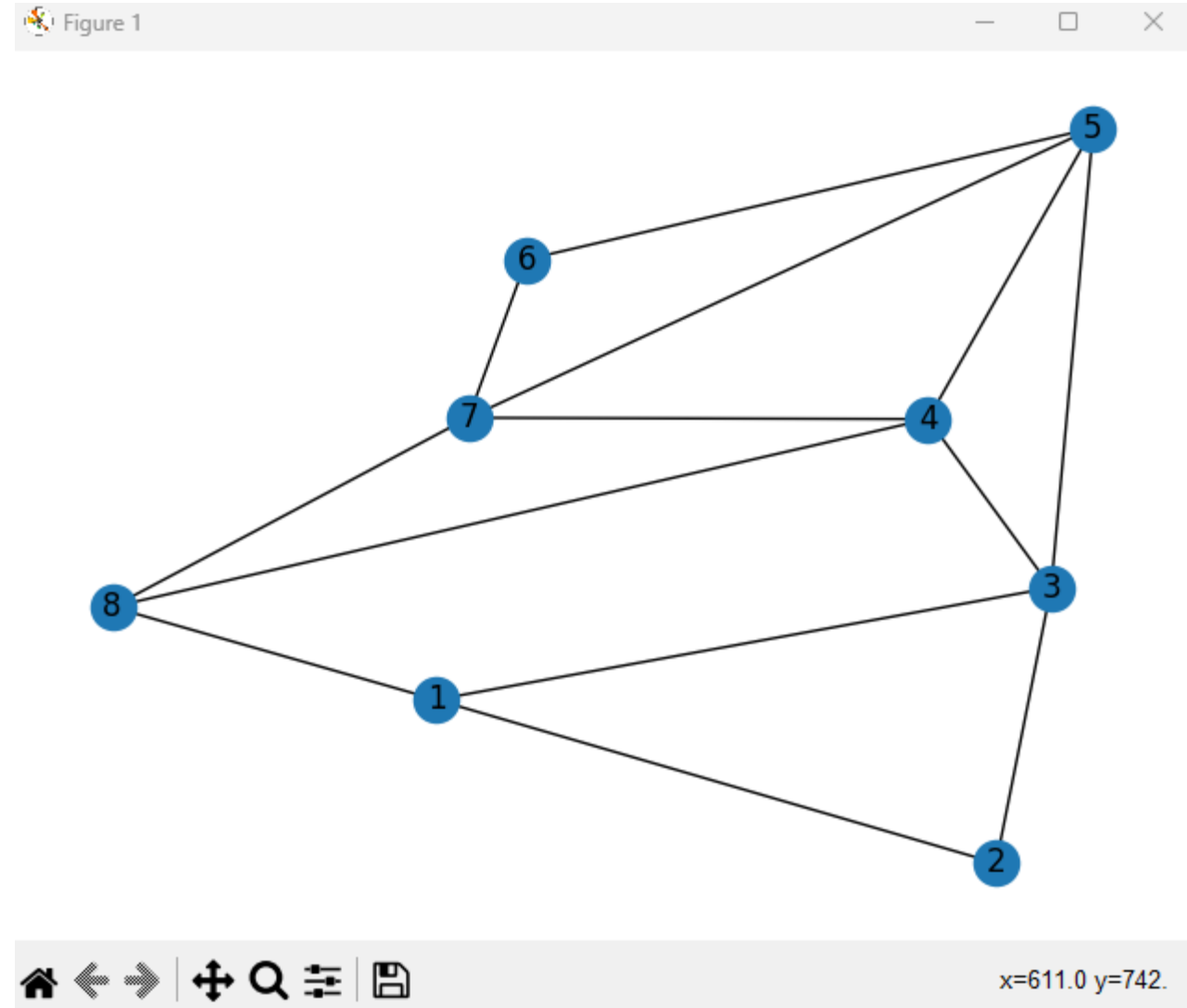
Représentation du graphe

Bibliothèque Python utilisée : *NetworkX*



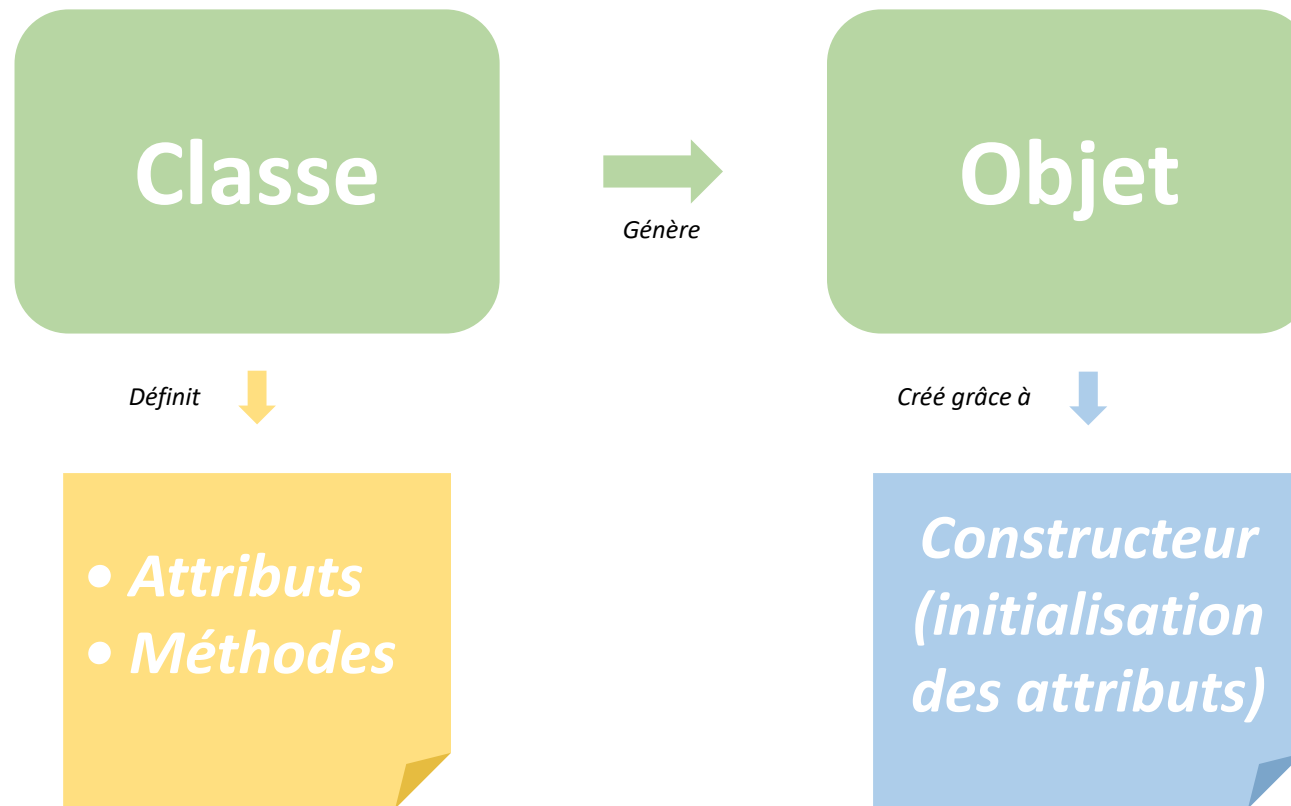
Source : <https://networkx.org>

Comment suis-je parvenu à réaliser ceci ?



I.2. Programmation orientée objet (POO)

I.2.a. De quoi s'agit-il ?



I.2. Programmation orientée objet (POO)

I.2.a. De quoi s'agit-il ?

I.2.b. Quelques exemples simples

```
class Point:
    def __init__(self, x: float, y: float):
        self.x = x
        self.y = y

    def get_x(self) -> float:
        return self.x

    def get_y(self) -> float:
        return self.y
```

```
class Node(Point):
    def __init__(self, id: int, x: int, y: int):
        super().__init__(x, y)
        # Identifiant
        self.id = id

    def get_id(self) -> int:
        return self.id
```

I.2. Programmation orientée objet (POO)

I.2.a. De quoi s'agit-il ?

I.2.b. Quelques exemples simples

I.2.c. POO et base de données (BDD)

Structure de la table *stations*

```
CREATE TABLE stations (  
    id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,  
    name VARCHAR(255),  
    localisation_x FLOAT,  
    localisation_y FLOAT,  
    current_gondola INT NOT NULL DEFAULT 0  
);
```

Structure de la classe *Database*



Database

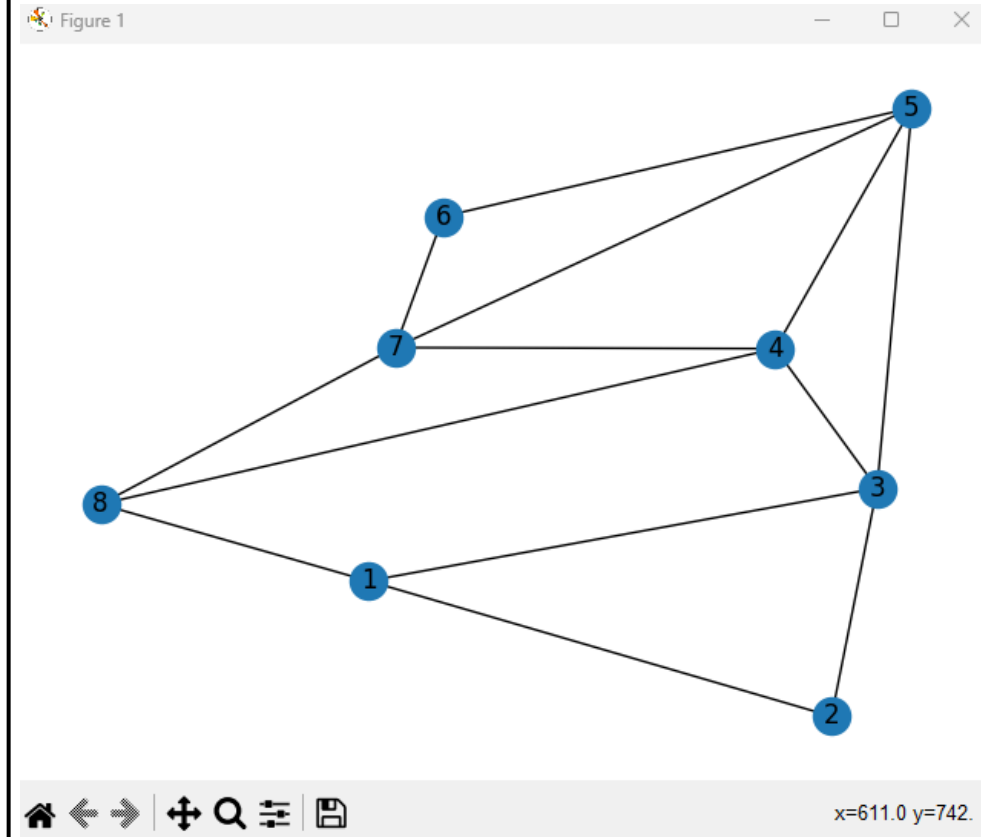
```
set(self, sql: str) -> None:  
get(self, sql: str) -> list[tuple]:
```


I.3. Réalisation de la carte interactive

Démarche

1. Recherche du plan de la ville de Belfort
2. Documentation de la bibliothèque *Matplotlib*
3. Création d'une classe *Map*, comportant 3 fonctionnalités principales :
 - Affichage du plan (interactif)
 - Positionnement des stations sur le plan & insertion dans la BDD
 - Affichage du graphe

