Rapport IA41

Esteban Becker | Pierre-Olivier Cayetanot | Yann Derré | Tom Palleau

A22

Résumé

Notre projet planifie les livraisons qu'un livreur doit effectuer entre ses arrêts. Ainsi, l'utilisateur (le livreur) pourra entrer son point de départ, ainsi qu'une liste de destinations à livrer. En utilisant les données d'un site de cartographie, le programme va planifier l'itinéraire le plus rapide pour sa tournée et l'afficher à l'utilisateur.

Pour résoudre ce problème, nous mettrons en applications les notions et algorithmes vus en cours. Nous avons également implémenté d'autres algorithmes nécessaires afin de relier les destinations entre elles de manière optimale.

Afin de rester dans les limites de la matière, nous nous sommes concentrés principalement sur l'implémentation de la recherche du graphe, choisissant d'utiliser des librairies déjà existantes comme OSMnx pour l'acquisition des données depuis l'API d'OpenStreetMap, et l'affichage des résultats via Folium.

Enfin, nous reflèterons sur les limitations des technologies choisies, ainsi que les évolutions futures pour une éventuelle implémentation sur le terrain.

Voici la répartition de la charge de travail sur ce projet :
— Yann Derré : Recherche des librairies, création de l'interface, implémentation A*
— Esteban Becker : Implémentation TSP | Ants Algorithhm, Dijkstra
— Pierre-Olivier Cayetanot : Implémentation TSP | Christofides
— Tom Palleau : Implémentation Pairwise exchange
Le code est consultable sur GitHub : github.com/pcayetan/projet_IA41

Table des matières

Première partie

Interface et choix des librairies

L'interface est réalisée avec la librarie PyQt, permettant de réaliser des interfaces QT directement depuis notre projet Python.

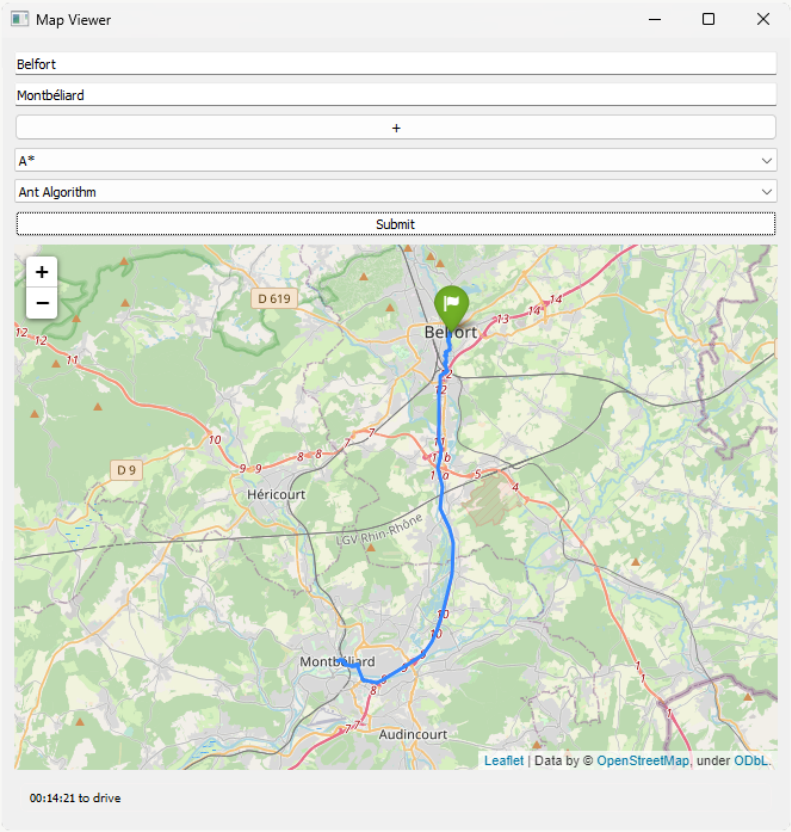


FIGURE 1 – UI du programme

Nous avons également fait appel à d'autres librairies pour remplir les fonctions en dehors de l'UV, notamment la génération de la carte, ainsi que l'acquisition des données depuis OpenStreetMap.

- la librairie **OSMnx** nous permet d'obtenir les données en JSON depuis l'API d'OpenStreetMap, et nous les transforme en Graph NetworkX avec lequel nous interagissons pour l'exploration et la simplification.
- la librairie **Folium** nous permet de générer une carte interactive en Javascript en fournissant directement notre Graphe. Il utilise également les calques d'OpenStreetMap, nous permettant de superposer notre route de manière précise.
- la librairie **goip2** nous permet de localiser l'utilisateur afin de prégénérer en cache un graphe de sa zone locale avant l'affichage du programme pour une recherche dans son environnement proche.

Deuxième partie

Résolution du problème

1 Étapes de résolution du problème

Pour résoudre ce problème de planification, nous avons découpé ce problème en deux sous problèmes

— Premièrement, nous calculons les meilleurs itinéraires entre chacun des points que nous voulons relier pour créer un graphe orienté et complet où chaque nœud est une des adresses à visiter. Chacune des arêtes contient le temps de trajet estimé et le chemin réel à effectuer.

— Dans un second temps, nous utilisons un algorithme de résolution du problème du voyageur de commerce. Ce problème ne peut être résolu exactement dans un temps convenable : c'est un problème np-difficile. On est face à une explosion combinatoire (Avec 20 villes, il existe $6,082 \times 10^{16}$ combinaisons). On est donc face à un problème d'optimisation combinatoire.

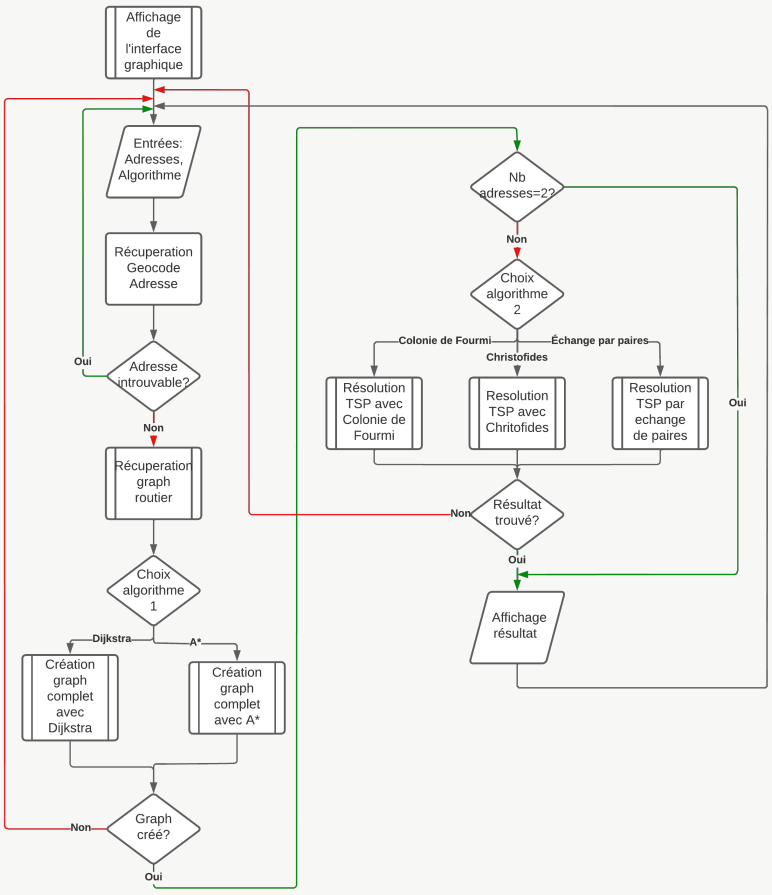


FIGURE 2 – Algorigramme de la résolution du problème

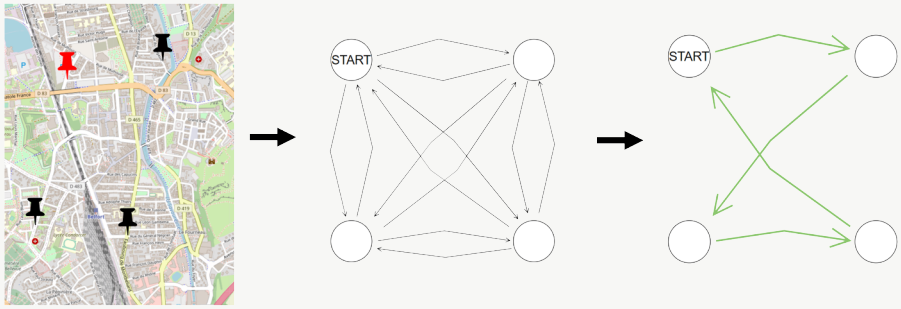


FIGURE 3 – Les différentes étapes de résolution

1.1 Création du graphe orienté et complet

Pour créer le graphe orienté et complet, nous utilisons un algorithme permettant de trouver l'itinéraire le plus court entre deux points pour chaque combinaison de points de départs et d'arriver.

Cette étape est donc de complexité quadratique. Il s'agit probablement de la prochaine piste d'amélioration à étudier pour ce programme. On pourrait imaginer un élagage des possibilités en fonction de leur distance à vol d'oiseau et ne calculer que les itinéraires entre les points les plus proches.

1.1.1 Structure de donnée du graphe simplifié

Après l'exécution de tous les algorithmes, le graphe simplifié est stocké à l'aide d'un dictionnaire de dictionnaire contenant une liste et une valeur de temps. La liste est la liste des nœuds réels à visiter.

Ainsi par exemple pour connaitre le temps de trajet entre le nœud A et le nœud B, on utilise la ligne :

```
time = graphe[A][B]["time"]
```

2 Recherche d'un itinéraire le plus court entre deux points

Pour trouver l'itinéraire le plus court, nous avons utilisé au choix l'algorithme bidirectionnel de Dijkstra ou A*. Il faut noter que l'algorithme de Dijkstra est identique à celui de A* mais avec une heuristique nulle.

