Numéro de candidat : **40657**

Thème : **La ville**

Exploitation optimale des couloirs aériens pour une nouvelle mobilité urbaine

Problématique retenue

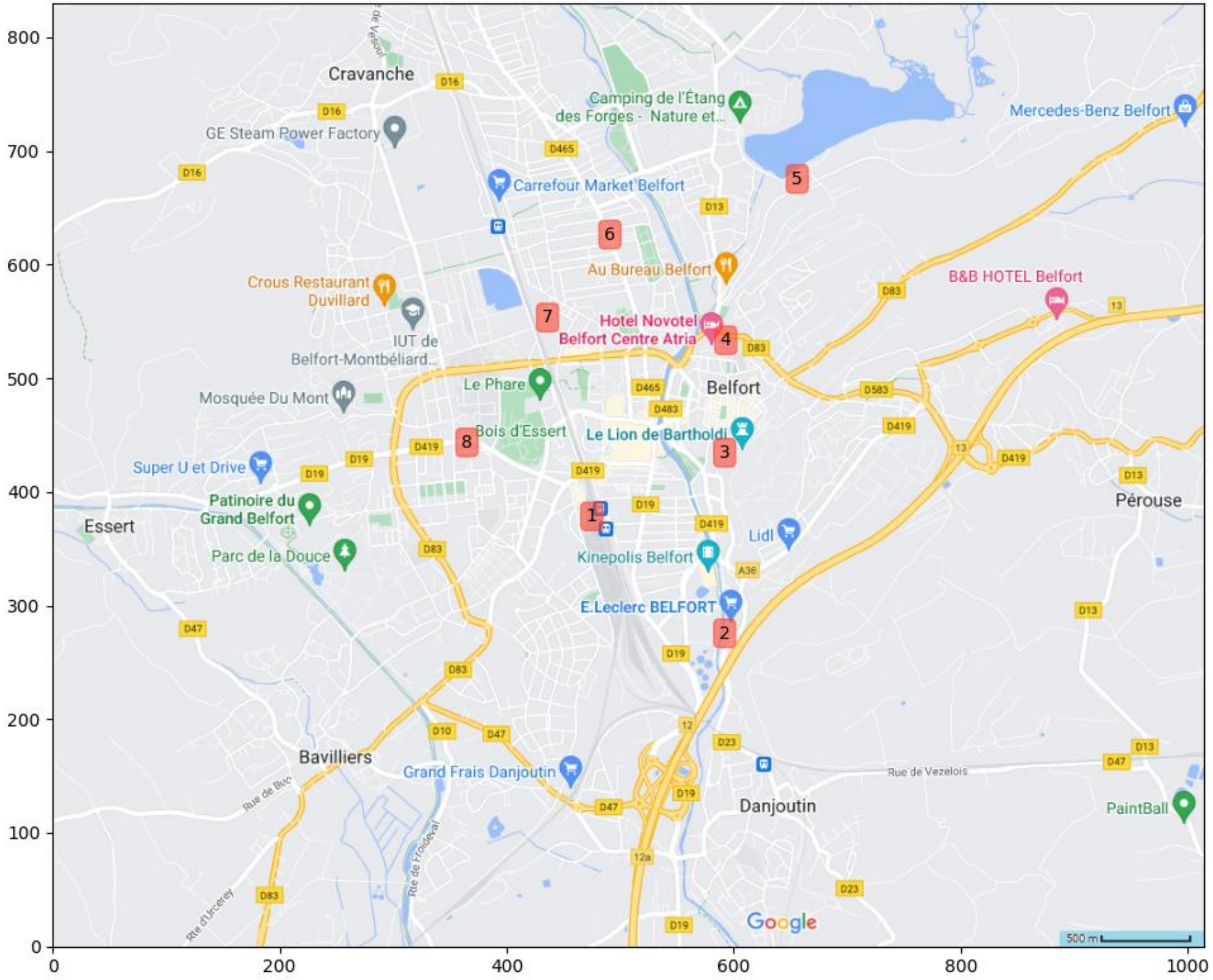
Comment peut-on implémenter des algorithmes permettant d'une part de représenter le fonctionnement d'un nouveau moyen de transport urbain et d'autre part de s'approcher d'une fluidité optimale du trafic ?