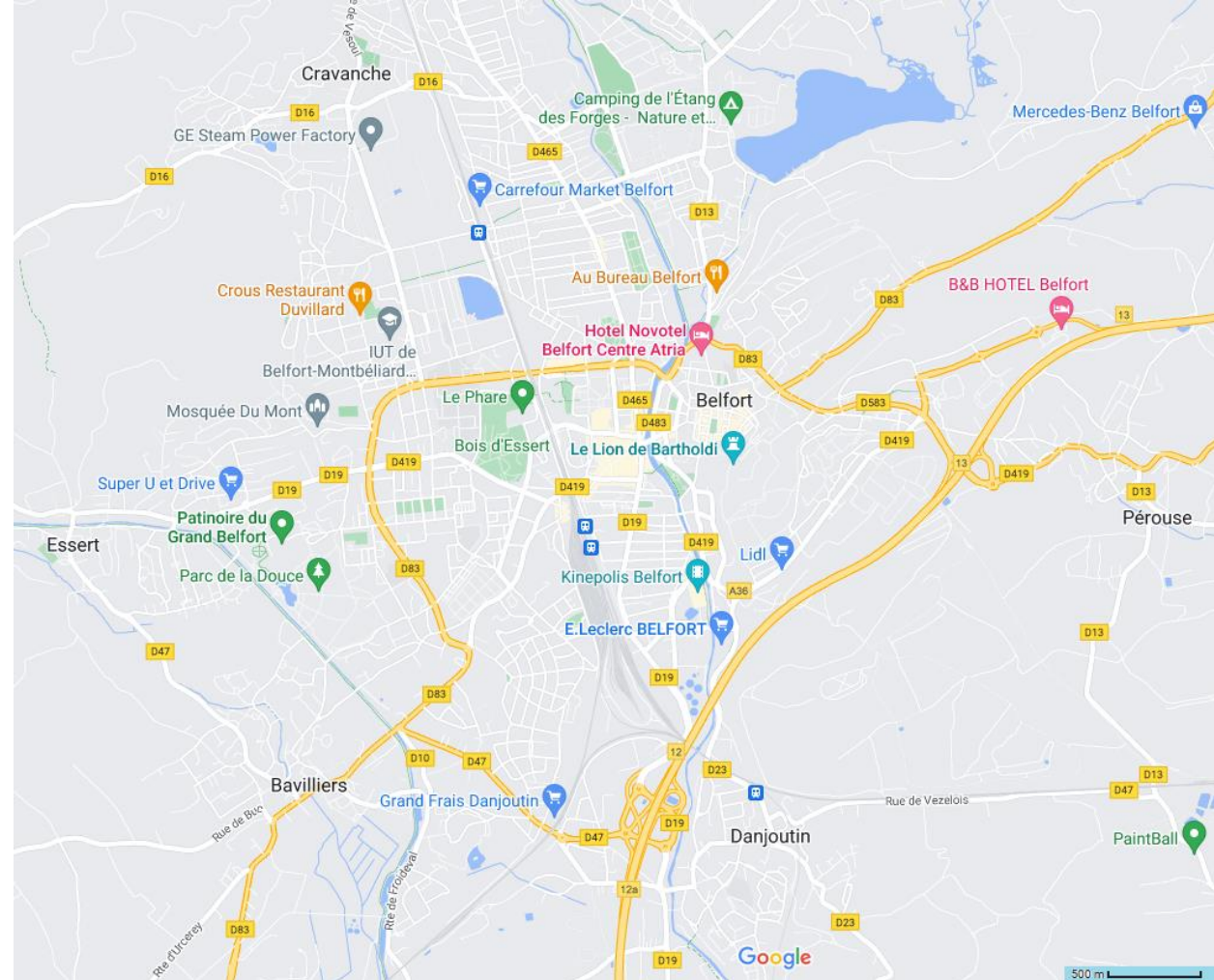
Intérêt : Etude de l'optimalité du trafic



II. Algorithme important : algorithme A*

II.1. Mise en situation – Personal Rapid Transit (PRT)

- Définition PRT
- Principe



II.2. Algorithme A*

II.2.a. Définitions – notations

$$G = (S, A, w)$$

Définition : heuristique pour la recherche d'un sommet t

$$h : S \rightarrow \mathbb{R}_+ \mid h(t) = 0$$

Notation : poids d'un plus court chemin entre deux sommets a et b

$$d(a, b) = \min(\{w(p) \text{ avec } p \text{ un chemin de } a \text{ à } b\})$$

Définition : heuristique admissible pour la recherche d'un sommet t

$$h \text{ est admissible lorsque } \forall s \in S, h(s) \leq d(s, t)$$

II.2.b. Propriétés des nœuds

- **g** : coût de déplacement
- **h** : heuristique
- **f** : $g + h$ (priorité d'un nœud)
- **Nœud parent**

Getters

```
get_cost(self) -> int | None:  
get_heuristic(self) -> float:  
get_f(self) -> float:  
get_parent_node(self) -> Self:
```

Setters

```
set_cost(self, cost: float | None) -> None:  
set_heuristic(self, heuristic: float | None) -> None:  
set_parent_node(self, node: Self | None) -> None:
```

II.2. Algorithme A*

II.2.a. Définitions – notations

II.2.b. Propriétés des nœuds

II.2.c. File de priorité

Interface

`add(self, x: Node) -> None:`

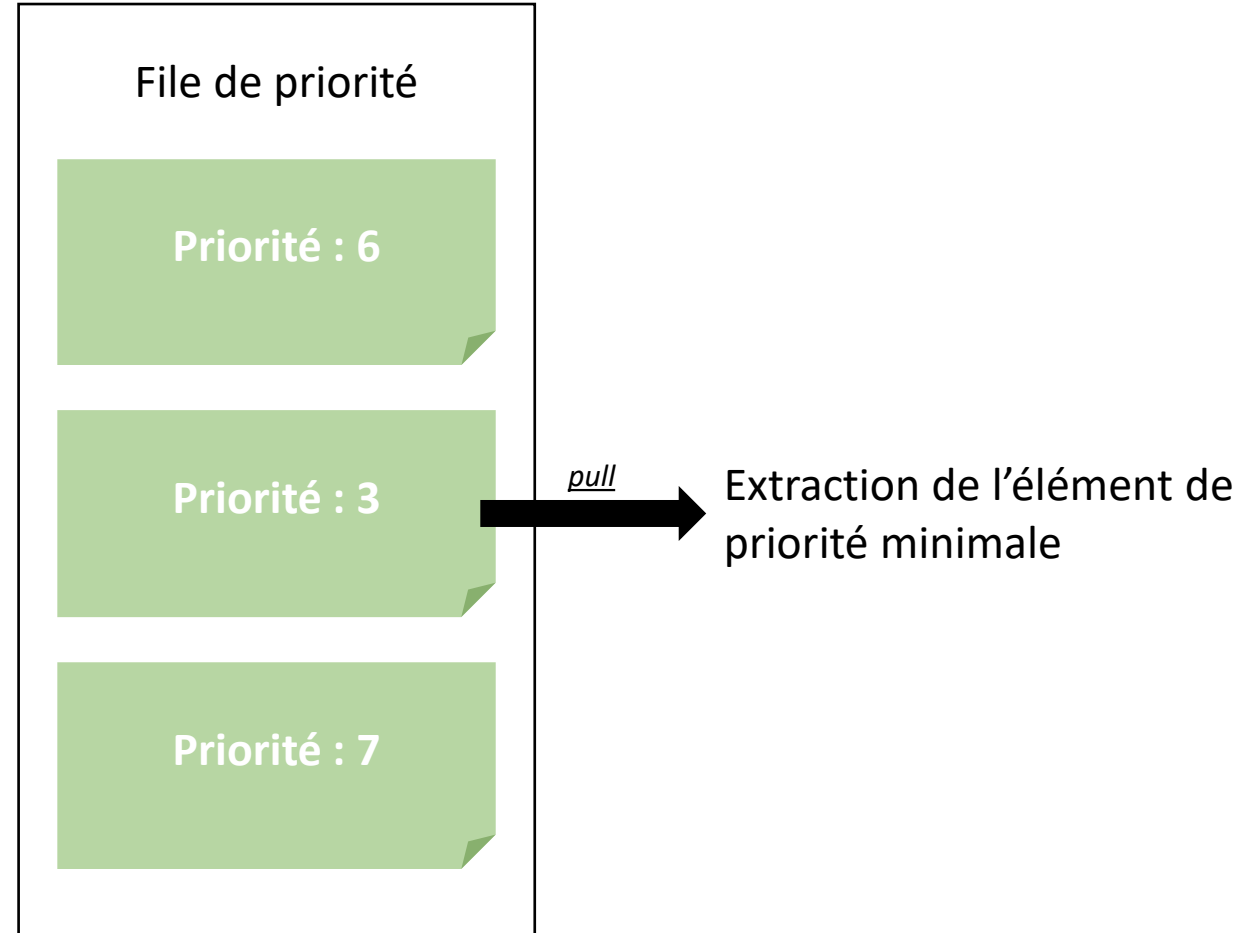
`pull(self) -> Node:`

`is_empty(self) -> bool:`

`has(self, x: Node) -> bool:`

Priorité

Valeur de f



II.2. Algorithme A*

II.2.a. Définitions – notations

II.2.b. Propriétés des nœuds

II.2.c. File de priorité

II.2.d. L'algorithme sur un exemple simple

Sommet de départ : **8**

Sommet d'arrivée : **5**

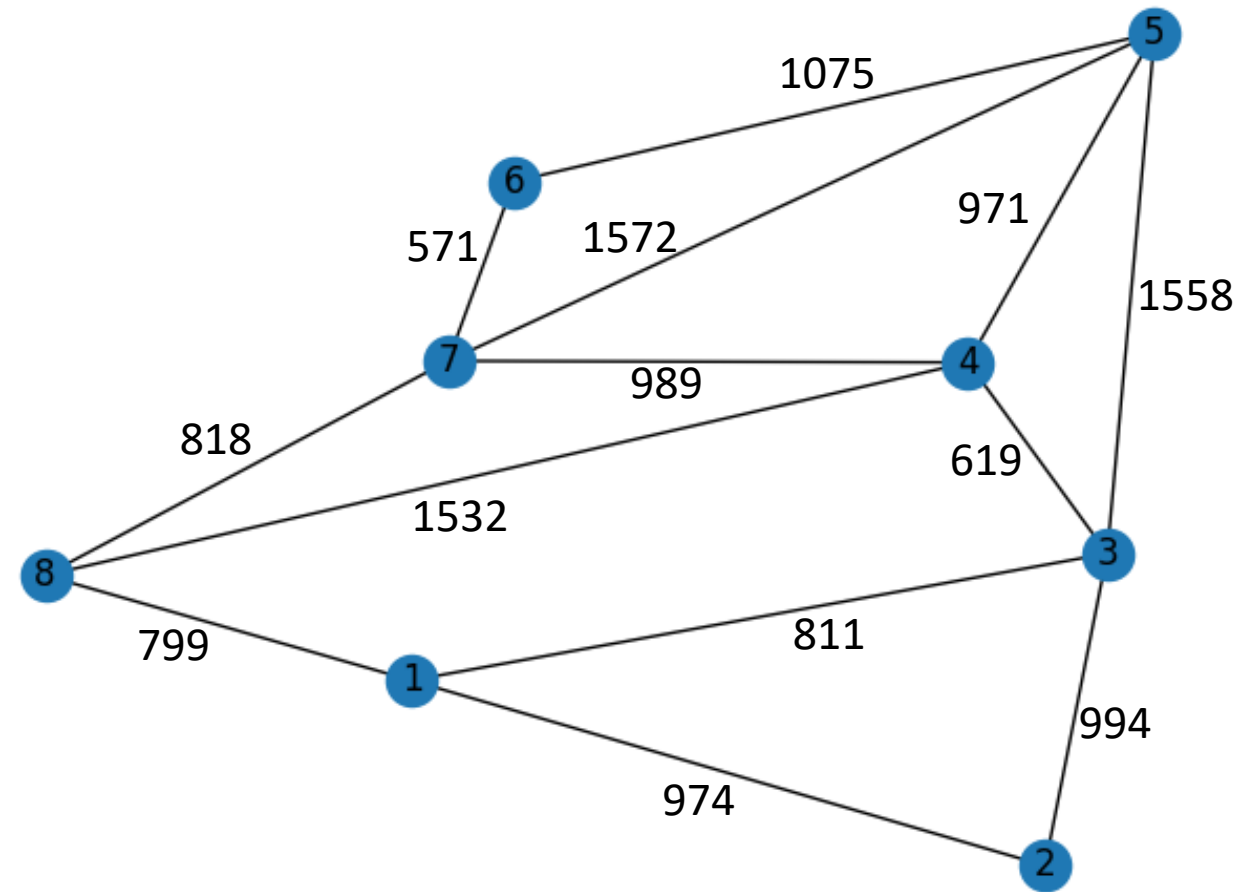
Initialisation du coût de départ & de l'heuristique pour **8**