Contrairement à l'algorithme A* vu en cours et l'algorithme bidirectionnel de A* explore le graphe en alternance depuis le point de départ et depuis le point d'arrivée.

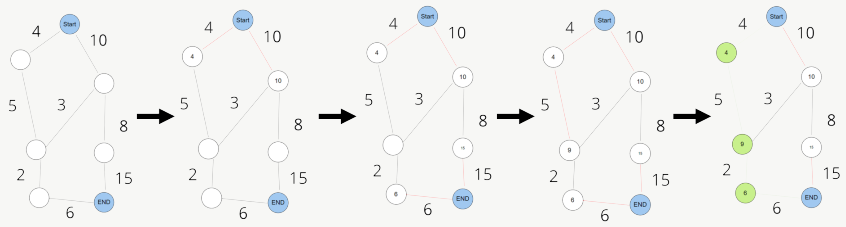


FIGURE 4 – Exemple de resolution d'un Dijkstra bidirectionelle

Dans notre code, cela se traduit par

```
while to_explore[0] and to_explore[1]:
    direction = 1 - direction #Switch direction
    ( _ , dist, v) = pop(to_explore[direction])
```

Nous arrêtons la recherche quand le nœud en cours d'exploration est présent dans les nœuds déjà explorés en sens inverse. Pour stocker la liste des nœuds à visiter par ordre de priorité (temps le plus court pour y accéder) nous utilisons la structure du tas qui permet d'accéder rapidement à l'élément ayant la plus petite valeur.

Pour utiliser cette structure de donnée, nous avons utilisé la bibliothèque heapq. Pour plus d'informations sur les tas, voir ll'annexe A.1

2.1 A* Heuristique

Dans l'algorithme bidirectionnel A*, la fonction heuristique est un composant essentiel qui permet d'orienter la recherche vers le nœud cible en estimant la distance entre le nœud courant et la cible. Lors du choix d'une fonction heuristique, il est important de prendre en compte à la fois la précision de l'estimation et le coût de calcul du calcul de la distance.

Dans l'implémentation proposée, le temps pour parcourir la distance orthodromique à 30 km/h a été choisie comme fonction heuristique sur la distance euclidienne pour plusieurs raisons. Premièrement, elle fournit une bonne estimation de la distance réelle entre deux nœuds à la surface de la Terre.

$$d = R \arccos\left(\sin(\varphi_1)\sin(\varphi_2) + \cos(\varphi_1)\cos(\varphi_2)\cos(\lambda_1 - \lambda_2)\right) \tag{1}$$

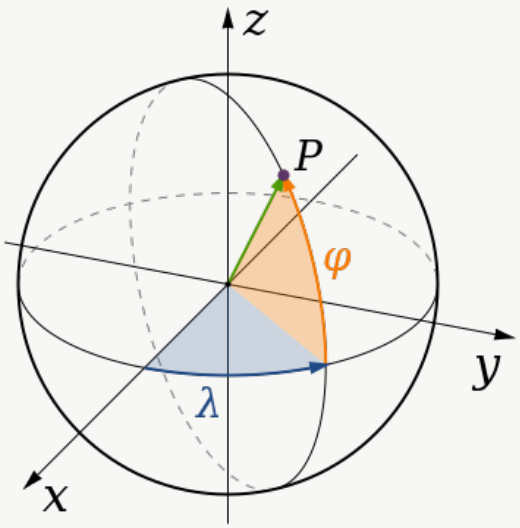


FIGURE 5 – Latitude and longitude coordinates on a sphere

Il est calculé en supposant que la Terre soit une sphère parfaite et en utilisant des formules trigonométriques pour déterminer le chemin le plus court entre les deux points sur la surface de la sphère. Cela la rend généralement plus précise que la distance euclidienne, qui suppose que la Terre est un plan plat et ne tient pas compte de la courbure de la surface de la Terre.

Cependant, la distance orthodromique peut être plus coûteuse en calcul que la distance euclidienne, qui est calculée à l'aide du théorème de Pythagore et nécessite moins de calculs. Malgré cela, la distance orthodromique a toujours été choisie comme fonction heuristique, car sa précision l'emporte sur le coût de calcul supplémentaire dans ce cas.

Dans l'ensemble, la distance orthodromique est un bon choix pour la fonction heuristique dans l'algorithme bidirectionnel A* puisqu'elle fournit une bonne estimation de la distance réelle entre deux nœuds et est relativement facile à calculer, bien qu'elle nécessite plus de calculs que la distance euclidienne.

Ce choix de 30 km/h est cohérent dans le cadre de notre problématique, le livreur opérant dans une ville.

Voici ci-joint l'implémentation en python de la fonction heuristique

```
1     def heuristic(u, v):
2         # Get the latitude and longitude coordinates of the nodes
```

```
        lat1 = Graph.nodes[u]["y"]
        lon1 = Graph.nodes[u]["x"]
        lat2 = Graph.nodes[v]["y"]
        lon2 = Graph.nodes[v]["x"]

        # Convert the coordinates to radians
        lat1 = lat1 * math.pi / 180
        lon1 = lon1 * math.pi / 180
        lat2 = lat2 * math.pi / 180
        lon2 = lon2 * math.pi / 180

        # Calculate the great circle distance between the two points
        a = math.sin((lat2-lat1)/2)**2 + math.cos(lat1) * math.cos(lat2) *
    math.sin((lon2-lon1)/2)**2
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
        d = 6371 * c #Get distance (km)
        t = (d / 30) * 3600 #Convert to seconds at 30 km/h

        return t
```

2.2 Dijkstra bidirectionnel

Pour l'algorithme de Dijkstra, nous utilisons le même que celui de A* mais avec une fonction d'heuristique nulle.

3 Résolution du problème du voyageur de commerce

Le problème du voyageur de commerce peut avoir une complexité exponentielle, ce qui en fait un problème de planification particulièrement complexe à résoudre.

3.1 L'algorithme des fourmis

L'algorithme des fourmis est un algorithme métaheuristique d'optimisation visant à résoudre le problème du voyageur. Cet algorithme est inspiré de la biologie avec des fourmis qui se déplacent le long d'un graphe déposant en des phéromones sur leur passage.

Nous avons donc un objet colonie de fourmis, cet objet est composé d'une liste d'objet fourmis. Chaque fourmi va effectuer un parcours, une fois le parcours effectué, chaque fourmi déposera des phéromones sur le chemin qu'elle a emprunté et influencera ainsi la prochaine génération de fourmis. À chaque génération, l'algorithme mémorise la meilleure fourmi pour renvoyer à la fin le chemin qu'à effectuer la meilleure fourmi. Si la meilleure fourmi n'a pas été amélioré après un certain nombre d'itérations, l'algorithme s'arrête.

Les algorithmes détaillés peuvent être trouvés dans l'annexe A.9

3.1.1 Choix de l'itinéraire pour une fourmi

Lors de la première génération, les fourmis sélectionnent aléatoirement leur chemin, cela permet d'augmenter l'exploration.

Lors des générations suivantes, les fourmis sélectionnent aléatoirement un chemin en utilisant la formule de probabilité suivante récupéré sur Wikipédia :

$$p_{xy}^k = \frac{(\tau_{xy}^\alpha)(\eta_{xy}^\beta)}{\sum_{z \in \text{allowed}_x} (\tau_{xz}^\alpha)(\eta_{xz}^\beta)}$$

Où τ_{xy} est la quantité de phéromones déposée et $\eta_{xy} = 1/temps_{xy}$ avec xy le chemin de x à y. α et β sont des paramètres définis lors de la création de la colonie qui influencent respectivement sur l'importance des phéromones et du temps de trajet sur le trajet xy

η_{xy} étant toujours inférieur à 1, quand on augmente β, l'importance du paramètre associé baisse.

3.1.2 Exemple de choix avec une fourmi

Ici une fourmi arrive à un nœud. L'agent n'est pas omniscient, il sait uniquement quels sont les nœuds adjacents qu'il peut rejoindre, le temps de trajet et les phéromones déposés dessus :

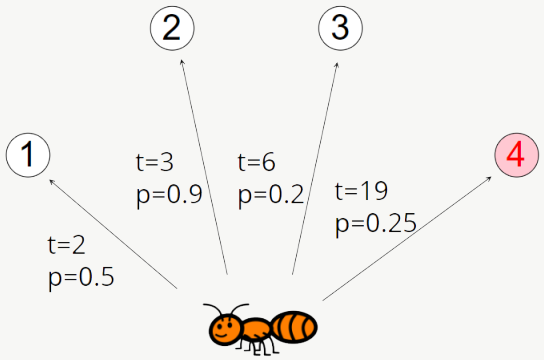


FIGURE 6 – Exemple de résolution de choix d'une fourmi

Sur le schéma ci-dessus, la valeur t correspond au temps de trajet et p la valeur de phéromones. Dans notre exemple, on prend $\alpha = 0.5$ et $\beta = 2$

— Dans un premier temps, la fourmi va attribuer une valeur pour chaque chemin en fonction des phéromones et du temps de trajet estimé :

1. $p' = 0.5^{0.5}(\frac{1}{2})^2 = 0.18$
2. $p' = 0.9^{0.5}(\frac{1}{3})^2 = 0.11$
3. $p' = 0.2^{0.5}(\frac{1}{6})^2 = 0.01$
4. Le numéro 4 a déjà été visité, il n'est donc pas pris en compte.

— Ensuite, on normalise le tout pour que la somme vaille 1 :

$$0.18 + 0.11 + 0.01 = 0.3$$

$$\frac{0.18}{0.3} = 0.6$$

$$\frac{0.11}{0.3} = 0.37$$

$$\frac{0.01}{0.3} = 0.03$$

— Enfin, on tire un nombre aléatoire entre 0 et 1, ici 0.7 :

1. $0.7 > 0.6$ ainsi, la fourmi ne prendra pas ce chemin. On effectue l'opération : $0.7 - 0.6 = 0.1$
2. $0.1 < 0.37$ ainsi la fourmi prendra le chemin 2.