Sommaire

I. Principe et implémentation possible

I.1. Généralités

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

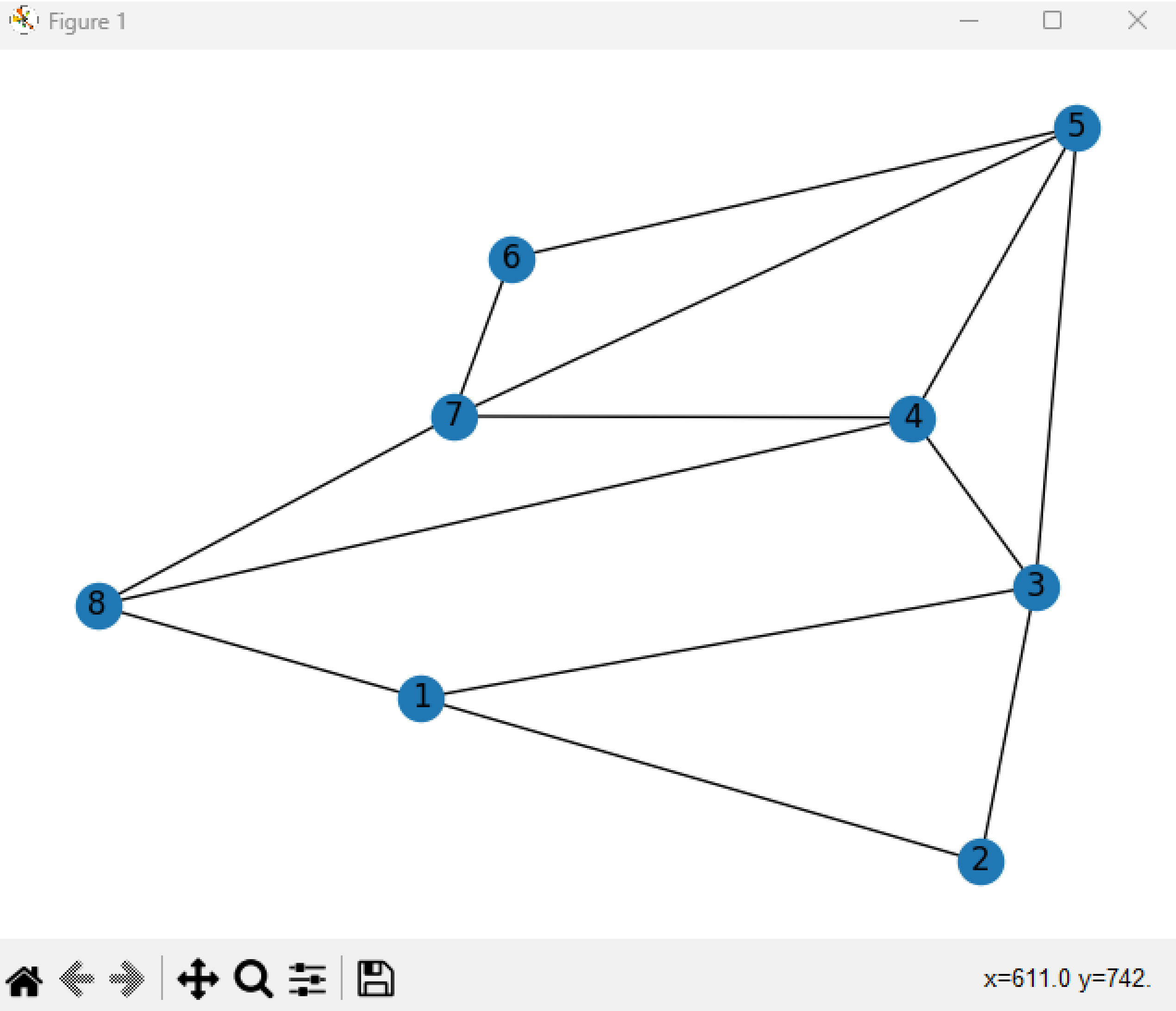


I.1. Généralités

Représentation du graphe
Bibliothèque Python utilisée : *NetworkX*

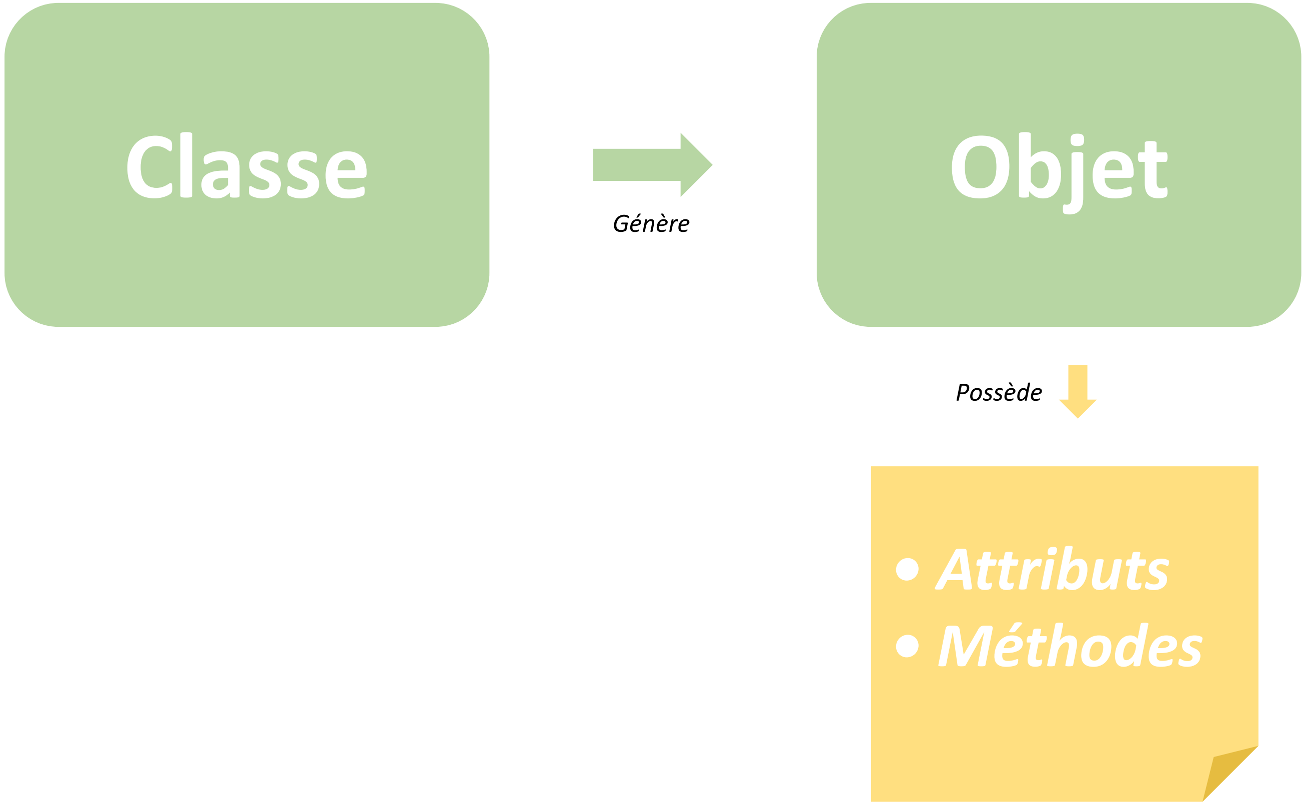
Source : https://networkx.org

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

```python
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
```

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

```sql
CREATE TABLE stations (
  id INT PRIMARY KEY NOT NULL AUTO_INCREMENT,
  name VARCHAR(255),
  localisation_x FLOAT,
  localisation_y FLOAT,
  capacity INT DEFAULT 25,
  current_people INT NOT NULL DEFAULT 0,
  current_gondola INT NOT NULL DEFAULT 0,
  is_main tinyint(1) NOT NULL
);
```