Extraction de l'élément de priorité minimale : **8**

Voisins de 8 : [**1**, **4**, **7**]

Pour chaque voisin **v** de **8** :

- Déf. du coût de déplacement : $g(v) = g(8) + w(8, v)$
- Calcul de l'heuristique : $h(v)$
- Détermination du nœud parent : **8**
- Ajout de **v** dans la **file de priorité**



File de
priorité

8
2326



II.2. Algorithme A*

II.2.a. Définitions – notations

II.2.b. Propriétés des nœuds

II.2.c. File de priorité

II.2.d. L'algorithme sur un exemple simple

Extraction de l'élément de priorité minimale : **7**

Voisins de 7 : **[5, 6, 4, 8]**

Pour chaque voisin **v** de **7** :

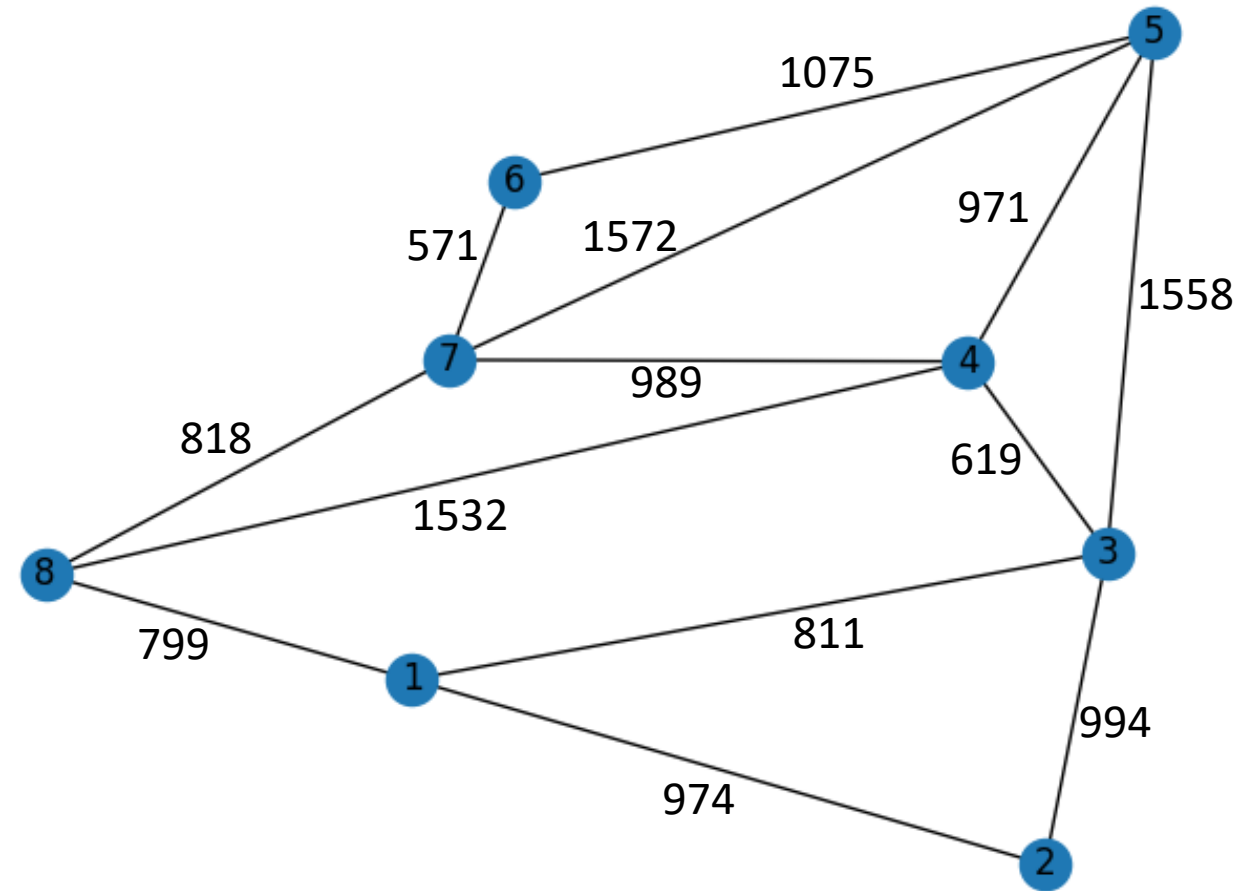
- Nouveau coût de déplacement $g' : g(7) + w(7, v)$
- Si $g(v)$ non déf. ou si $g' < g(v)$

Mise à jour de $g(v)$

Calcul de l'heuristique : $h(v)$

Détermination du nœud parent : **7**

Ajout de **v** dans la **file de priorité**



File de
priorité

1 2972	4 2503	7 2390
-----------	-----------	-----------



II.2. Algorithme A*

II.2.a. Définitions – notations

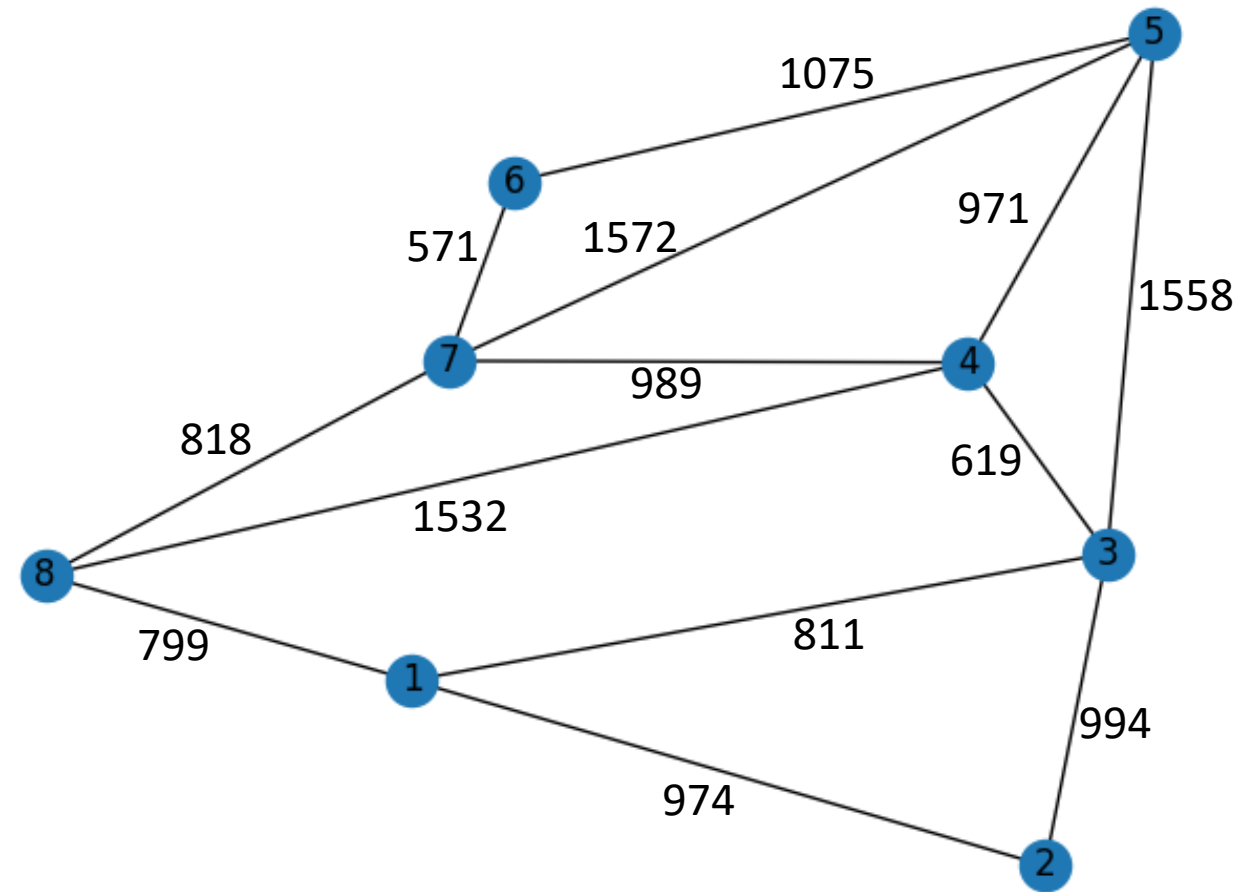
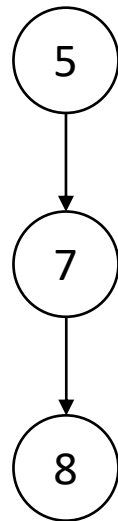
II.2.b. Propriétés des nœuds

II.2.c. File de priorité

II.2.d. L'algorithme sur un exemple simple

Extraction de l'élément de priorité minimale : **5**

Construction du chemin grâce aux nœuds parents :



File de
priorité

6
2465

1
2972

4
2503

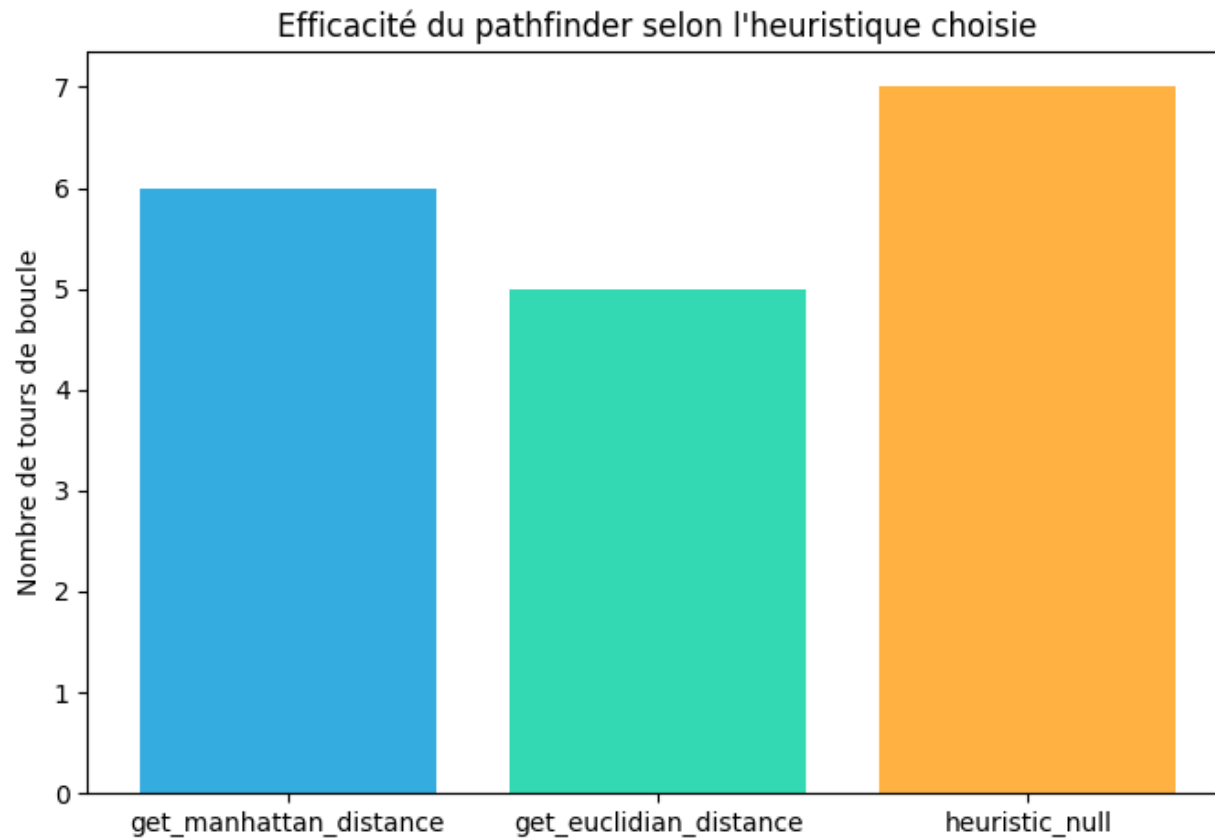
5
2390



II.3. Preuve de l'optimalité de l'algorithme A*

Importance du choix de l'heuristique

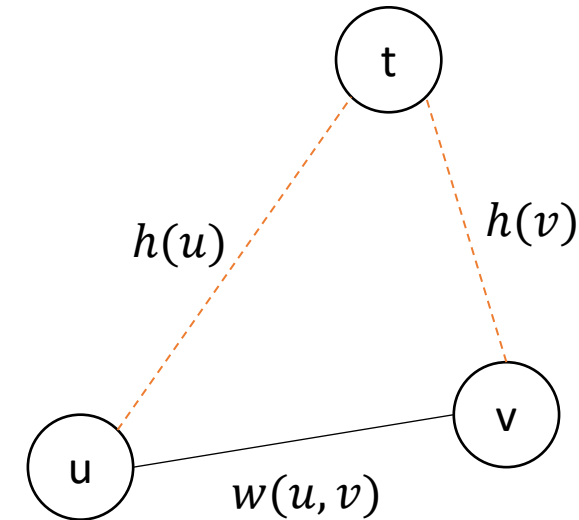
- Temps d'exécution



- Heuristique admissible

Définition : heuristique monotone (pour la recherche de t)

$$\forall (u, v) \in A, h(u) \leq w(u, v) + h(v)$$



Proposition : h est monotone \Rightarrow h est admissible

Proposition : h est admissible \Rightarrow plus court chemin

(Démonstrations en annexe)