Et cette étape est effectuée en boucle jusqu'à ce que la fourmi ait visité tous les nœuds ou soit bloqué.

3.1.3 Mise à jour des phéromones

Une fois que toutes les fourmis d'une génération ont terminé leur tour, il faut mettre à jour les phéromones. Encore une fois, nous avons utilisé la formule de Wikipédia :

$$\tau_{xy} \leftarrow (1-\rho)\tau_{xy} + \sum_{k}^{m} \Delta\tau_{xy}^{k}$$

Où ρ est le coefficient d'évaporation. Plus il est élevé, plus les fourmis favoriseront l'exploration et $\Delta\tau_{xy}^{k}$ la quantité de phéromones déposées par une fourmi. Dans notre cas, cette valeur est définie par : $\Delta\tau_{xy}^{k} = 1/L_k$ si la fourmi k est passée par le chemin xy, sinon $\Delta\tau_{xy}^{k} = 0$. L_k est le temps estimé de parcours du chemin qu'à effectuer la fourmi k

Lors des tests, nous nous sommes rendus compte que si aucune fourmi ne prend un chemin lors de la première génération, ce chemin ne sera plus exploré. En effet, la valeur de phéromones vaudra 0, et ainsi la probabilité de passer par ce chemin vaudra 0. Pour contourner ce problème, à la fin de la première exploration, la valeur de phéromone de tous les chemins non exploré est égale à $\Delta\tau_{xy}^{k} = 1/max\{L\}$ où L est la liste de tous les temps de parcours des fourmis. Cela revient à considérer que tous les chemins non explorés sont au potentiellement aussi intéressent que le pire chemin effectué lors de la première génération.

3.1.4 Exemple de mise à jour des phéromones

Imaginons un chemin du nœud 4 au nœud 6 ayant une valeur de phéromone de $0,6$ et $\rho = 0.4$

- Il faut commencer par faire s'évaporer les phéromones. $(1-0.4)0.6 = 0.36$
- Maintenant, il faut ajouter les phéromones des fourmis qui sont passées par ici :
  - Cette fourmi à fait le trajet : 1 2 6 5 4 3 en 18 secondes, elle n'est donc pas passée par 4 6. On ne fait rien pour ce chemin
  - Cette fourmi à fait le trajet : 1 5 4 6 3 2 en 5 secondes, elle est donc passée par ce chemin. On met à jour la valeur du chemin 4 à 6 : $0.36 + \frac{1}{5} = 0.61$

Et on répète ces étapes pour chaque chemin.

3.1.5 Paramètres

Lors de nos tests, nous nous sommes rendus compte de l'importance des paramètres. Avec les bons paramètres, nous étions capables d'obtenir de meilleurs résultats en un temps plus faible. Pour voir les résultats de tous les tests effectués, se reporter à l'annexe B.1

Les paramètres choisis sont donc :

Nombre de fourmis	25
α	0.75
β	3
ρ	0.1
ω	50

3.2 L'algorithme de Christofides

3.2.1 Introduction

L'algorithme de Christofides est un algorithme permettant de trouver des approximations de solutions au problème de voyageur de commerce. Il garantit que la solution sera au maximum d'un coût d'un facteur de 3/2 de la solution optimale, et était l'algorithme avec la meilleure approximation pour un temps polynomial (complexité n^2). Un nouvel algorithme, trouvé en 2020 et basé sur celui de Christofides, permet une amélioration à un facteur de $1,5 - 10^{-36}$.

3.2.2 Explication du fonctionnement général de l'algorithme de Christofides

À partir d'un graphe complet G=(V,E) dont les poids respectant l'inégalité triangulaire, l'algorithme peut être décrit à l'aide de 5 grandes étapes :

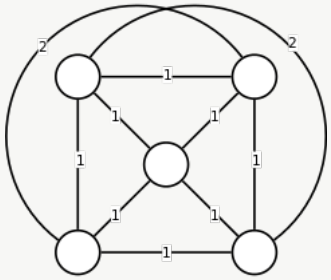


FIGURE 7 – Graphe G=(V,E)

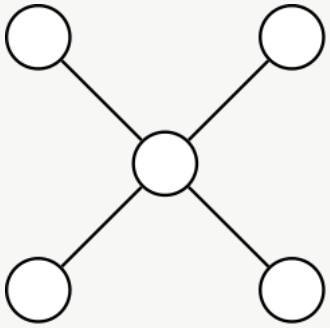


FIGURE 8 – 1. Créer l'arbre couvrant de poids minimal T de G

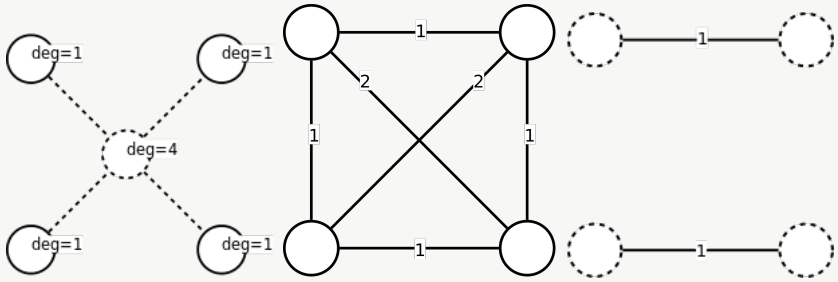


FIGURE 9 – 2. Soit O le set contenant les arêtes de degré impairs de T. Calculer le set de couplages parfait M de poids minimum dans le sous-graphe induit par les sommets de T

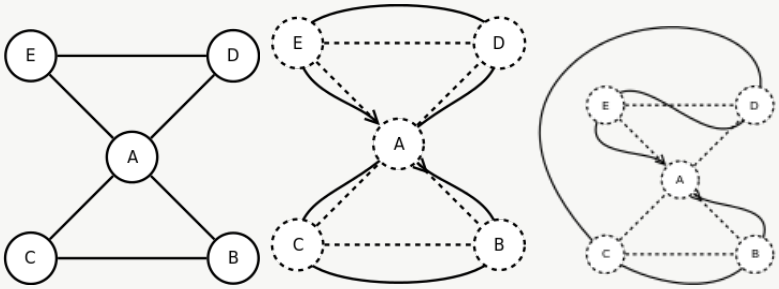


FIGURE 10 – Ajouter les arêtes de M et T à un graphe H (3), calculer le cycle Eulérien dans H (4), calculer le cycle Hamiltonien. (5)

Images venant de : https://fr.wikipedia.org/wiki/Algorithme_de_Christofides?uselang=fr

3.2.3 Explication détaillée de l'algorithme

Nous cherchons à créer un chemin dans lequel chaque sommet est visité une unique fois, soit un cycle Hamiltonien. D'après le théorème de Dirac, un graphe simple avec N sommets ($3 \leq n$) est Hamiltonien si chaque sommet est de degré n/2 ou supérieur. Le degré d'un sommet est simplement le nombre d'arêtes attachées à celui-ci.
Dans notre programme, l'algorithme de Christofides prend en entrée un graphe simple, non orienté, et complet. Un graphe complet à tous ses sommets directement connectés par une arête. Les graphes complets que nous utilisons dans le programme possèdent nécessairement 3 sommets ou plus pour pouvoir être utilisé dans le TSP, et possèdent donc un cycle Hamiltonien.

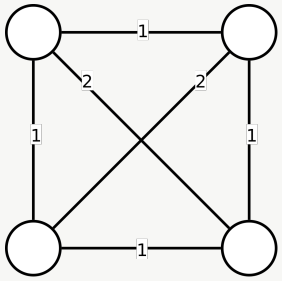


FIGURE 11 – Exemple de graphe complet à 4 arêtes

Nous cherchons donc ce cycle Hamiltonien. Or, un circuit Eulérien possède nécessairement un circuit Hamiltonien : il faut simplement enlever du chemin les arêtes visités à plusieurs reprises.
Pour qu'un graphe possède un circuit Eulérien, il faut que chacun de ses sommets soit de degré pair.
Pour obtenir ce graphe, l'algorithme de Christofides additionne les arêtes produites par deux sous algorithmes :

1. L'arbre couvrant minimal

   Un arbre couvrant minimal[8] est un arbre connectant l'ensemble des sommets d'un graphe à poids non dirigé, sans cycle et possédant un poids total minimum.
   Mais, tous les sommets de cet arbre ne sont pas de degré pair, et nous devons donc corriger cela pour créer le graphe permettant de calculer le cycle Eulérien. Nous prenons donc l'ensemble I des sommets à degré impair de l'arbre.
   Or par le lemme des poignets de main, et I étant un graphe composé uniquement de sommets de *degré* impairs, I possède donc un nombre de *sommets* pair, ce qui est parfait pour l'algorithme suivant.

2. L'algorithme de couplage parfait à poids minimum

   L'algorithme de couplages parfait à poids minimum[9] a besoin et produit, par définition, des couplages, c'est-à-dire des paires de sommets, relié par une seule arête. L'ensemble des sommets sont donc de degré 1. I ayant un nombre pair de sommets, cet algorithme peut être utilisé.

Additionner les arêtes du couplage parfait permet donc d'ajouté un degré aux sommets impairs de l'arbre couvrant, et donc d'obtenir un multigraphe avec uniquement de sommets de degré pair.
L'utilisation des algorithmes à poids minimaux dans les deux cas permet d'obtenir un multigraphe plus optimisé pour chercher un cycle Hamiltonien[10] avec un poids minimal.

3.2.4 Preuve simplifiée expliquant l'utilisation de l'arbre couvrant minimal et des couplages parfaits

Voici une preuve permettant de comprendre l'intérêt d'utiliser l'addition des couplages parfaits et des arêtes à degré impair. Source de la preuve : https://fr.wikipedia.org/wiki/Algorithme_de_Christofides

Soit U le cycle Hamiltonien de poids minimum, et $c(U)$ son poids.

L'arbre couvrant de poids minimum T est de poids inférieur ou égal au poids de U (c'est-à-dire $c(T) \leq c(U)$), car en enlevant une arête au cycle, on obtient un arbre couvrant.