Structure de la classe *Database*

```python
class Database:
    def __init__(self, host, user, password, name):
        self.host = host
        self.user = user
        self.password = password
        self.name = name

        self.connection = None
        self.cursor = None
```

Signatures des méthodes principales

```python
def get(self, sql: str) -> list[tuple]:

def set(self, sql: str) -> None:
```

I.3. Réalisation de la carte interactive

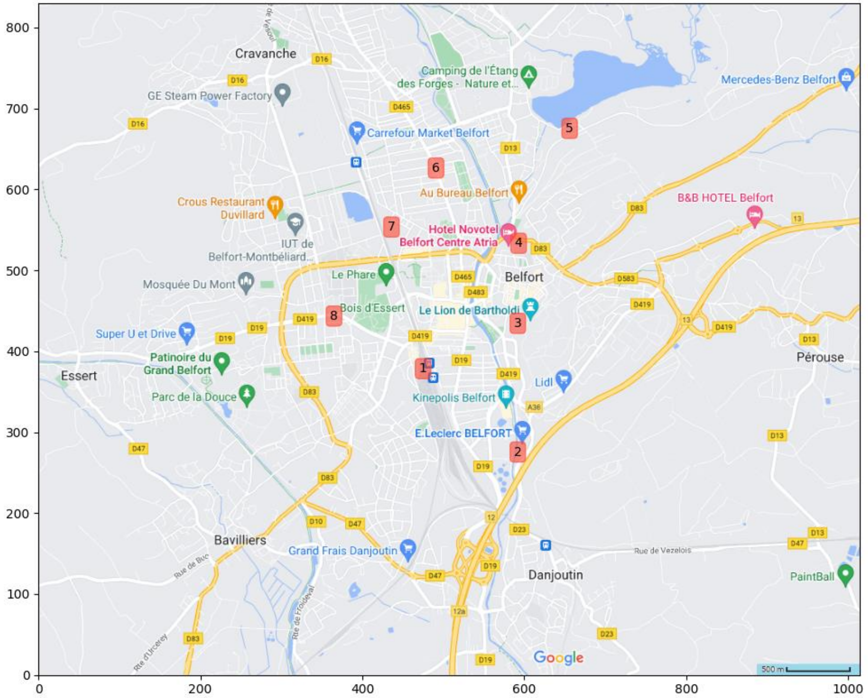
Démarche

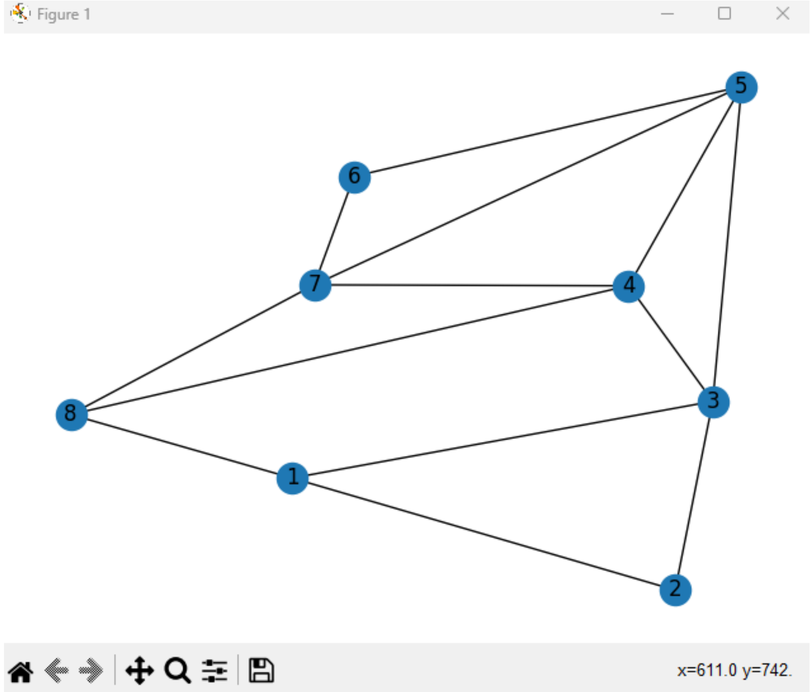
1. Recherche du plan de la ville de Belfort
2. Documentation de la bibliothèque *Matplotlib*
3. Création d'une classe *Map*, comportant 3 fonctionnalités principales :