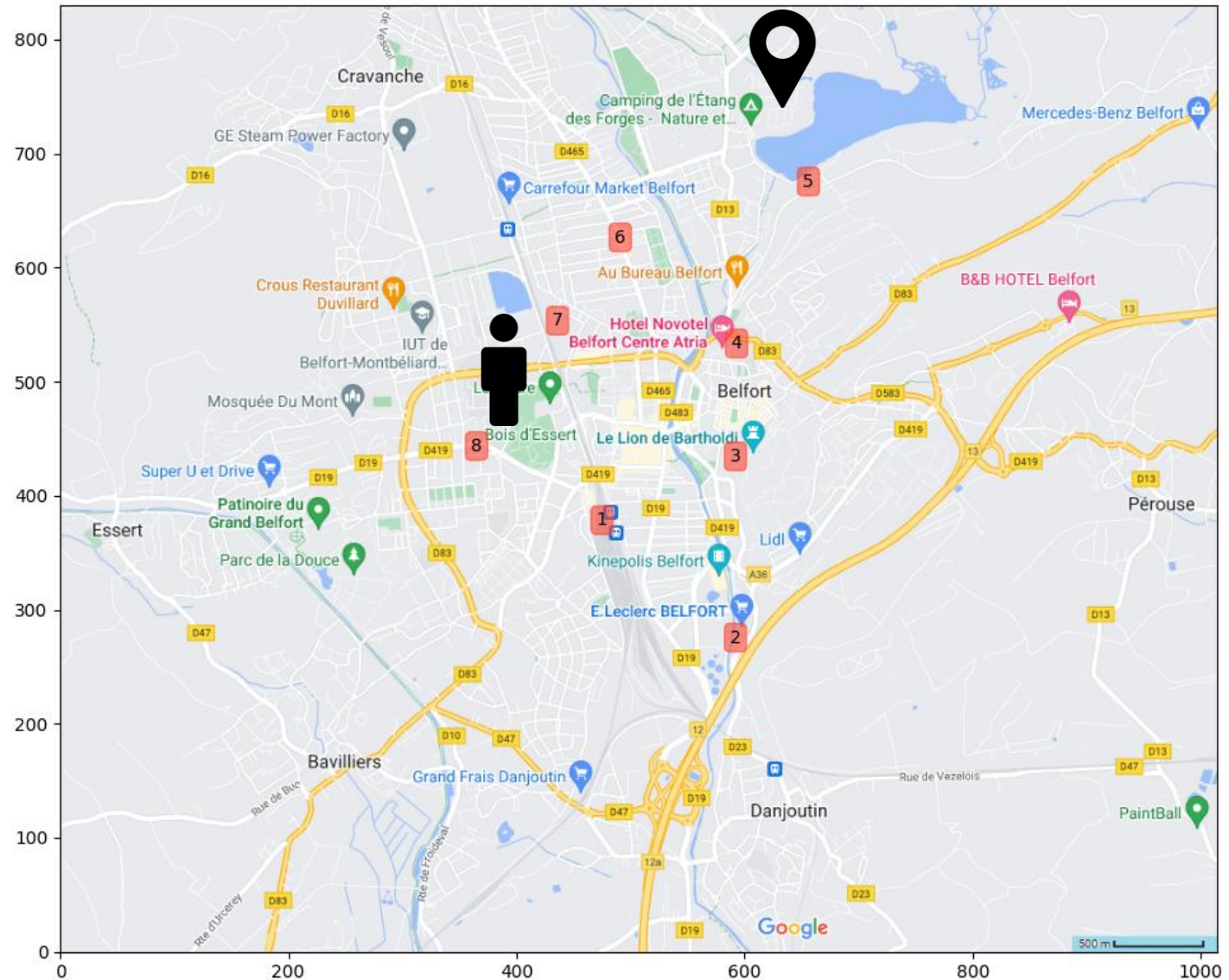
11.4. Example

- Recherche de la station la plus proche du lieu de départ
- Recherche de la station la plus proche de la destination
- Calcul du plus court chemin entre ces deux stations

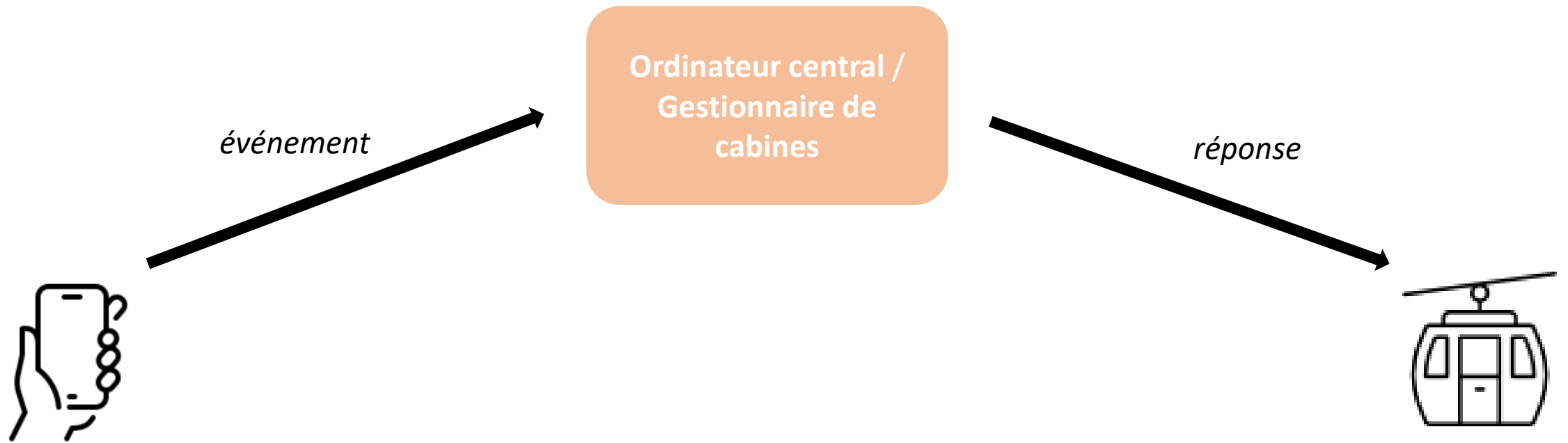
Résultat :

```
Chemin : [8, 7, 5]  
Temps de trajet en cabine : ~3min
```

Comment gérer l'appel d'une cabine ?



III. Répartition des cabines

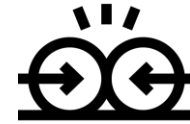


III. Répartition des cabines

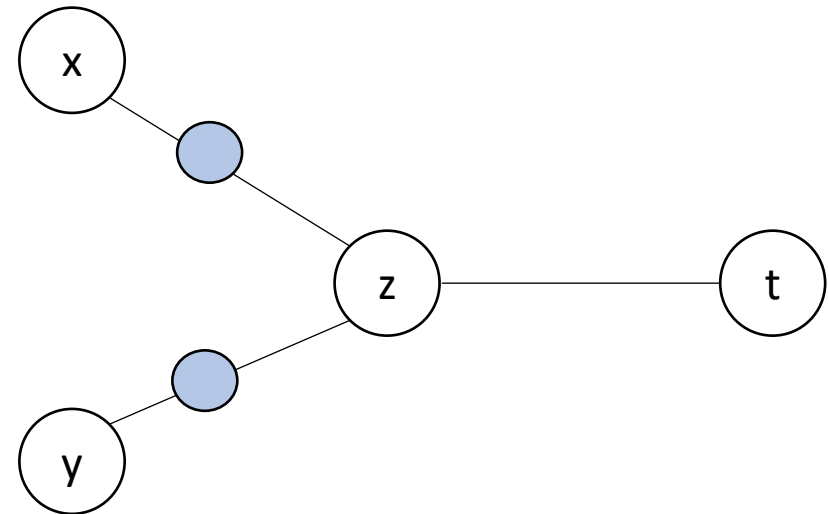
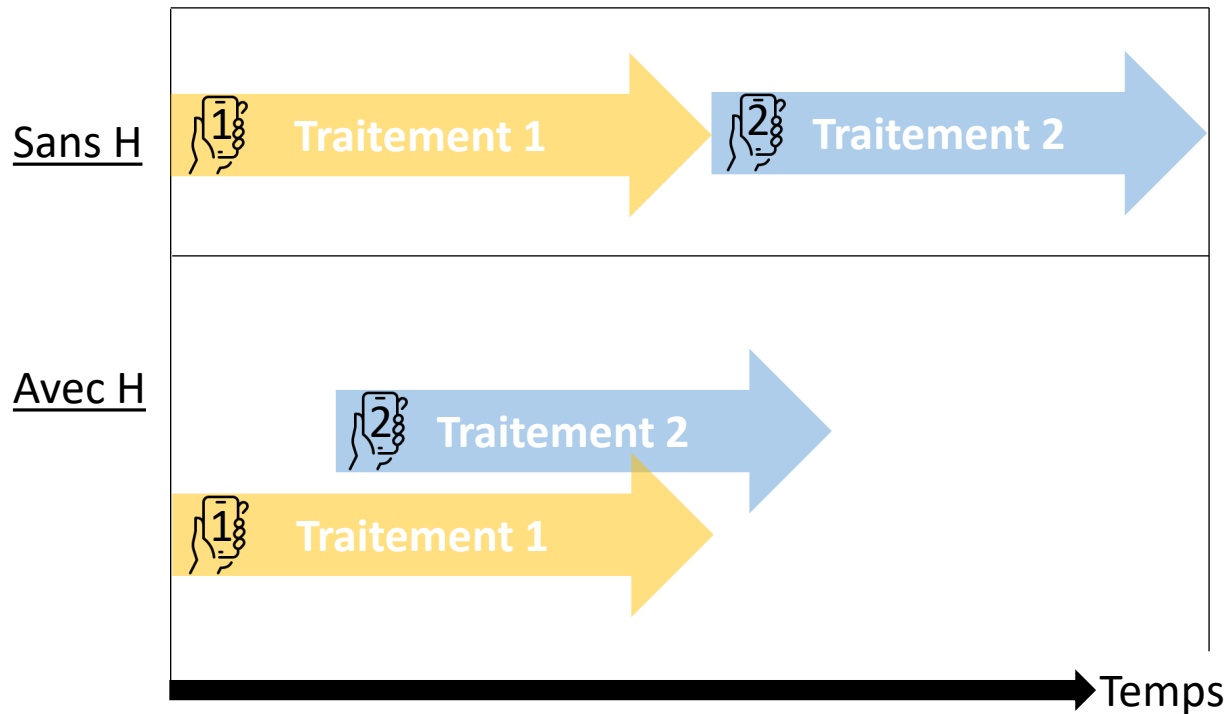
III.1. Hypothèses de travail