Le couplage trouvé par l'algorithme est de poids inférieur ou égal à $\frac{1}{2}c(U)$. En effet, si on considère le cycle Hamiltonien de poids minimum sur le sous graphe induit par les sommets de degré impair, on a un poids inférieur ou égal à $c(U)$ du fait de l'inégalité triangulaire, et en prenant une arête sur deux de ce cycle (qui est de longueur paire puisque constitué d'un nombre pair de sommets) on obtient un couplage qui est de poids inférieur à la moitié du poids du cycle (si ce n'est pas le cas on peut prendre le complémentaire).

D'où un poids final majoré par $c(T) + \frac{1}{2}c(U) \leq \frac{3}{2}c(U)$: noter que la transformation du cycle Eulérien en cycle Hamiltonien ne peut pas faire croître le poids grâce à l'inégalité triangulaire.

3.2.5 Les algorithmes utilisés

1. Cycle Eulérien : algorithme de Hierholzer
2. Arbre couvrant à poids minimum : algorithme de Prim
3. Couplages parfait à poids minimum
   Nous utilisons une implémentation de la bibliothèque networkx du "Blossom algorithm" de Edmonds, faisant un couplage parfait à poids maximum. Nous inversons simplement les poids avant, permettant l'obtention du couplage parfait à poids minimum.
   Les autres algorithmes utilisés, notamment le shortcutting permettant de trouver le cycle Hamiltonien, ainsi que l'algorithme regroupant les sommets de degré impair, sont simples et n'ont pas de nom.
   Les algorithmes de Hierholzer et de Prim sont détaillés en annexe.

3.3 Pairwise Exchange

3.3.1 Introduction

L'algorithme Pairwise Exchange est un algorithme permettant de trouver de manière aléatoire une solution au problème du voyageur. Il alterne aléatoirement des liaisons n fois et garde le graphe le moins coûteux en distance total.

3.3.2 Graphe de référence

Afin de pouvoir échanger les liaisons, il est important de connaître le poids de toutes les liaisons pour calculer le poids total à chaque échange de liaison. En utilisant l'algorithme de son choix (Dijkstra, A*, ...) pour calculer le poids entre chaque point, on crée un graphe étoile qui servira de référence.

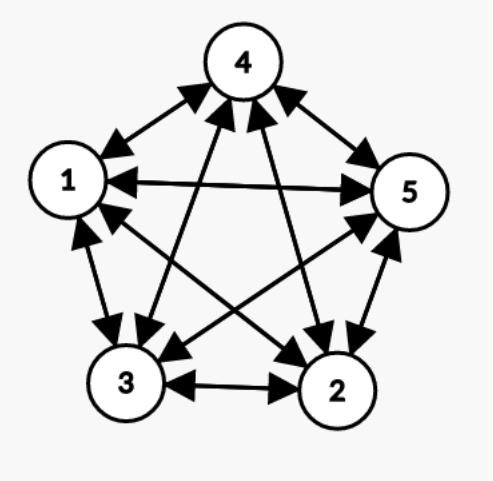
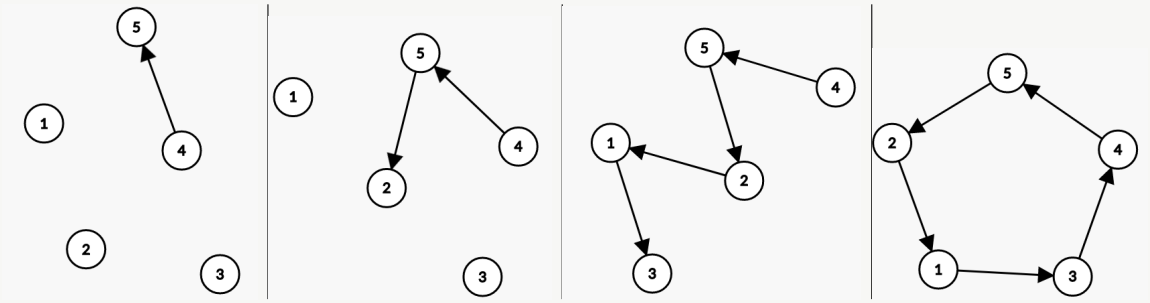


FIGURE 12 – Exemple de graphe complet à 5 noeuds chaque liaisons possède un poids

3.3.3 Créer un graphe lambda

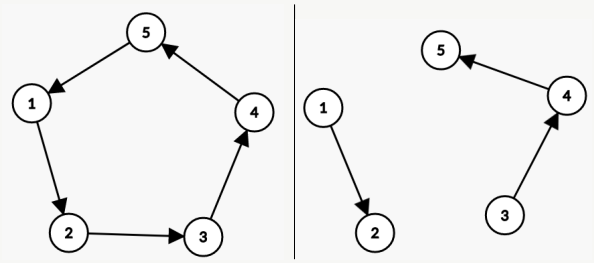
Avant de pouvoir échanger aléatoirement des liaisons, il faut d'abord créer un graphe. Pour cela, on relie un point aléatoire avec le plus proche, puis avec son point le plus proche mis à part ceux déjà reliés.



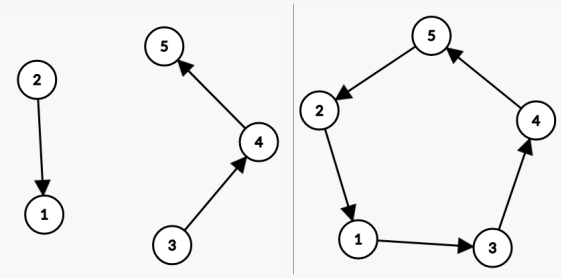
3.3.4 Échanger les liaisons

L'échange de liaisons se fait par récursion. Cela permet qu'une permutation de liaisons qui empire seulement avec 1 échange sera tout de même maintenue pour le 4ème échange qui aura peut-être un poids plus léger que celui d'origine. Une fois que le graphe le plus bas de la récursion a été atteint, on filtre le graphe le plus léger en remontant la récursion.

L'échange de liaisons se fait en 2 temps. On sélectionne d'abord aléatoirement deux liaisons avec aucun point en comment (A->B B->C ne marche par exemple).



Il faut ensuite inverser une de la moitié du graphe (puisque le graphe est directionnel) puis relier à nouveaux les deux moitiés avec les nouvelles liaisons.



Les inversions de moitié de graphe et les ajouts des nouvelles liaisons se font à l'aide du graphe de référence qui sert à indiquer les nouveaux poids des nouvelles liaisons.

4 Limites

Lors de la conception du projet, nous avons observé des limitations bridant considérablement la performance de notre projet.

1. **L'obtention des données via l'API** : demandant parfois beaucoup de données à OpenStreetMap pour construire notre Graphe, nous sommes limités sur notre portée à cause de timeouts de 180s. Effectuer des trajets sur de longues distances est actuellement excessivement cher.
2. **Python** : Bien qu'étant crossplatform sur les ordinateurs, notre projet est limité en vitesse par le langage imposé. Nos efforts de parallélisation n'ont pas été satisfaisants.
3. **PyQT** : Malgré une meilleure modularité que tkinter pour créer des UI sur des projets Python, nous avons été bloqués par les possibilités de customisation via CSS, sachant que notre carte est affichée par un script Javascript dans une fenêtre Chromium. D'autres Frameworks pourraient nous offrir une meilleure flexibilité pour un usage commercial.

Troisième partie

Conclusion

Au cours de ce projet, nous avons développé un système de navigation qui utilise une variété d'algorithmes pour aider les utilisateurs à trouver le chemin le plus court.

L'un des principaux défis auxquels nous avons été confrontés lors du développement de ce système a été de trouver les algorithmes les plus efficaces à utiliser. Nous avons expérimenté plusieurs algorithmes différents, y compris ceux que nous avons vus en classe et de nouveaux que nous avons implémenté nous-mêmes.

Nous avons constaté que **la combinaison des algorithmes A*, Dijkstra, avec un algorithme des Fourmis et Christofides était la solution la plus efficace** pour trouver le chemin le plus court entre plusieurs emplacements, et nous l'avons implémenté comme composant central de notre système de navigation.

Un autre défi auquel nous avons été confrontés a été de **coordonner nos efforts en tant que groupe**. Nous avons dû travailler ensemble pour répartir les tâches et communiquer efficacement pour nous assurer que le projet soit terminé à temps. Malgré ces défis, nous avons réussi à mener à bien le projet et nous estimons que nos compétences en coordination de groupe se sont améliorées en conséquence.

Ce projet n'était pas seulement éducatif, mais aussi intéressant et potentiellement précieux en tant que produit commercial. Alors que nous continuons à développer nos compétences et nos connaissances en informatique, nous pourrons peut-être revenir sur ce projet et l'**affiner davantage en un produit qui pourrait être mis à la disposition du grand public**.

Dans l'ensemble, nous sommes satisfaits des résultats de ce projet et pensons que notre système de navigation a le potentiel d'être un outil utile pour les entreprises de livraison. Notre intérêt pour le problème, combiné à une meilleure coordination de groupe, **a contribué au succès de ce projet.**

Quatrième partie
Annexe

A Algorithmes

A.1 Heap

Un heap, appelé tas en français, est une structure de données qui permet de stocker des éléments par ordre de priorité. Il s'agit d'un arbre binaire complet. Le nœud parent est toujours prioritaire au nœud fils, ainsi le nœud prioritaire sera toujours à la racine. Dans la mémoire de l'ordinateur, un tas est représenté par une liste. Deux opérations nous intéressent : heap_push pour ajouter un élément dans le tas, et heap_pop pour récupérer l'élément prioritaire.