    - Affichage du plan (interactif)
    - Positionnement des stations sur le plan & insertion dans la BDD
    - Affichage du graphe

Intérêt

Etude de l'optimalité du trafic

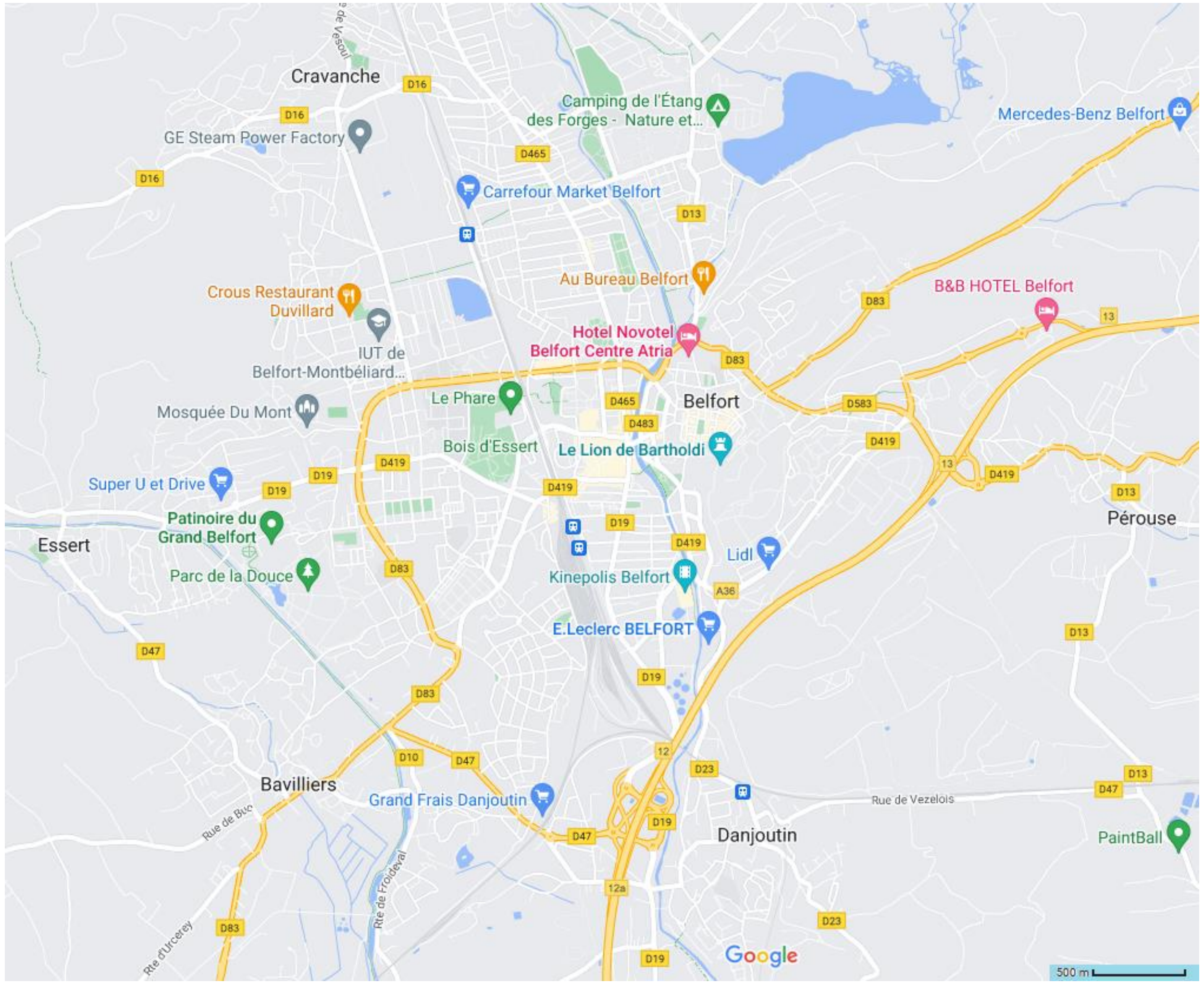




II. Algorithme important : algorithme A*

II.1. Mise en situation – Personal Rapide 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