Temps réel &
gestion de plusieurs tâches en même temps



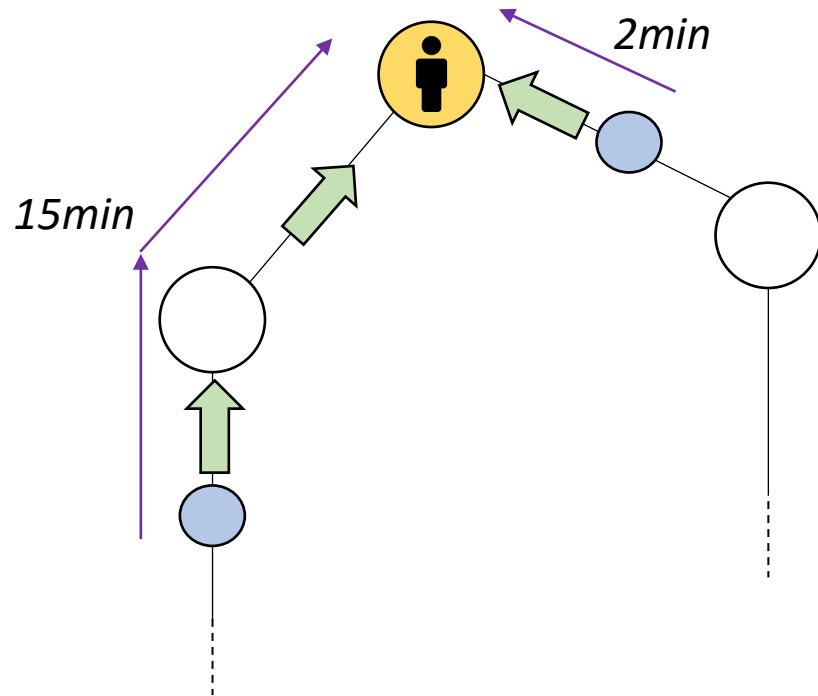
Gestion des collisions



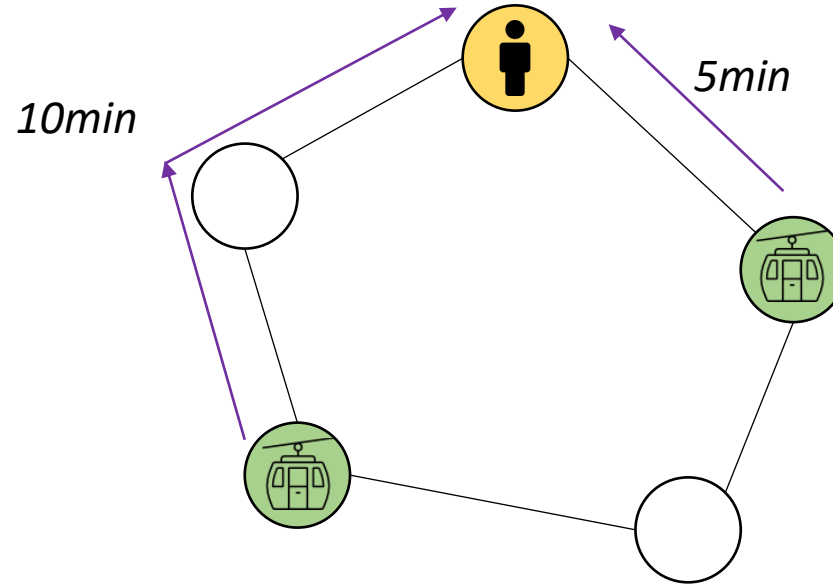
III. Répartition des cabines

III.2. Réponse (appel d'une cabine)

- Attente d'une cabine en cours de route



- Appel d'une cabine depuis une station

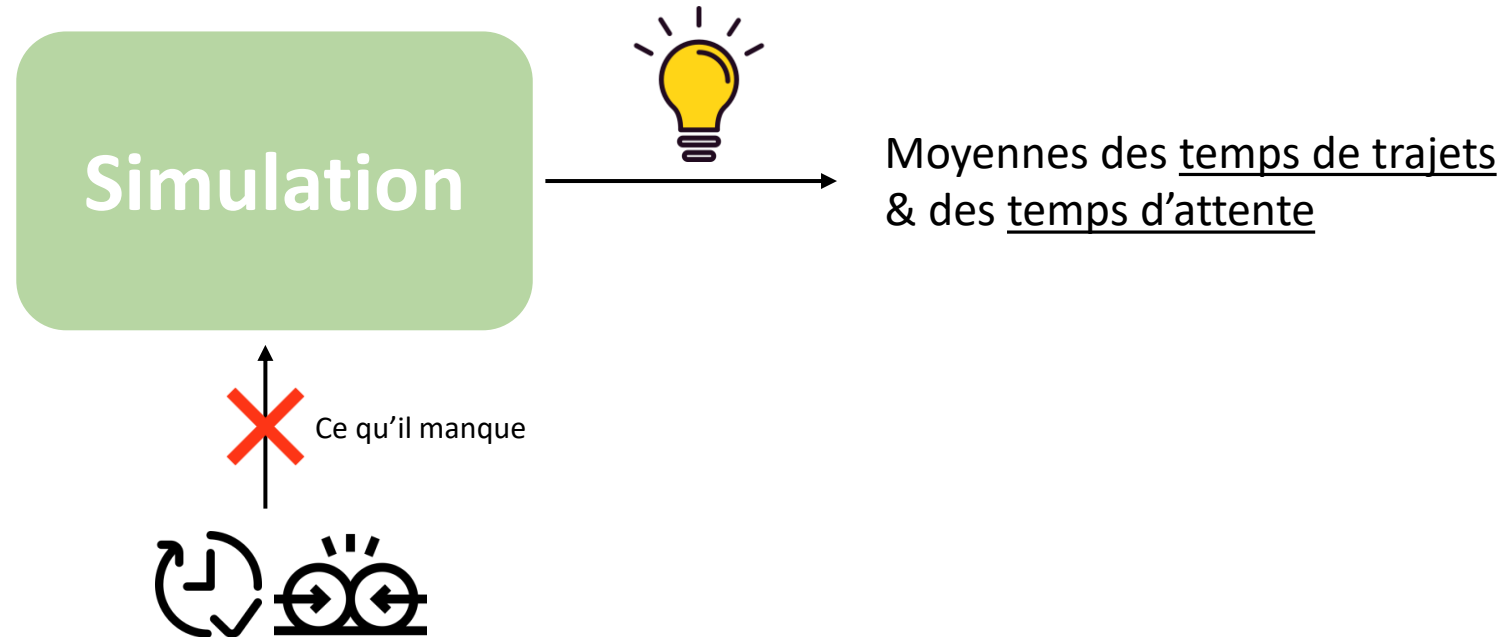


Recherche de ces informations à l'aide de la base de données

III. Répartition des cabines





III.2. Problème rencontré

Optimalité du trafic ?



Conclusion

Validation des objectifs :

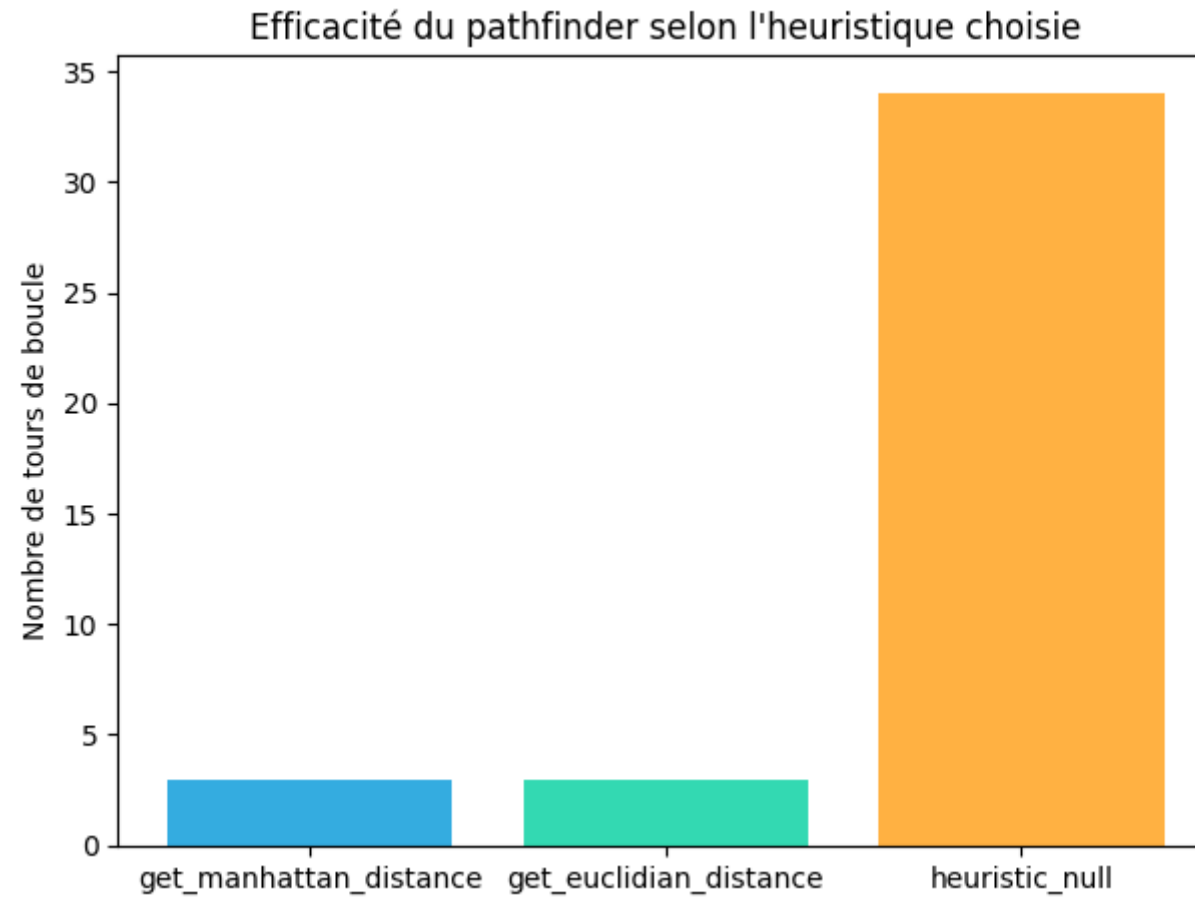
- Représentation visuelle d'un réseau (POO, BDD) 
- Etude de l'algorithme A* 
- Etude de la répartition des cabines en fonction des demandes 
- Comparaison avec un moyen de transport actuel 

Je vous remercie de votre attention.