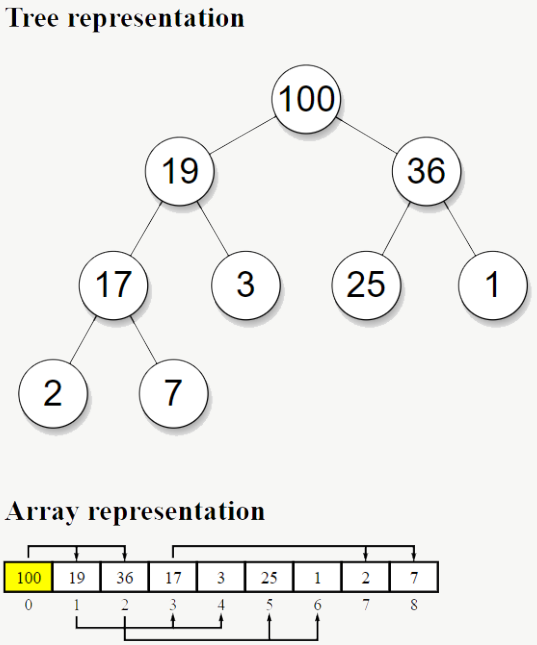


FIGURE 13 – Représentation d'un tas. CC BY-SA 3.0 (Wikipedia, Ermishin)

L'avantage de cette structure est sa rapidité. Dans une liste classique, pour trouver l'élément le plus petit, on doit parcourir la liste en entier, ce qui implique une complexité linéaire. Alors que pour un tas, la valeur prioritaire est toujours à la racine. Une fois récupéré, on remet la dernière valeur à la place de la racine et on tamise le tas. Tamiser le tas permet de déplacer la nouvelle valeur pour respecter la condition : "Le nœud parent est toujours prioritaire au nœud fils". La complexité de cette opération est logarithmique, ce qui est particulièrement intéressant.

Rajouter un élément dans la liste est aussi de complexité logarithmique.

A.1.1 heap_push

```
1 Fonction heap_push (heap : H, element: E) -> heap
2
3 ajouter element en derniere position de H
4 H.taille = H.taille + 1
5
6 Indice = H.taille - 1
7 IndiceParent = (Indice - 1) // 2
8
9 Tant que Indice != 0 et H[Indice] < H[IndiceParent]
```

```
10
11      Echanger H[Indice] et H[IndiceParent]
12      Indice = IndiceParent
13      IndiceParent = (Indice - 1) // 2
```

A.1.2 heap_pop

```
1  Fonction heap_pop (heap: H) -> element
2
3  elem = H[0]
4  H[0] = H[H.taille]
5  H.taille = H.taille - 1
6
7  Indice = 0
8  IndiceFeuille = 1
9
10 Tant que H[Indice] > H[IndiceFeuille] et H[Indice] > H[IndiceFeuille+1] et
       Indice < H.taille
11
12      Si H[IndiceFeuille] < H[IndiceFeuille+1]
13          Echanger H[Indice] et H[IndiceFeuille]
14          Indice = IndiceFeuille
15      Sinon
16          Echanger H[Indice] et H[IndiceFeuille + 1]
17          Indice = IndiceFeuille + 1
18
19      IndiceFeuille = (Indice * 2) + 1
```

A.2 A* bidirectionel

Pour faire fonctionner A* en bidirectionnel, toutes les structures sont stockées dans des duets. Pour accéder aux données, nous utilisons une variable direction que nous faisons alterner entre la valeur 0 et 1.

```
1  Fonction A* bidirectionnel (Graphe : G, Noeud : source, Noeud : destination)
       -> Temps, Chemin
2
3  Si source = destination
4       retourne 0, source
5
6  destination_heurisitc = CreerDuet(destination, source)
7
8  verifier = CreerDuetDeDictionnaire()
9  chemins = CreerDuetDeDictionnaire()
10 vu = CreerDuetDeDictionnaire((souce ,0);(destination, 0))
11 a_visiter = CreerDuetDeTas()
12
13 heap_push(avisiter[0], (heuristic(source, destination),0,source)
14 heap_push(avisiter[1], (heuristic(destination, souce),0,destination)
15
16 chemin = ajouterchemin(0,source,chemins)
17 chemin = ajouterchemin(1,destination,chemins)
18
19 Tant que a_visiter non vide
20
21      direction = 1 - direction
22
23      distance , noeud = pop(a_visiter[direction])
24
25      Si noed dans verifier[direction]
```

```
26          continuer
27
28      verifier = ajouterverifier(direction,noed,distance,verifier)
29
30      Si noeud est dans verifier[ 1- direction]
31
32          retourne RecupererDistance(noeud, verifier), RecupererChemin(noeud,
    chemins)
33
34      Pour voisin dans ListVoisins(noeud)
35
36          Si voisin pas dans verifier[direction]
37
38              poids = DureeTrajet(noeud, voisin, direction, graph)
39
40              si voisin pas dans vu alors
41                  vu = AjouterAVu(noeud, poids, vu)
42                  heap_push(avisiter[direction], (heurisitc(voisin,
    destination_heurisitc[direction]), poids, noeud)
43                  chemin = AjouterChemin(direction, RecupererChemin(noeud,
    chemins) + voisin, chemins)
44
45              sinon si poids < ResupererTemps(direction, noeud, vu)
46                  vu = AjouterAVu(noeud, poids, vu)
47                  heap_push(avisiter[direction], (heurisitc(voisin,
    destination_heurisitc[direction]), poids, noeud)
48                  chemin = AjouterChemin(direction, RecupererChemin(noeud,
    chemins) + voisin, chemins)
49              FinSi
50      Fin pour
51      retoune "Pas de chemin"
```

A.3 Algorithme des Fourmis

A.3.1 AlgorithmeDesFourmis

```
1 Fonction AlgorithmeDesFourmis( nombre: alpha, beta, rho; entier: omega,
    nb_fourmis; graph : graph; debut : noeud) -> chemin
2
3 nb_iter=0
4 1erPassage = Vrai
5 meilleur_duree = infini
6 meilleur_chemin = []
7
8 Tant que nb_iter < omega
9     Pour i dans nb_fourmis
10        chemin[i] = RechercheFourmis(alpha, beta, graph, debut, 1erPassage)
11    Si min(chemin.duree) < meilleur_duree alors
12        meilleur_duree = min(chemin.duree)
13        meilleur_chemin = min(chemin.chemin)
14        omega = 0
15    Sinon
16        omega = omega +1
17
18    MettreAJourPheromones(graph, chemin, rho, 1erPassage)
19    1erPassage = Faux
20 Fin Tant que
21
22 retourne meilleur_chemin
```

A.4 RechercheFourmis

```
Fonction RechercheFourmis( nombre: alpha, beta, passage; graph: graph; noeud: debut; bool: 1erPassage) -> chemin, nombre

AVisiter = RecupererNoed( graph )
Actuel = debut
Chemin = debut
Duree = 0

Tant que AVisiter non vide

    Si 1erPassage alors
        Pour Noeud dans AVisiter
            Proba[Noeud] = 1/taille(AVisiter)
        Fin Pour
    Sinon
        Somme = 0
        Pour Noeud dans AVisiter
            Proba[Noeud] = pheromone(Actuel , Noeud) ^ alpha *  distance(Actuel , Noeud) ^ Beta
            Somme = Proba[Noeud]
        Fin Pour
        Pour proba dans Proba[Noeud]
            proba = proba/Somme
        Fin Pour
    Fin Si

    Suivant = ChoixAleatoire(AVisiter, Proba)
    Duree = Duree + distance(Actuel , Suivant)
    Chemin = Ajouter(Chemin , Suivant)
    Actuel = Suivant
Fin Tant que
retourne Chemin, Duree
```

A.5 MettreAJourPheromones

```
Fonction MettreAJourPheromones( graph : graph; liste de chemin: chemins; nombre: rho, bool: PremierPassage)
Pour liaison dans graph
    Si PremierPassage:
        liaison[pheromones] = 1/max(chemins.duree)
    Sinon
        liaison[pheromones] = liaison[pheromones] * (1 - rho)
    Fin si
Fin Pour
Pour chemin dans chemins
    Pour liaison dans chemin
        graph[liaison][pheromones] = graph[liaison][pheromones] + 1 / chemin.duree
```

A.6 Algorithme de pairwise exchange

L'algorithme Pairwise Exchange est un algorithme permettant de trouver de manière aléatoire une solution au problème du voyageur. Il alterne aléatoirement des liaisons n fois et garde le graphe le moins coûteux en distance total.

```
Fonction PairwiseExchange(full_graph: graph_reference, ring_graph: graph, entier: nb_recursion)-> ring_graph
```

```
    graph1, graph2 = supprimer_2liaisons(graph)
    graph1 = inverser(graph1)
    graph = connecter(graph1, graph2)
    graph = mettre_poid_a_jour(graph, graph_reference)

    Si nb_recursion = 1 alors
        retourne graphe
    Sinon
        graphe' = PairwiseExchange(graph, nb_recursion - 1)
    Fin Si

    Si poids(graphe) < poids(graphe') alors
        retourne graphe
    Sinon
        retourne graphe'
    Fin si
```

A.7 Algorithme de Christofides

```
Fonction Christofides(Graphe non dirige: G) -> liste: Circuit Hamiltonien
    arbre = algorithme_de_Prim(G)
    graphe_impaire = copie(G)
    Pour sommet dans arbre:
      Si degree(sommet) non impaire:
        graphe_impaire.enlever(sommet)
    pairage_minimaux = pairage_minimaux(graphe_impaire)
    graphe_eulerien = pairage_minimaux + arbre
    circuit_eulerien = algorithme_de_hierholzer(graphe_eulerien)
    circuit_hamiltonien = [circuit_eulerien.premier()]
    Pour sommet dans circuit_eulerien:
      si sommet dans non dans circuit_hamiltonien:
        circuit_hamiltonien.ajouter(sommet)
    retourner circuit_hamiltonien
```