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

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

II.2.b. Propriétés des noeuds

- g : coût de déplacement
- h : heuristique
- f : g + h (heuristique totale)
- Nœud parent

Getters

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

Setters

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

II.2. Algorithme A*

II.2.a. Définitions – notations
II.2.b. Propriétés des noeuds
II.2.c. File de priorité

Interface

```
def add(self, x: Node) -> None:
def pull(self) -> Node:
def is_empty(self) -> bool:
def has(self, x: Node) -> bool:
```

Définition de la classe

```
class PriorityQueue:
    def __init__(self, get_highest_priority_element: callable):
        self.get_highest_priority_element = get_highest_priority_element
        self.content = []
```

II.2. Algorithme A*

II.2.a. Définitions – notations
II.2.b. Propriétés des noeuds
II.2.c. File de priorité
II.2.d. Pseudo code

```
# Initialisation des variables
F = file de priorité
start        <- Noeud de départ
goal         <- Noeud final
start.g      <- 0
start.h      <- heuristique évaluée en start
Ajouter start à F
current_node <- start

# Boucle principale
Tant que (current_node != goal) et (F n'est pas vide):
  Extraire l'élément de priorité minimale de F et le nommer current_node.
  Pour chaque voisin n de current_node :
    current_cost = current_node.cost + w(current_node, n)
       Si (n.cost n'est pas défini) ou (si current_cost est plus petit que n.cost) :
         n.cost = current_cost
         n.parent_node = current_node
         n.h = heuristique évaluée en n
         Ajouter n à la file de priorité F

Si u = goal, retourner le chemin
Sinon, échec
# Fin du programme
```

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

Proposition : heuristique admissible => plus court chemin

Démonstration (par l'absurde) :

$\mathcal{C}_{\mathcal{A}^*} = s_0 \ldots s_m$: chemin de poids *d* retourné par A*

$f(t) = g(t) = d \quad \star$

$\mathcal{C} = a_0 \ldots a_n$: chemin de poids minimal *d'*.

Récurrence : $\forall i \in [|\ 0, n\ |], g(a_i) = d(s, s_i)$ et $f(a_i) \leq d'$.

- Initialisation : $g(a_0) = d(s, a_0) \qquad f(a_0) \leq d'$
- Hérédité : soit $i \in [|\ 0, n\ |] \mid g(a_i) = d(s, a_i)$ et $f(a_i) \leq d'$.
  Montrons que $g(a_{i+1}) = d(s, a_{i+1})$ et $f(a_{i+1}) \leq d'$.

(HR) $f(a_i) \leq d' < d$.

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

$$g(a_{i+1}) = g(a_i) + w(a_i, a_{i+1})$$

$$g(a_{i+1}) = d(s, a_i) + w(a_i, a_{i+1}) = d(s, a_{i+1}) \quad \text{(HR)}$$

$$f(a_{i+1}) \leq d(s, a_{i+1}) + d(a_{i+1}, t) = d' \quad \text{(Admissibilité)}$$