Annexe

Code et démonstrations

Comparaison des heuristiques pour un réseau plus grand



```
1 from database.database import Database
2 from build_network.map import Map
3 from pathfinder.road_network import RoadNetwork,
  Point
4 from route_manager import RouteRequest,
  GondolaManager
5
6 db = Database("localhost", "root", "", "tipe_ville")
7 city = Map("build_network/img/map3.png", db)
8
9 # Permet de définir les stations (les noeuds)
10 # city.display_interactive_map()
11 city.display_map_with_stations()
12
13 # Permet de définir les liens entre les stations, ie
  lesquelles sont reliées entre elles ?
14 city.set_adjacency_matrix([
15     [0, 1, 1, 0, 0, 0, 0, 1],
16     [1, 0, 1, 0, 0, 0, 0, 0],
17     [1, 1, 0, 1, 1, 0, 0, 0],
18     [0, 0, 1, 0, 1, 0, 1, 1],
19     [0, 0, 1, 1, 0, 1, 1, 0],
20     [0, 0, 0, 0, 1, 0, 1, 0],
21     [0, 0, 0, 1, 1, 1, 0, 1],
22     [1, 0, 0, 1, 0, 0, 1, 0]
23 ])
24
25 city.display_graph()
26 network = RoadNetwork(city.get_stations(), city.
  get_adjacency_matrix())
27 gondola_manager = GondolaManager(city)
28
29 # network.compare(1, 6, [Point.get_manhattan_distance
  , Point.get_euclidian_distance, Point.heuristic_null
  ], ["#34ace0", "#33d9b2", "#fffb142"])
30
31 request = RouteRequest((413, 454), (604, 744))
32 print(request.get_route_data_str(network))
33 gondola_manager.handle_request(request.get_route_data
  (network), network)
34
```

```
1 import time
2 from pathfinder.road_network import RoadNetwork,
  Point
3 from math import ceil
4 from build_network.map import Map
5
6
7 class RouteRequest:
8     """
9     Classe représentant une demande de trajet.
10    """
11    def __init__(self, start_coord: tuple[float,
float], destination_coord: tuple[float, float]):
12        # Coordonnées du point de départ
13        start_x, start_y = start_coord
14        self.start = Point(start_x, start_y)
15
16        # Coordonnées de la destination
17        destination_x, destination_y =
destination_coord
18        self.destination = Point(destination_x,
destination_y)
19
20    def get_route_data_str(self, network: RoadNetwork
) -> str:
21        data = self.get_route_data(network)
22        # print("Chemin : " + str(data[0]))
23        # print("Temps de trajet en cabine : ~" + str
(ceil(data[2])) + "min")
24        return "Vous allez passer par les stations "
+ str(data[0]) + ". Le temps de votre trajet en
cabine est estimé à environ " + str(ceil(data[2])) +
" min, pour une distance de " + str(ceil(data[1])) +
" mètres."
25
26    def get_route_data(self, network: RoadNetwork
) -> tuple[list[int], float, float]:
27        """
28        Connaissant les coordonnées du lieu de départ
et celles de la destination, cette méthode va
déterminer :
```



```

29         - la station la plus proche du point de
        départ
30         - la station la plus proche de la destination
31         Ainsi, elle sera en mesure de déterminer le
        chemin que la cabine devra suivre pour emmener l'
        utilisateur au
32         lieu souhaité.
33         :return: (le chemin de la cabine pour emmener
        l'usager au lieu souhaité, sa longueur, la durée du
        trajet)
34         """
35
36         # Détermination de la station de départ et
        celle d'arrivée
37         stations = network.get_nodes()
38         start_station = stations[0]
39         final_station = stations[0]
40         for station in network.get_nodes():
41             if Point.get_euclidian_distance(self.
        start, station) < Point.get_euclidian_distance(self.
        start, start_station):
42                 start_station = station
43             if Point.get_euclidian_distance(self.
        destination, station) < Point.get_euclidian_distance(
        self.destination, final_station):
44                 final_station = station
45
46         # Calcul du plus court chemin pour aller de
        la station de départ jusqu'à la station d'arrivée
47         path = network.pathfinder(start_station.
        get_id(), final_station.get_id(), Point.
        get_euclidian_distance)[0]
48         network.reset_nodes_properties()
49
50         return network.parse_nodes_by_id(path),
        network.path_weight(path), network.time(path)
51
52
53 class GondolaManager:
54     """
55     Classe représentant le gestionnaire de cabines.

```

```

56     Se charge de la répartition des cabines en
fonction des demandes des usagers.
57     """
58     def __init__(self, city_map: Map):
59         self.city_map = city_map
60
61     def move_gondola(self, start_station_id: int,
destination_station_id: int) -> None:
62         pass
63
64     def handle_request(self, request_data: tuple[list
[int], float, float], network: RoadNetwork) -> None:
65         """
66         V3
67         La demande d'un trajet correspond à un
événement. L'envoi d'une cabine vers la station de
départ constitue la
68         réponse de cet événement. Cette dernière ne
se fait pas au hasard, et doit nécessiter un temps d'
attente
69         minimal pour l'utilisateur.
70
71         Le programme calcule 2 possibilités, et devra
choisir celle qui nécessite le moins de temps d'
attente :
72         - On détermine l'ensemble des trajets en
cours ayant pour destination la station de départ.
73         Pour chacun d'entre eux, on détermine le
temps de trajet restant, pour ensuite déterminer le
plus petit.
74         - On détermine l'ensemble des stations dans
lesquelles il y a au moins une cabine non utilisée.
On regarde
75         si depuis l'une d'entre elles une cabine peut
accéder à la station de départ en un temps inférieur
.
76         Repérage des stations dans lesquelles il y a
des cabines "vides", c'est à dire non utilisées.
77
78         Pour la réponse, il faut connaître :
79         - les données de la demande (pour connaître

```

```

79 la station de départ)
80     - le réseau, pour pouvoir déplacer une
      cabine vers la station de départ (à l'aide du graphe
      donc...)
81     - la base de donnée, pour pouvoir effectuer
      les meilleurs choix en fonction de l'état actuel du
      réseau de
82     transport.
83     :return:
84     """
85     db = self.city_map.db
86
87     # On détermine les stations de départ et d'
arrivée du trajet planifié
88     route_start_node_id = request_data[0][0]
89     route_destination_node_id = request_data[0
][[-1]
90
91     # On définit le temps d'attente t de l'
usager, ainsi que le chemin que suivra la cabine
pour se rendre à la
92     # station souhaitée.
93     t = float("inf")
94     path = None
95
96     # 1. On détermine les trajets planifiés qui
ont pour station d'arrivée la station souhaitée.
97     # MAJ de t : t <- minimum des temps
restants de chacun de trajets
98     req = db.get("SELECT min(arrival) FROM
scheduled_routes WHERE destination_id = " + str(
route_start_node_id) + ";")
99     if req != [(None,)]: # s'il existe bien un
minimum...
100         for scheduled_route in req:
101             arrival = scheduled_route[0] # date
d'arrivée
102             current_time = time.time()
103             temp_t = ceil((arrival -
current_time) / 60)
104             if temp_t < t:

```

```

105             t = temp_t
106             path = None # Le chemin est en
cours... inutile de le renseigner à nouveau
107
108             # 2. On détermine les stations dans
lequelles des cabines sont en attente
109             # On regarde si, depuis l'une d'entre
elle, la cabine peut accéder à notre station en un
temps plus petit
110             req = db.get("SELECT id FROM stations WHERE
current_gondola > 0;")
111             if len(req) != 0:
112                 for station in req:
113                     station_id = station[0]
114                     temp_path = network.pathfinder(
station_id, route_start_node_id, Point.
get_euclidian_distance)[0]
115                     temp_t = network.time(temp_path)
116                     if temp_t < t:
117                         t = temp_t
118                         path = temp_path
119
120             # Attente de la cabine...
121             if path is not None and len(path) > 1:
122                 self.move_gondola(path[0].get_id(), path
[-1].get_id())
123                 print("Une cabine vient d'être envoyée
... Elle arrive dans environ " + str(t) + " minute(s
).")
124             elif t == 0:
125                 print("Une cabine est déjà disponible
sur place !")
126             else:
127                 print("Une cabine arrive dans environ "
+ str(t) + " minute(s).")
128
129             if t != float("inf"):
130                 # La demande est traitée, ajout du
trajet dans la BDD
131                 departure = time.time() + t * 60
132                 arrival = departure + request_data[2] *

```

```
132 60
133         db.set("INSERT INTO scheduled_routes (
            start_id, destination_id, arrival) VALUES (" + str(
            route_start_node_id) + ", " + str(
            route_destination_node_id) + ", " + str(arrival) +
            ");")
134
```

```
1 from math import sqrt, ceil
2 from typing import Self
3 import matplotlib.pyplot as plt
4
5
6 class Point:
7     def __init__(self, x: float, y: float):
8         self.x = x
9         self.y = y
10
11     def get_x(self) -> float:
12         return self.x
13
14     def get_y(self) -> float:
15         return self.y
16
17     @classmethod
18     def get_euclidian_distance(cls, p1: Self, p2:
19 Self) -> float:
20         """
21         Norme 2 dans  $R^2$ 
22         """
23         dx = p2.get_x() - p1.get_x()
24         dy = p2.get_y() - p1.get_y()
25         return sqrt(dx ** 2 + dy ** 2) * 50 / 8 #
26         Prise en compte de l'échelle
27
28     @classmethod
29     def get_manhattan_distance(cls, n1: Self, n2:
30 Self) -> float:
31         """
32         Norme 1 dans  $R^2$ 
33         """
34         dx = abs(n2.get_x() - n1.get_x())
35         dy = abs(n2.get_y() - n1.get_y())
36         return (dx + dy) * 50 / 8 # Prise en compte
37         de l'échelle
38
39     @classmethod
40     def heuristic_null(cls, n1: Self, n2: Self) ->
41 float:
```



```

37         """
38         Fonction nulle
39         """
40         return 0
41
42
43 class Node(Point):
44     """
45     Classe permettant de représenter un noeud d'un
46     graphe.
47     Dans le cadre de l'étude de l'algorithme A*, un
48     noeud possèdera différentes propriétés (toutes
49     initialisées à None)
50     telles que :
51     - le coût de déplacement
52     - l'heuristique
53     - le noeud parent
54     """
55     def __init__(self, id: int, x: int, y: int):
56         super().__init__(x, y)
57         # Identifiant
58         self.id = id
59         # Propriétés
60         self.g = None
61         self.h = None
62         self.parent_node = None
63
64     @classmethod
65     def get_highest_heuristic_node(cls, n1: Self, n2
66     : Self) -> Self:
67         """
68         :param n1:
69         :param n2:
70         :return: Retourne le noeud possédant la plus
71         petite valeur de f
72         """
73         f1 = n1.get_f()
74         f2 = n2.get_f()
75         if f1 < f2:
76             return n1
77         else:

```

```

73         return n2
74
75     def get_id(self) -> int:
76         """
77         :return: Retourne l'identifiant de ce noeud
78         """
79         return self.id
80
81     def get_cost(self) -> int | None:
82         """
83         :return: Retourne le coût associé à ce noeud
84         """
85         return self.g
86
87     def get_heuristic(self) -> float:
88         """
89         :return: Retourne l'heuristique associée à
ce noeud
90         """
91         return self.h
92
93     def set_cost(self, cost: float | None) -> None:
94         """
95         Attribue la valeur 'cost' à 'g'
96         :param cost:
97         """
98         self.g = cost
99
100    def set_heuristic(self, heuristic: float | None
) -> None:
101        """
102        Attribue la valeur 'heuristic' à 'h'
103        :param heuristic:
104        :return:
105        """
106        self.h = heuristic
107
108    def set_parent_node(self, node: Self | None) ->
None:
109        """
110        Attribue un nouveau noeud parent

```

```

111         :param node:
112         :return:
113         """
114         self.parent_node = node
115
116     def get_f(self) -> float:
117         """
118         :return: Retourne la somme du coût et de l'
119         heuristique
120         """
121         return self.get_heuristic() + self.get_cost
122         ()
123
124     def get_parent_node(self) -> Self:
125         """
126         :return: Retourne le noeud parent, ie le
127         noeud duquel on vient pour arriver à CE noeud
128         """
129         return self.parent_node
130
131 class PriorityQueue:
132     def __init__(self, get_highest_priority_element
133     : callable):
134         self.get_highest_priority_element =
135         get_highest_priority_element
136         self.content = []
137
138     def add(self, x: Node) -> None:
139         """
140         Ajoute un élément au contenu de la file de
141         priorité
142         :param x:
143         :return:
144         """
145         self.content.append(x)
146
147     def pull(self) -> Node:
148         """
149         :return: Retourne l'élément possédant la
150         plus grande priorité

```

```

145         """
146         highest = self.content[0]
147         for element in self.content:
148             highest = self.
get_highest_priority_element(highest, element)
149             self.content.remove(highest)
150         return highest
151
152     def is_empty(self) -> bool:
153         """
154         :return: Retourne True si la file est vide,
False sinon
155         """
156         return len(self.content) == 0
157
158     def has(self, x: Node) -> bool:
159         """
160         Retourne True si l'élément x est déjà dans
la file de priorité, False sinon
161         :param x:
162         :return:
163         """
164         return x in self.content
165
166
167 class RoadNetwork:
168     """
169     Classe représentant un réseau routier, ie un
ensemble de routes.
170     Un objet de type 'réseau routier' peut être
instancié grâce à :
171     - un ensemble de points, appelés noeuds (
possédant donc des coordonnées) correspondant ici à
des stations.
172     - une matrice d'adjacence permettant d'
établir les liaisons entre les différents noeuds.
173     """
174     def __init__(self, nodes: list[Node],
adjacency_matrix: list[list[float]]):
175         self.nodes = nodes
176         self.matrix = adjacency_matrix