A.8 Algorithme de Prim

L'algorithme de Prim est un algorithme permettant de trouver des arbres couvrants à poids minimal dans un graphe à poids non dirigé. Cette version de l'algorithme de Prim utilise un tas pour obtenir des temps logarithmique et non linéaire.

```
Fonction algorithme_de_Prim(Graphe Non Dirige : G) -> Graphe : Arbre

arbre = Graphe vide
noeuds = Graphe.noeuds()

Tant que G non vide:
    premier_sommet =  G.sommet_de_depart
    frontiere = tas
    visite = [premier_sommet]
    Pour deuxieme_sommet dans premier_sommet.sommet_adjacent():
        pousser_tas(tas, (distance, premier_sommet, deuxieme_sommet))
    Tant que G non vide et tas non vide:
        distance, premier_sommet, deuxieme_sommet = pop(tas)
        Si deuxieme_sommet non dans visite:
            arbre.ajouter_arete(distance, premier_sommet, deuxieme_sommet)
            visite.ajouter(deuxieme_sommet)
            G.enlever(deuxieme_sommet)
            Pour troisieme_sommet, poids dans deuxieme_sommet.sommet_adjacent
    ():
```

```
                Si troisieme_sommet dans visite:
                    continuer
                Sinon
                    pousser_tas(tas, (poids, troisieme_sommet)
retourner arbre
```

A.9 Algorithme de Hierholzer

```
 Fonction algorithme_de_hierholzer( Graphe Eulerien : G) -> chemin :
Circuit Eulerien
   chemin = [] # liste pour stocker le chemin eulerien
   pile = [] # pile pour suivre le chemin en cours
   sommet_actuel = G.sommet_de_depart # on commence au sommet de depart

   tant que Vrai :
     # s'il y a des aretes sortantes du sommet actuel
     si sommet_actuel a des voisins :
       # on met le sommet actuel dans la pile
       pile.ajouter(sommet_actuel)
       # on choisit une arete sortante et on la suit
       prochain_sommet = choisir_arete_sortante(sommet_actuel)
       enlever_arete(sommet_actuel, prochain_sommet) # on retire l'arete du
 graphe
       sommet_actuel = prochain_sommet # on se deplace au prochain sommet
     sinon :
       # s'il n'y a plus d'aretes sortantes, on ajoute le sommet actuel au
chemin
       chemin.ajouter(sommet_actuel)
       # si la pile est vide, c'est qu'on a trouve un chemin eulerien
       si pile est vide :
         retourner chemin
       # si la pile n'est pas vide, on revient au sommet precedent
       sommet_actuel = pile.pop()
```

B Test des paramètres

B.1 Algorithme des fourmis

Les tests suivants ont été effectués sur un Intel I7-7200U avec Windows 11. Chacun des résultats est la moyenne obtenue lors de 5 exécutions.

α	β	ρ	Nombre de fourmis	ω	Temps de trajet	Temps d'exécution
0.25	2	0.25	25	25	6382.62	0.5334017276763916
0.25	2	0.5	75	25	6199.9	1.4918540477752686
0.25	2	0.5	75	50	6106.44	2.9183793544769285
0.25	2	0.5	75	75	5992.44	4.780342245101929
0.25	2	0.75	25	25	6087.2	0.4921590328216553
0.25	2	0.75	25	50	6051.26	0.9841760158538818
0.25	2	0.75	25	75	6018.680000000001	1.5047954082489015
0.25	2	0.75	50	75	5829.359999999999	2.4526071071624758
0.25	2	0.75	75	25	6223.280000000001	1.542613649368286
0.25	2	0.75	75	50	6171.1	2.140722227096558
0.25	2	0.75	75	75	5896.22	3.4297035694122315
0.25	3	0.25	25	25	6043.539999999999	0.5230113983154296
0.25	3	0.25	25	50	5706.82	0.8594446659088135
0.25	3	0.25	25	75	5842.64	1.3631756782531739
0.25	3	0.25	50	25	5921.160000000001	0.7332822322845459
0.25	3	0.25	50	50	5590.26	1.859013319015503
0.25	3	0.25	50	75	5776.280000000001	3.084506559371948
0.25	3	0.25	75	25	5821.0199999999995	1.3606409072875976
0.25	3	0.25	75	50	5774.460000000001	2.8647966384887695
0.25	3	0.25	75	75	5742.1	4.19717321395874
0.25	3	0.5	25	25	5810.359999999999	0.5172399520874024
0.25	3	0.5	25	50	5771.160000000001	1.2678606033325195
0.25	3	0.5	25	75	5835.7	1.6471052169799805
0.25	3	0.5	75	75	5755.360000000001	4.45426378250122
0.25	3	0.75	25	25	5792.5	0.5613523006439209
0.25	3	0.75	25	50	5818.680000000001	1.0124485969543457
0.25	3	0.75	25	75	5723.740000000001	1.4232296466827392
0.25	3	0.75	50	25	5863.820000000001	0.7858451843261719
0.25	3	0.75	50	50	5634.420000000001	1.477259635925293
0.25	3	0.75	50	75	5654.800000000001	2.369653034210205
0.25	3	0.75	75	25	5734.38	1.3544184684753418
0.5	2	0.75	75	25	5859.9	1.5048388481140136
0.5	2	0.75	75	50	5659.379999999999	2.5201520919799805
0.5	2	0.75	75	75	5721.660000000001	3.6955799579620363
0.5	3	0.25	25	25	5621.680000000001	0.5109799861907959
0.5	3	0.25	25	50	5573.42	1.0034493923187255
0.5	3	0.25	25	75	5486.58	1.3198012351989745

α	β	ρ	Nombre de fourmis	ω	Temps de trajet	Temps d'exécution
0.5	3	0.5	25	25	5537.82	0.5799487590789795
0.5	3	0.5	25	50	5640.24	0.9870368003845215
0.5	3	0.5	25	75	5563.820000000001	1.5986988544464111
0.5	3	0.5	50	25	5503.219999999999	0.9373448371887207
0.5	3	0.5	50	50	5474.040000000001	1.6333470821380616
0.5	3	0.5	50	75	5472.960000000001	1.8242981910705567
0.5	3	0.5	75	25	5564.780000000001	1.2515198707580566
0.5	3	0.5	75	50	5519.620000000001	2.6544025421142576
0.5	3	0.5	75	75	5487.180000000001	4.193868494033813
0.5	3	0.75	25	25	5572.360000000001	0.48598346710205076
0.5	3	0.75	25	50	5523.040000000001	0.8150070667266845
0.5	3	0.75	25	75	5466.560000000001	1.2977030754089356
0.5	3	0.75	50	25	5657.6	0.9466145038604736
0.5	3	0.75	50	50	5547.08	1.9342488765716552
0.5	3	0.75	50	75	5540.14	2.4100293636322023
0.5	3	0.75	75	25	5542.660000000001	1.3365312099456788
0.5	3	0.75	75	50	5619.280000000001	3.0334658145904543
0.5	3	0.75	75	75	5543.340000000001	3.0532582283020018
0.75	2	0.25	25	25	5531.480000000001	0.5820566177368164
0.75	2	0.25	25	50	5553.74	1.1094482421875
0.75	2	0.25	25	75	5511.36	1.122687292098999
0.75	2	0.25	50	25	5613.660000000001	1.1091316223144532
0.75	2	0.25	50	50	5530.2	1.9099299430847168
0.75	2	0.25	50	75	5502.74	2.74507360458374
0.75	2	0.25	75	25	5626.640000000001	1.642433500289917
0.75	2	0.25	75	50	5588.240000000001	3.156670618057251
0.75	2	0.25	75	75	5473.88	3.799357843399048
0.75	2	0.5	25	25	5651.82	0.6330392360687256
0.75	2	0.5	25	50	5463.72	0.9846024990081788
0.75	2	0.5	25	75	5598.48	1.1780989646911622
0.75	2	0.5	50	25	5480.4400000000005	0.8058325767517089
0.75	2	0.5	50	50	5461.08	1.5482290744781495
0.75	2	0.5	50	75	5489.099999999999	2.2821822643280028
0.75	2	0.5	75	25	5481.5599999999995	1.6614581108093263
0.75	2	0.5	75	50	5531.24	3.0279126167297363
0.75	2	0.5	75	75	5515.800000000001	3.9432828426361084
0.75	2	0.75	25	25	5545.6	0.47348713874816895
0.75	2	0.75	25	50	5522.68	0.5828664779663086
0.75	2	0.75	25	75	5577.52	1.1098317623138427
0.75	2	0.75	50	25	5569.0199999999995	0.868532657623291
0.75	2	0.75	50	50	5477.540000000001	1.7741246700286866
0.75	2	0.75	50	75	5478.960000000001	2.278688335418701
0.75	2	0.75	75	25	5456.220000000001	1.4076149940490723
0.75	2	0.75	75	50	5487.18	2.3571809768676757
0.75	2	0.75	75	75	5489.280000000001	3.736506462097168

α	β	ρ	Nombre de fourmis	ω	Temps de trajet	Temps d'exécution
0.75	3	0.25	25	25	5668.5599999999995	0.4293826580047607
0.75	3	0.25	25	50	5542.06	0.6550972938537598
0.75	3	0.25	25	75	5544.16	1.1598420143127441
0.75	3	0.25	50	25	5522.6	0.7774882316589355
0.75	3	0.25	50	50	5655.120000000001	1.3602131843566894
0.75	3	0.25	50	75	5543.12	1.9375534534454346
0.75	3	0.25	75	25	5599.660000000001	1.4794768333435058
0.75	3	0.25	75	50	5463.14	1.99662446975708
0.75	3	0.25	75	75	5514.5599999999995	3.755352592468262
0.75	3	0.5	25	25	5693.700000000001	0.33858366012573243
0.75	3	0.5	25	50	5606.32	0.7887917518615722
0.75	3	0.5	25	75	5566.2	0.7877325534820556
0.75	3	0.5	50	25	5568.0	0.8401674747467041
0.75	3	0.5	50	50	5500.160000000002	1.3885233879089356
0.75	3	0.5	50	75	5577.88	1.9434394359588623
0.75	3	0.5	75	25	5580.620000000001	1.084169578552246
0.75	3	0.5	75	50	5466.56	2.003276062011719
0.75	3	0.5	75	75	5470.68	3.3793604373931885
0.75	3	0.75	25	25	5729.820000000001	0.34229369163513185
0.75	3	0.75	25	50	5696.44	0.5005423545837402
0.75	3	0.75	25	75	5709.16	0.7992749214172363
0.75	3	0.75	50	25	5469.4	0.676978588104248
0.75	3	0.75	50	50	5622.5	1.517338466644287
0.75	3	0.75	50	75	5517.4	1.5078088760375976
0.75	3	0.75	75	25	5599.22	0.955992603302002
0.75	3	0.75	75	50	5534.0199999999995	1.996931219100952
0.75	3	0.75	75	75	5581.24	2.4667196750640867