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

$$g(a_{i+1}) \leq d(s, a_i) + w(a_i, a_{i+1}). \quad \text{(HR)}$$

$$g(a_{i+1}) = d(s, a_i) + w(a_i, a_{i+1})$$

$$f(a_{i+1}) \leq d'$$

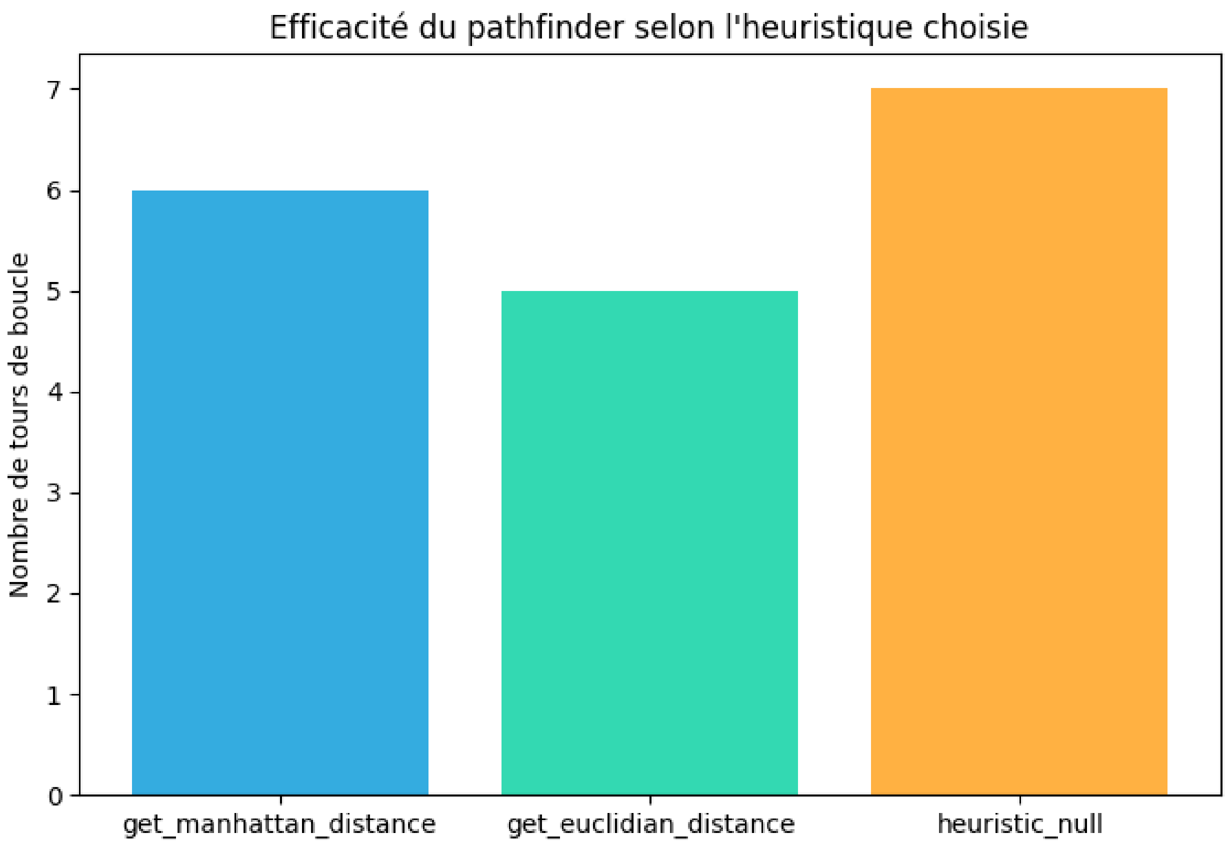
- Conclusion :

$f(t) = f(a_n) \leq d' < d$, absurde. ■

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

Importance du choix de l'heuristique

- Temps d'exécution



- Heuristique admissible

Définition : heuristique monotone

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

Proposition : h monotone => h admissible

Démonstration (par récurrence) :
Soient s un sommet, un plus cours chemin de s à t noté $s_0 \cdots s_n$. Récurrence :

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

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

- Hérédité : $h(s_{i-1}) \leqslant w(s_{i-1}, s_i) + h(s_i)$ (Monotonie)

$$\leqslant w(s_{i-1}, s_i) + \sum_{j=i}^{n-1} w(s_j, s_{j+1}) \quad \text{(HR)}$$

$$= \sum_{j=i-1}^{n-1} w(s_j, s_{j+1}) \quad \blacksquare$$

III. Répartition des cabines selon les demandes

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 ✗