```

```

177
178         self.network = None
179         self.set_network_matrix()
180
181     @classmethod
182     def parse_nodes_by_id(cls, nodes: list[Node
183 ]) -> list[int]:
184         """
185         :param nodes:
186         :return: Retourne la liste des identifiants
187         de chaque noeud composant la liste 'nodes' entrée en
188         paramètre
189         """
190         return [node.get_id() for node in nodes]
191
192     def get_adjacency_matrix(self) -> list[list[
193 float]]:
194         """
195         :return: Retourne la matrice d'adjacence
196         permettant de décrire les liaisons entre les
197         différents noeuds
198         """
199         return self.matrix
200
201     def get_nodes(self) -> list[Node]:
202         """
203         :return: Retourne la liste des noeuds
204         composant ce réseau
205         """
206         return self.nodes
207
208     def get_node_by_id(self, id: int) -> Node | None
209     :
210         """
211         :param id: identifiant du noeud recherché
212         :return: Retourne le noeud possédant l'
213         identifiant demandé
214         """
215         for node in self.get_nodes():
216             if node.get_id() == id:
217                 return node

```

```

209         return None
210
211     def set_network_matrix(self) -> None:
212         """
213         Initialise la matrice network, notée M,
214         telle que :
215         Quelque soit (i, j) deux identifiants de
216         noeuds,
217         - Si i et j correspondent à deux noeuds liés
218         , alors M[i][j] est la distance les séparant
219         - Si i = j, M[i][j] vaut 0
220         - Si i et j correspondent à des noeuds non
221         liés, alors M[i][j] vaut + l'infini (inf).
222         """
223         self.network = []
224         n = len(self.get_adjacency_matrix())
225         for i in range(n):
226             line = []
227             for j in range(n):
228                 if self.get_adjacency_matrix()[i][j]
229                 ] == 1:
230                     node_i = self.get_node_by_id(i
231                     + 1)
232                     node_j = self.get_node_by_id(j
233                     + 1)
234                     distance = Point.
235                     get_euclidian_distance(node_i, node_j)
236                     line.append(distance)
237                 else:
238                     if i == j:
239                         line.append(0)
240                     else:
241                         line.append(float("inf"))
242             self.network.append(line)
243
244     def get_network(self) -> list[list[float]]:
245         """
246         :return: Retourne la matrice représentant le
247         réseau de transport.
248         """
249         return self.network

```

```

241
242     def get_neighbors(self, node: Node) -> list[Node
    ]:
243         """
244         :param node:
245         :return: Retourne la liste des voisins du
    noeud choisi
246         """
247         # i est la ligne de la matrice d'adjacence
    correspondant au noeud d'identifiant i.
248         i = node.get_id() - 1
249         neighbors = []
250         for j in range(len(self.get_adjacency_matrix
    ()))
251             if self.get_adjacency_matrix()[i][j] ==
    1:
252                 neighbors.append(self.get_node_by_id
    (j + 1))
253         return neighbors
254
255     def build_path_to(self, start: Node, final: Node
    ) -> list[Node]:
256         """
257         Construit récursivement le chemin allant de
    'final' à 'start'.
258         :param final:
259         :param start:
260         :return: Retourne la liste de noeuds
    construite récursivement correspondant à ce chemin.
261         """
262         if final == start:
263             return [final]
264         else:
265             res = self.build_path_to(start, final.
    get_parent_node())
266             res.append(final)
267             return res
268
269     def time(self, path: list[Node]) -> int:
270         """
271         :param path:

```

```

272         :return: la durée d'un trajet
273         """
274         distance = self.path_weight(path)
275         return ceil(distance * 60 / 50000) # Prise
        en compte de la vitesse moyenne d'une cabine
276
277     def weight(self, n1: Node, n2: Node) -> float:
278         """
279         :param n1:
280         :param n2:
281         :return: Retourne le poids de l'arête
        partant de n1 jusqu'à n2.
282         """
283         return self.get_network()[n1.get_id() - 1][
        n2.get_id() - 1]
284
285     def path_weight(self, nodes: list[Node]) ->
        float:
286         """
287         Calcule le poids d'un chemin, défini comme
        la somme des poids des arêtes qui le composent.
288         :param nodes:
289         :return: Retourne la valeur du poids du
        chemin désigné par la liste 'nodes' entrée en
        paramètre.
290         """
291         n = len(nodes)
292         res = 0
293         for k in range(n - 1):
294             res = res + self.weight(nodes[k], nodes[
        k + 1])
295         return res
296
297     def reset_nodes_properties(self) -> None:
298         """
299         Ré-initialise les propriétés de chaque noeud
        . Cette methode doit être appelée avant la recherche
        d'un plus
300         court chemin, pour s'assurer de bien
        initialiser correctement toutes les propriétés de
        chaque noeud.

```



```

301         :return:
302         """
303         for node in self.get_nodes():
304             node.set_parent_node(None)
305             node.set_cost(None)
306             node.set_heuristic(None)
307
308     def pathfinder(self, start_id: int, goal_id: int
309 , heuristic: callable) -> tuple[list[Node], float]:
310         """
311         Implémentation de l'algorithme A*.
312         :param heuristic:
313         :param start_id:
314         :param goal_id:
315         :return: un couple de la forme :
316         (le chemin le plus court allant du noeud
317         ayant pour id 'start_id' au noeud ayant pour id '
318         goal_id', tours de boucle)
319         """
320         # Element d'analyse
321         t = 0
322
323         # Création de la file de priorité
324         prio_queue = PriorityQueue(Node.
325 get_highest_heuristic_node)
326
327         # Initialisation des propriétés du noeud de
328         départ
329
330         start = self.get_node_by_id(start_id)
331         goal = self.get_node_by_id(goal_id)
332         start.set_cost(0)
333         start.set_heuristic(heuristic(start, goal))
334
335         # Ajout du noeud de départ à la file de
336         priorité
337         prio_queue.add(start)
338
339         # Initialisation du noeud actuel :
340         u = start
341         while u != goal and not prio_queue.is_empty
342         ():

```

```

335         u = prio_queue.pull()
336         for neighbor in self.get_neighbors(u):
337             new_cost = u.get_cost() + self.
weight(u, neighbor)
338             if neighbor.get_cost() is None or
new_cost < neighbor.get_cost():
339                 # Mise à jour du coût de
déplacement, du noeud parent et de l'heuristique
340                 neighbor.set_cost(new_cost)
341                 neighbor.set_parent_node(u)
342                 neighbor.set_heuristic(heuristic
(neighbor, goal))
343                 # Ajout de ce voisin dans la
file de priorité
344                 if not prio_queue.has(neighbor):
345                     prio_queue.add(neighbor)
346             t = t + 1
347
348         if u == goal:
349             return self.build_path_to(start, u), t
350         else:
351             return [], t
352
353     def compare(self, start_node_id: int,
final_node_id: int, heuristics: list[callable],
colors: list[str]) -> None:
354         """
355         Affiche un graphique permettant d'analyser l
'efficacité de la méthode pathfinder en fonction des
heuristiques
356         choisies.
357         :param start_node_id:
358         :param final_node_id:
359         :param heuristics:
360         :param colors:
361         """
362         fig, ax = plt.subplots()
363         heuristic_names = []
364         loops_set = []
365
366         for heuristic in heuristics:

```

```
367         path_info = self.pathfinder(  
    start_node_id, final_node_id, heuristic)  
368         loops_number = path_info[1]  
369         self.reset_nodes_properties()  
370         heuristic_names.append(heuristic.  
    __name__)  
371         loops_set.append(loops_number)  
372  
373         ax.bar(heuristic_names, loops_set, color=  
    colors)  
374         ax.set_ylabel("Nombre de tours de boucle")  
375         ax.set_title("Efficacité du pathfinder selon  
    l'heuristique choisie")  
376         plt.show()  
377
```

Contenu du fichier = database.py

File - C:\Users\Jean-Sebastien\Desktop\type-la-ville-master\database\database.py

```
1 import mysql.connector
2
3
4 class Database:
5     def __init__(self, host, user, password, name):
6         self.host = host
7         self.user = user
8         self.password = password
9         self.name = name
10
11         self.connection = None
12         self.cursor = None
13
14     def prepare_db(self):
15         """
16         Méthode permettant de se connecter à la base
17         de données et de positionner l'objet curseur.
18         A appeler avant chaque action !
19         :return:
20         """
21         self.connection = mysql.connector.connect(
22             host=self.host,
23             port=3306,
24             user=self.user,
25             database=self.name,
26             password=self.password
27         )
28
29         self.cursor = self.connection.cursor()
30
31     def close_connection(self):
32         """
33         Méthode permettant de fermer la connexion à
34         la base de données et le curseur.
35         A appeler après chaque action !
36         :return:
37         """
38         self.cursor.close()
39         self.connection.close()
40         self.cursor = None
41         self.connection = None
```

```
40
41     def get(self, sql: str) -> list[tuple]:
42         """
43         Permet de récupérer des informations
44         contenues dans la base de données.
45         :param sql:
46         :return: n-tuple
47         """
48         self.prepare_db()
49         self.cursor.execute(sql)
50         result = self.cursor.fetchall()
51         self.close_connection()
52         return result
53
54     def set(self, sql: str) -> None:
55         """
56         Permet d'effectuer une action sur la base de
57         données (insertion, modification, suppression).
58         :param sql:
59         :return:
60         """
61         self.prepare_db()
62
63         self.cursor.execute(sql)
64         self.connection.commit()
65
66         self.close_connection()
```

Nom du fichier : database.sql

File - C:\Users\Jean-Sebastien\Desktop\lape-la-ville-master\database\database.sql

```
1 --
2 -- Structure de la table `scheduled_routes`
3 --
4
5 CREATE TABLE scheduled_routes (
6   id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
7   start_id INT NOT NULL,
8   destination_id INT NOT NULL,
9   arrival INT NOT NULL
10 );
11
12 --
13 -- Structure de la table `stations`
14 --
15
16 CREATE TABLE stations (
17   id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
18   name VARCHAR(255),
19   localisation_x FLOAT,
20   localisation_y FLOAT,
21   current_gondola INT NOT NULL DEFAULT 0
22 );
```

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3 from PIL import Image
4 from database.database import Database
5 from math import floor
6 from pathfinder.road_network import Node
7 import networkx as nx
8
9
10 class Station(Node):
11     """
12     Représente une station.
13     Hérite de la classe Node qui représente un noeud
14     d'un graphe.
15     """
16     def __init__(self, id, name, x, y):
17         super().__init__(id, x, y)
18         self.name = name
19         self.current_gondola = 0
20
21     def add_gondola(self) -> None:
22         self.current_gondola = self.current_gondola
23         + 1
24
25     def remove_gondola(self) -> None:
26         self.current_gondola = self.current_gondola
27         - 1
28
29     def get_current_number_of_gondola(self):
30         return self.current_gondola
31
32
33 class Map:
34     """
35     Représente la carte d'une ville.
36     """
37     def __init__(self, city_map: str, db: Database =
None):
38         self.map = city_map
39         self.image = np.asarray(Image.open(self.map))
40         self.fig, self.ax = plt.subplots()
```