C Listing :

Voici tout le code python utilisé trié par fichier :

C.1 main.py

```
import sys
import os

from PyQt5.QtWidgets import *
from PyQt5.QtWebEngineWidgets import *
from PyQt5.QtCore import *
import folium
from folium.features import DivIcon
from folium import Marker, Icon
import osmnx as ox
import geoip2.database
import requests
import time

import graph_tools.TSP_solver as tsp_solver


class Form(QWidget):
    def __init__(self):
```

```
        super().__init__()
        self.preload()
        self.initUI()

    def preload(self):
        start_time = time.time()

        ox.settings.log_console = False
        ox.settings.use_cache = True

        # Make an HTTP request to the httpbin.org website
        response = requests.get('http://httpbin.org/ip')

        # Get the user's IP address from the response
        user_ip = response.json()['origin']

        # Open the GeoIP2 database file
        reader = geoip2.database.Reader('testtools/dataset/GeoLite2-City.mmdb'
    )

        # Look up the user's IP address
        response = reader.city(user_ip)

        # Get the user's latitude and longitude coordinates
        latitude = response.location.latitude
        longitude = response.location.longitude

        # Use OSMnx to get the nearest network to the user's location
        G = ox.graph_from_point((latitude, longitude),
                                dist=1000, network_type='drive')
        G = ox.add_edge_speeds(G)
        # calculate travel time (seconds) for all edges
        G = ox.add_edge_travel_times(G)
        end_time = time.time()

        # Use folium to plot the map
        m = folium.Map(location=[latitude, longitude], zoom_start=15)

        # Add the map to the web page
        m.save('route.html')

        # Calculate the elapsed time
        elapsed_time = end_time - start_time
        # Print the elapsed time
        print(f'Elapsed time: {elapsed_time:.2f} seconds')

    def initUI(self):
        self.setSizePolicy(QSizePolicy.Expanding, QSizePolicy.Expanding)
        self.setWindowTitle("Map Viewer")
        # Create a URL pointing to the CSS file
        css_url = QUrl.fromLocalFile("style.css")
        # Create a QFile object to read the CSS file
        css_file = QFile(css_url.toLocalFile())
        # Open the file
        css_file.open(QFile.ReadOnly | QFile.Text)
        # Read the file
        css = QTextStream(css_file).readAll()
        # Close the file
        css_file.close()        # Set the stylesheet for the form
        self.setStyleSheet(css)
        # Create the input fields
        self.input1 = QLineEdit()
```

```
81          self.input1.setPlaceholderText("Start location")
82          self.input2 = QLineEdit()
83          self.input2.setPlaceholderText("End location")
84
85          # Create the drop-down menu
86          self.algorithmComboBox1 = QComboBox()
87          self.algorithmComboBox1.addItem("A*")
88          self.algorithmComboBox1.addItem("Dijkstra")
89
90          self.algorithmComboBox2 = QComboBox()
91          self.algorithmComboBox2.addItem("Ant Algorithm")
92          self.algorithmComboBox2.addItem("Christofides")
93          self.algorithmComboBox2.addItem("Pairwise exchange")
94
95          # Create the button and connect it to the handleButtonClick() method
96          self.button = QPushButton("Submit")
97          self.button.clicked.connect(self.handleButtonClick)
98
99          # Create the HTML preview widget
100         self.preview = QWebEngineView()
101         dirname = os.path.dirname(__file__)
102         filename = os.path.join(dirname, 'route.html')
103         url = QUrl.fromLocalFile(filename)
104         self.preview.load(url)
105         self.inputs = []
106         # Add the widget to the list of inputs
107         self.inputs.append(self.input1)
108
109         # Add the widget to the list of inputs
110         self.inputs.append(self.input2)
111
112         # Create a QPushButton widget
113         self.button1 = QPushButton("+")
114
115         # Connect the clicked signal of the button to a slot
116         self.button1.clicked.connect(self.add_input)
117
118         # Create the layout and add the widgets to it
119         self.playout = QVBoxLayout()
120         self.playout2 = QVBoxLayout()
121         # Add self.playout2 to the main layout
122         self.playout.addLayout(self.playout2)
123         self.playout2.addWidget(self.input1)
124         self.playout2.addWidget(self.input2)
125         self.playout.addWidget(self.button1)
126         self.playout.addWidget(self.algorithmComboBox1)
127         self.playout.addWidget(self.algorithmComboBox2)
128         self.playout.addWidget(self.button)
129         self.playout.addWidget(self.preview)
130         self.setLayout(self.playout)
131
132     def add_input(self):
133         # Create a new QLineEdit widget
134         self.input = QLineEdit()
135
136         # Set the placeholder text for the widget
137         self.input.setPlaceholderText("New input")
138
139         # Add the widget to the list of inputs
140         self.inputs.append(self.input)
141
142         # Add the widget to the layout
```

```
143         self.playout2.addWidget(self.input)
144 
145     def get_inputs(self):
146         # Create an empty list to store the text entered in the inputs
147         inputs_list = []
148         # Use a for loop to add the text entered in each input to the list
149         for input in self.inputs:
150             # Check if the input is empty
151             if input.text():
152                 # Add the text entered in the input to the list
153                 inputs_list.append(input.text())
154         # Move the second input to the end of the list
155         inputs_list.append(inputs_list.pop(1))
156         # Print the list
157         print(inputs_list)
158         # Return the list
159         return inputs_list
160 
161     #The function that will be called when the button is clicked
162     def handleButtonClick(self):
163         # Get the input from the fields
164 
165         input_list = self.get_inputs()
166         #Line used to debug quickly
167         #input_list = ['Belfort, France', 'Botans, France', 'andelnans, France
    ', 'Danjoutin, France', 'Sevenans, France','Territoire de Belfort, Perouse
    ','Moval, France','Urcerey, France','Essert, France, Territoire de Belfort
    ', 'Bavilliers','Cravanche','Vezelois','Meroux','Dorans','Bessoncourt','
    Denney','Valdoie","Ch vremont, Territoire de Belfort, France","Fontenelle,
     Territoire de Belfort, France","Sermamagny, Territoire de Belfort, France
    ","Eloie, Territoire de Belfort, France"]
168 
169         ox.settings.log_console = True
170         ox.settings.use_cache = True
171 
172         # Call the construct_graph method, passing the start and end locations
     as arguments
173         try:
174             graph, route, time, geocode_list = tsp_solver.main_solver(
175                 input_list, name_algorithm1=self.algorithmComboBox1.
    currentText(), name_algorithm2=self.algorithmComboBox2.currentText())
176             print("The time to travel the route is: ", time, " seconds")
177             # Create a QLabel widget to display the time
178             # divide time by 3600 to get the number of hours
179             hours, seconds = divmod(time, 3600)
180             # divide the remainder by 60 to get the number of minutes
181             minutes, seconds = divmod(seconds, 60)
182 
183             # Convert the hours, minutes, and seconds to strings and add
    leading zeros if necessary
184             hours_string = str(int(hours)).zfill(2)
185             minutes_string = str(int(minutes)).zfill(2)
186             seconds_string = str(int(seconds)).zfill(2)
187             # Create a string in the format "hours:minutes:seconds"
188             time_string = f"{hours_string}:{minutes_string}:{seconds_string}
    to drive"
189             # Check if the time_label widget already exists
190             if hasattr(self, 'time_label'):
191                 # If the time_label widget already exists, set the text of the
     widget to the updated time
192                 self.time_label.setText(time_string)
193             else:
```

```
194                 # If the time_label widget does not exist, create a new QLabel
     widget to display the time
195                 self.time_label = QLabel()
196                 # Set the text of the time_label to the time string
197                 self.time_label.setText(time_string)
198                 # Add the time_label to the layout
199                 self.playout.addWidget(self.time_label, 0, Qt.AlignRight)
200 
201         except ValueError as err:
202             #print an msgbox if there is a problem with the input
203             msg = QMessageBox()
204             msg.setIcon(QMessageBox.Critical)
205             msg.setText(err.args[0])
206             msg.setWindowTitle("Error")
207             msg.setStandardButtons(QMessageBox.Ok)
208             msg.exec_()
209             return
210 
211         except ConnectionError:
212             #print an msgbox if there is a problem with the connection
213             msg = QMessageBox()
214             msg.setIcon(QMessageBox.Critical)
215             msg.setText("Connection error, please try again")
216             msg.setWindowTitle("Error")
217             msg.setStandardButtons(QMessageBox.Ok)
218             msg.exec_()
219             return
220 
221         except:
222             #print an msgbox if the route is not possible
223             msg = QMessageBox()
224             msg.setIcon(QMessageBox.Critical)
225             msg.setText("Unknow error, please try again")
226             msg.setWindowTitle("Error")
227             msg.setStandardButtons(QMessageBox.Ok)
228             msg.exec_()
229             return
230 
231         # Plot the route on a map and save it as an HTML file
232         route_map = ox.plot_route_folium(
233             graph, route, tiles='openstreetmap', route_color="red",
    route_width=10)
234 
235         # Create a Marker object for the start location
236         start_latlng = (float(geocode_list[0][1]), float(geocode_list[0][0]))
237         start_marker = Marker(location=(
238             start_latlng[::-1]), popup='Start Location', icon=Icon(icon='
    glyphicon-flag', color='green'))
239 
240         # Add the start and end markers to the route_map
241         start_marker.add_to(route_map)
242 
243         # Create a Marker object for each location in the route
244         for i in range(1, len(geocode_list)-1):
245             latlng = (float(geocode_list[i][1]), float(geocode_list[i][0]))
246             # create a Marker object for the location containing a number icon
247             marker = Marker(location=(
248                 latlng[::-1]), popup='Location', icon=Icon(icon='glyphicon-
    flag', color='blue'))
249             marker = Marker(location=(latlng[::-1]), popup='Location', icon=
    DivIcon(icon_size=(
```

```
                150, 36), icon_anchor=(7, 20), html='<div style="font-size: 18
    pt; color : black">'+str(i)+'</div>'))
            marker.add_to(route_map)

        # Save the HTML file
        route_map.save('route.html')

        dirname = os.path.dirname(__file__)
        filename = os.path.join(dirname, 'route.html')
        url = QUrl.fromLocalFile(filename)
        print(os.path.exists(filename))
        self.preview.load(url)
        print("Route sent")


if __name__ == '__main__':
    app = QApplication(sys.argv)
    form = Form()
    form.show()
    sys.exit(app.exec_())
```