```

38         self.db = db
39         self.graph = None
40         self.adjacency_matrix = None
41
42     def place_station(self, event) -> None:
43         """
44         Sur la carte de la ville, un clique de souris
45         permet de positionner virtuellement une station.
46         Lors d'un clique, un nouvel enregistrement se
47         fait dans la table "stations" de la base de données.
48         Cet enregistrement contient notamment les
49         coordonnées de la station.
50         :param event:
51         :return:
52         """
53         x = event.xdata
54         y = event.ydata
55         self.db.set("INSERT INTO stations (name,
56 localisation_x, localisation_y) VALUES ('test', " +
57 str(floor(x)) + "," + str(floor(y)) + ");")
58
59     def display_interactive_map(self) -> None:
60         """
61         Affiche la carte de la ville sur laquelle il
62         est possible de positionner des stations.
63         :return:
64         """
65         self.fig.canvas.mpl_connect('
66 button_press_event', self.place_station)
67         plt.axis([0, 1014, 0, 830])
68         plt.imshow(self.image, extent=(0, 1014, 0,
69 830))
70         plt.show()
71
72     def display_map_with_stations(self) -> None:
73         """
74         Affiche la carte de la ville ainsi que les
75         différentes stations.
76         :return:
77         """
78         for station in self.get_stations():

```



```

70         plt.text(x=station.get_x(),
71                  y=station.get_y(),
72                  s=str(station.get_id()),
73                  horizontalalignment="center",
74                  bbox=dict(boxstyle="round",
color="#FF4633", alpha=0.6))
75         plt.axis([0, 1014, 0, 830])
76         plt.imshow(self.image, extent=(0, 1014, 0,
830))
77         plt.show()
78
79     def get_stations(self) -> list[Station]:
80         """
81         :return: L'ensemble des stations de la ville
82
83         """
84         stations = self.db.get("SELECT id, name,
localisation_x, localisation_y FROM stations")
85         stations_object = []
86         for i in range(len(stations)):
87             station = stations[i]
88             id, name, x, y = station[0], station[1
], station[2], station[3]
89             stations_object.append(Station(id, name
, x, y))
90         return stations_object
91
92     def set_adjacency_matrix(self, matrix: list[list
]) -> None:
93         """
94         Initialise la matrice d'adjacence des
stations : celle-ci représente les liens entre elles
95
96         :param matrix:
97         :return:
98         """
99         self.adjacency_matrix = matrix
100
101     def get_adjacency_matrix(self) -> list[list]:
102         """
103         :return: La matrice d'adjacence

```

```

102         """
103         return self.adjacency_matrix
104
105     def display_graph(self) -> None:
106         """
107         Crée et affiche le graphe représentant les
108         stations de la ville
109         :return:
110         """
111         self.graph = nx.Graph()
112         for station in self.get_stations():
113             self.graph.add_node(station.get_id(),
114                                 pos=(station.get_x(), station.get_y()))
115         try:
116             n = len(self.get_adjacency_matrix())
117             for i in range(n):
118                 for j in range(n):
119                     if self.get_adjacency_matrix()[i
120 ][j] == 1:
121                         self.graph.add_edge(i + 1, j
122 + 1)
123         except:
124             print("Il faut créer la matrice d'adj
125 . !")
126
127         nx.draw(self.graph, nx.get_node_attributes(
128 self.graph, 'pos'), with_labels=True)
129         plt.show()
130

```

Une preuve de l'optimalité de l'algorithme A*

Pour toute la suite, on fixe $G = (S, A, w)$ un graphe orienté pondéré par un poids positif.

Définition : heuristique pour la recherche d'un sommet

Soit $t \in S$.

Une heuristique pour la recherche de t est une application $h : S \rightarrow \mathbb{R}_+ \mid h(t) = 0$.

Notation : poids du plus court chemin entre deux sommets

Soit $a, b \in S$.

On note $d(a, b)$ le poids du plus court chemin entre a et b , c'est-à-dire :
 $d(a, b) = \min_p \text{chemin de } a \text{ à } b (w(p))$

Définition : heuristique admissible

Soient $t \in S$ et h une heuristique pour la recherche de t .

On dit que h est *admissible* lorsque $\forall s \in S, h(s) \leq d(s, t)$.

Autrement dit, h est *admissible* si h ne surestime jamais le coût de la résolution.

Définition : heuristique monotone

Soient $t \in S$ et h une heuristique pour la recherche de t .

On dit que h est *monotone* lorsque $\forall (u, v) \in A, h(u) \leq w(u, v) + h(v)$.

Proposition : admissibilité d'une heuristique monotone

Soient $t \in S$ et h une heuristique pour la recherche de t .

Si h est monotone, alors h est admissible.

Démonstration

Par définition d'une heuristique admissible, montrons que : $\forall s \in S, h(s) \leq d(s, t)$.
Soit alors $s \in S$.

Soit un plus court chemin de s à t noté $s_0 \dots s_n$. Montrons par récurrence que :

$$\forall i \in [0, n], h(s_i) \leq \sum_{j=i}^{n-1} w(s_j, s_{j+1}).$$

- Initialisation :

$$h(s_n) = h(t) = 0 = \sum_{j=n}^{n-1} w(s_j, s_{j+1}).$$

- Hérédité : soit $i \in [1, n]$ tel que $h(s_i) \leq \sum_{j=i}^{n-1} w(s_j, s_{j+1})$. Montrons que

$$h(s_{i-1}) \leq \sum_{j=i-1}^{n-1} w(s_j, s_{j+1}). \text{ Par monotonie de } h, \text{ on a}$$

$$h(s_{i-1}) \leq w(s_{i-1}, s_i) + h(s_i). \text{ Or, par hypothèse de récurrence on a } h(s_i) \leq \sum_{j=i}^{n-1} w(s_j, s_{j+1}),$$

$$\text{donc } w(s_{i-1}, s_i) + h(s_i) \leq w(s_{i-1}, s_i) + \sum_{j=i}^{n-1} w(s_j, s_{j+1}) = \sum_{j=i-1}^{n-1} w(s_j, s_{j+1}), \text{ d'où finalement :}$$

$$h(s_{i-1}) \leq \sum_{j=i-1}^{n-1} w(s_j, s_{j+1}). \text{ La propriété est donc héréditaire.}$$

- Conclusion : la propriété à démontrer étant initialisée et héréditaire, elle est donc vraie.

$$\text{Elle est en particulier vraie pour } i = 0 : h(s_0) = h(s) \leq \sum_{j=0}^{n-1} w(s_j, s_{j+1}) = d(s, t) \text{ (puisque}$$

$s_0 \dots s_n$ est un plus court chemin de s à t). Ce résultat étant vrai quelque soit s , h est donc admissible. CQFD.

Proposition : optimalité de l'algorithme A*, implémenté par la méthode pathfinder

Soient $s, t \in S$ et h une heuristique admissible pour la recherche de t .

Si l'appel de la méthode pathfinder avec comme paramètres les sommets s et t retourne un chemin, alors ce dernier est un plus court chemin de s à t .

Démonstration

Quelques notations : pour un sommet s donné, on note $g(s)$ (resp. $f(s)$) le coût total pour accéder à s (resp. la priorité de s).

Soit $C_{A^*} = s_0 \dots s_m$ le chemin retourné par l'algorithme A*, où $s_0 = s$ et $s_m = t$. On note d le poids de ce chemin. Notons qu'au moment de l'extraction de t , sa priorité est $f(t) = g(t) + h(t)$. Comme h est une heuristique pour la recherche de t , $h(t) = 0$. Il vient donc que $f(t) = g(t) = d$ (désignons par \star cette égalité), puisque $g(t)$ correspond au coût de déplacement total pour aller en t .

On suppose maintenant par l'absurde qu'il existe un chemin $C = a_0 \dots a_n$, où $a_0 = s$ et $a_n = t$, de poids minimal d' . Ainsi : $d' < d$.

Montrons par récurrence la propriété suivante : $\forall i \in [0, n]$, $g(a_i) = d(s, a_i)$ et $f(a_i) \leq d'$.

- Initialisation ($i = 0$) :

D'une part : $g(a_i) = g(a_0)$ et $g(a_0) = 0$ puisqu'au début de l'algorithme, le coût g associé au sommet a_0 est initialisé à 0. De plus, $d(s, a_i) = d(s, a_0) = d(s, s) = 0$, donc $g(a_0) = d(s, a_0)$ car le poids est positif.

D'autre part : $f(a_i) = f(a_0) = g(a_0) + h(a_0) = h(a_0) = h(s)$. Or, h est une heuristique admissible pour la recherche de t , ce qui implique que $h(s) \leq d(s, t)$. Notons que $d(s, t) = d'$ puisque par définition, $d(s, t)$ est le poids du plus court chemin de s à t . On obtient donc l'inégalité $f(a_0) \leq d'$.

Le sommet de départ vérifie les deux conditions de la propriété que l'on cherche à démontrer : elle est donc initialisée.

- Hérédité : soit $i \in [0, n]$ | $g(a_i) = d(s, a_i)$ et $f(a_i) \leq d'$. Montrons que $g(a_{i+1}) = d(s, a_{i+1})$ et $f(a_{i+1}) \leq d'$.

Tout d'abord, d'après l'hypothèse de récurrence, le sommet a_i possède la priorité : $f(a_i) \leq d' < d$. Il est donc extrait avant le sommet t . Considérons maintenant son voisin a_{i+1} et distinguons deux cas, imposés par l'algorithme :

1. Si $g(a_{i+1})$ n'est pas défini ou si $g(a_{i+1})$ est défini et que $g(a_i) + w(a_i, a_{i+1}) < g(a_{i+1})$;

Alors l'algorithme attribue à $g(a_{i+1})$ la valeur $g(a_i) + w(a_i, a_{i+1})$, donc à ce stade, $g(a_{i+1}) = g(a_i) + w(a_i, a_{i+1})$. Or, par hypothèse de récurrence, $g(a_i) = d(s, a_i)$. On déduit donc que $g(a_{i+1}) = d(s, a_i) + w(a_i, a_{i+1}) = d(s, a_{i+1})$. (En effet, si $d(s, a_i) + w(a_i, a_{i+1}) > d(s, a_{i+1})$, alors le chemin $sa_1 \dots a_i a_{i+1}$ ne serait pas un chemin de poids minimal de s à a_{i+1} . Il existerait donc un chemin $s \dots a_{i+1}$, noté \mathcal{L} , tel que $w(\mathcal{L}) < w(sa_1 \dots a_i a_{i+1})$. Notons $\tilde{\mathcal{L}}$ le chemin $\mathcal{L}a_{i+1}a_{i+2} \dots t$. On aurait donc $w(\tilde{\mathcal{L}}) = w(\mathcal{L}) + w(a_{i+1}a_{i+2} \dots t) < w(s \dots a_i a_{i+1}) + w(a_{i+1}a_{i+2} \dots t) = w(\mathcal{C})$. Or, le chemin \mathcal{C} est un plus court chemin de s à t , donc $w(\mathcal{C}) = d(s, t)$ ce qui impliquerait que $w(\tilde{\mathcal{L}}) < d(s, t)$: c'est absurde).

De plus, $g(a_{i+1})$ étant maintenant défini, on peut calculer $f(a_{i+1})$: $f(a_{i+1}) = g(a_{i+1}) + h(a_{i+1})$. Or, h est admissible donc $h(a_{i+1}) \leq d(a_{i+1}, t)$. On a alors que : $f(a_{i+1}) \leq d(s, a_{i+1}) + d(a_{i+1}, t) = d'$.

2. Sinon, $g(a_{i+1})$ est déjà défini et $g(a_{i+1}) \leq g(a_i) + w(a_i, a_{i+1})$;

Par hypothèse de récurrence sur $g(a_i)$, on a $g(a_{i+1}) \leq d(s, a_i) + w(a_i, a_{i+1})$, donc nécessairement $g(a_{i+1}) = d(s, a_i) + w(a_i, a_{i+1})$ par minimalité de $d(s, a_i)$. On établit alors que $g(a_{i+1}) = d(s, a_{i+1})$.

En reprenant le même raisonnement fait précédemment pour la majoration de $f(a_{i+1})$, on a : $f(a_{i+1}) \leq d'$.

Ainsi, dans tous les cas, la propriété est héréditaire.

- Conclusion : la propriété étant initialisée et héréditaire, elle est alors vraie. Elle est en particulier vraie pour le dernier sommet t ; $f(t) = f(a_n) \leq d' < d$, ce qui contredit l'égalité désignée par \star . L'hypothèse d'un chemin plus court que celui retourné par A^* aboutit à une contradiction, ce qui prouve l'optimalité du chemin renvoyé par l'algorithme A^* . CQFD.