C.2 algorithms

C.2.1 ant_colony.py

```
#Solve the TSP problem with the ant_colony algorithm

class ant_colony:

    class ant():

        def __init__(self, graph, start_node, alpha, beta, first_pass=False,
    heuristic=None):
            """Create an ant

            Args:
                graph: The graph to visit
                start: The node where the ant starts
                alpha: The alpha parameter of the algorithm, usually smaller
    than 1
                beta: The beta parameter of the algorithm, usually bigger than
     1"""

            self.graph = graph
            self.start_node = start_node
            self.path = [start_node]
            self.current = start_node
            self.distance = float(0)
            self.alpha = alpha
            self.beta = beta

            self.finished = False
            self.first_pass = first_pass

            if heuristic is None:
                self.heuristic = self._heuristic

        def run(self):
            """Run the ant until it has visited all the nodes"""
            while len(self.path) < len(self.graph):
                self._move()
```

```
34
35              self.path.append(self.start_node)
36              self.finished = True
37
38          def _move(self):
39              """Move the ant to the next node and update the path and the
    distance"""
40
41              #Get the neighbors of the current node
42              neighbors = list(self.graph[self.current].keys())
43
44              #Remove the nodes already visited
45              for node in self.path:
46                  if node in neighbors:
47                      neighbors.remove(node)
48
49              #If there is no neighbor, the ant is stuck
50              if len(neighbors) == 0:
51                  self.distance += self.graph[self.current][self.start_node]["
    time"]
52                  self.current = self.start_node
53                  self.path.append(self.start_node)
54
55                  self.finished = True
56                  return
57
58              #Compute the probability of each neighbor
59              probabilities = self._probability(neighbors)
60
61              #Choose the next node
62              next_node = self._choose(probabilities)
63
64              #Add the distance to the total distance
65              self.distance += self.graph[self.current][next_node]["time"]
66
67              #Update the current node and the path
68              self.current = next_node
69              self.path.append(next_node)
70
71          def _probability(self, visitable_nodes):
72              """Compute the probability of going to the node
73              Use the function on the wikipedia page
74
75              Args:
76                  node: The node to go to
77
78              Returns:
79                  The probability of going to the node"""
80
81              sum_of_probabilities = 0
82              probabilities = {}
83
84              #If it's the first pass, the probability is 1 for each node
85              if self.first_pass:
86                  for node in visitable_nodes:
87                      probabilities[node] = 1
88                      sum_of_probabilities += probabilities[node]
89              else:
90                  for node in visitable_nodes:
91                      probabilities[node] = self.heuristic(node) ** self.beta *
    self.graph[self.current][node]["pheromone"] ** self.alpha
92                      sum_of_probabilities += probabilities[node]
```

```
93
94              #Normalize the probabilities
95              if sum_of_probabilities == 0:
96                  for node in visitable_nodes:
97                      probabilities[node] = 1 / len(visitable_nodes)
98              else:
99                  for node in visitable_nodes:
100                     probabilities[node] /= sum_of_probabilities
101
102             #Compute the probability
103             return probabilities
104
105         def _choose(self, probabilities):
106             """Choose the next node to go to
107
108             Args:
109                 probabilities: The probabilities of each node
110
111             Returns:
112                 The node to go to"""
113
114             from random import random
115
116             #Choose a random number between 0 and 1
117             r = random()
118
119             #Choose the node
120             for node in probabilities.keys():
121                 r -= probabilities[node]
122                 if r <= 0:
123                     return node
124
125
126         def _heuristic(self, node):
127             """Compute the heuristic of the node
128             Use the function on the wikipedia page
129
130             Args:
131                 node: The node to compute the heuristic
132
133             Returns:
134                 The heuristic of the node"""
135             return 1 / self.graph[self.current][node]["time"]
136
137     def __init__(self, graph, start_node, alpha=0.5, beta=2, rho=0.5, n_ants
    =50, omega=100, first_pass=True, heuristic=None):
138         """Create an ant_colony object
139
140         Args:
141             graph: The graph to visit
142             start: The node where the ants start
143             alpha: The alpha parameter of the algorithm, usually smaller than
    1
144             beta: The beta parameter of the algorithm, usually bigger than 1
145             rho: The rho parameter of the algorithm, usually between 0 and 1
146             n_ants: The number of ants
147             omega: The omega parameter of the algorithm, stop the algorithm if
     the best ant has not improved for omega iterations
148             first_pass: If True, the ants will visit all the nodes randomly on
     the first pass
149             heuristic: The heuristic function to use"""
150
```

```
151         self.graph = graph
152         self.start_node = start_node
153         if alpha < 0 :
154             raise ValueError("alpha must be positive")
155         self.alpha = alpha
156         if beta < 0 :
157             raise ValueError("beta must be positive")
158         self.beta = beta
159         if rho < 0 or rho > 1:
160             raise ValueError("rho must be between 0 and 1")
161         self.rho = rho
162         self.n_ants = n_ants
163         self.omega = omega
164         self.first_pass = first_pass
165         self.heuristic = heuristic
166 
167         self.ants = []
168 
169     def run(self):
170         """Run the algorithm"""
171         #Create the ants
172         best_ant = self.ant(self.graph, self.start_node, self.alpha, self.beta
    , self.first_pass, self.heuristic)
173         best_ant.path = []
174         best_ant.distance = float("inf")
175         #Run the iterations
176         no_improvement = 0
177         iteration = 0
178         while no_improvement < self.omega:
179             self._iteration()
180             iteration += 1
181             old_best_ant_distance = best_ant.distance
182             for ant in self.ants:
183                 if ant.distance < best_ant.distance:
184                     best_ant = ant
185 
186             #If the best ant has not improved, increment no_improvement
187             if old_best_ant_distance == best_ant.distance:
188                 no_improvement += 1
189             else:
190                 no_improvement = 0
191 
192         print("Number of iterations: " + str(iteration))
193         #Find the best ant
194 
195         return best_ant.path
196 
197     def _iteration(self):
198         """Run one iteration of the algorithm"""
199         #Create the ants
200         self.ants = []
201         for i in range(self.n_ants):
202             self.ants.append(self.ant(self.graph, self.start_node, self.alpha,
    self.beta, self.first_pass, self.heuristic))
203 
204         #Run the ants
205         for ant in self.ants:
206             ant.run()
207 
208         #Update the pheromone
209         self._update_pheromone()
210 
```

```
211         self.first_pass = False
212 
213     def _update_pheromone(self):
214         """Update the pheromone of each edge with the formula on the wikipedia
    page"""
215         for edge in self.graph:
216             for edge2 in self.graph[edge]:
217                 #Remove the pheromone
218                 if not self.first_pass:
219                     self.graph[edge][edge2]["pheromone"] *= (1 - self.rho)
220                 else:
221                     max = 0
222                     #If it's the first pass, the pheromone is equal to 1
    devided by the maximum distance
223                     for ant in self.ants:
224                         if ant.distance > max:
225                             max = ant.distance
226                     self.graph[edge][edge2]["pheromone"] = 1 / max
227 
228             #Add the pheromone
229         for ant in self.ants:
230             for i in range(len(ant.path) - 1):
231                 self.graph[ant.path[i]][ant.path[i + 1]]["pheromone"] += 1 /
    ant.distance
```

C.2.2 astar.py

```
1 from heapq import heappop, heappush
2 import networkx as nx
3 import math
4 
5 
6 def astar(Graph, source, target):
7     """Find shortest weighted paths in G from source to target using  A*
    algorithm.
8     Parameters:
9     -----------
10    G : NetworkX oriented graph
11    source : node
12       Starting node for path
13    target : node
14         Ending node for path
15    weight : string, optional (default='weight')
16         Edge data key corresponding to the edge weight
17    Returns:
18    --------
19    distance : dictionary
20         Dictionary of shortest weighted paths keyed by target.
21    """
22 
23 
24    def heuristic(u, v):
25        # Get the latitude and longitude coordinates of the nodes
26        lat1 = Graph.nodes[u]["y"]
27        lon1 = Graph.nodes[u]["x"]
28        lat2 = Graph.nodes[v]["y"]
29        lon2 = Graph.nodes[v]["x"]
30 
31        # Convert the coordinates to radians
32        lat1 = lat1 * math.pi / 180
33        lon1 = lon1 * math.pi / 180
```

```
        lat2 = lat2 * math.pi / 180
        lon2 = lon2 * math.pi / 180

        # Calculate the great circle distance between the two points
        a = math.sin((lat2-lat1)/2)**2 + math.cos(lat1) * math.cos(lat2) *
    math.sin((lon2-lon1)/2)**2
        c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
        d = 6371 * c
        t = (d / 30) * 3600 #Convert to seconds at 30 km/h

        return t

    if source == target:
        return (0, [source])

    push = heappush
    pop = heappop

    #Create an heuristic target variable to let it switch between the source
    and the target
    heuristic_target = [target, source]


    neighbor_list=[Graph._succ,Graph._pred]

    #We are using the two way A* algorithm, so we need to keep track of the
    distances from both the source and the target
    out = [{}, {}] #List of dictionaries, each dictionary contains the
    distance from the source/target to each node
    path = [{},{}]  # dictionary of paths
    seen = [{source : 0},{target : 0}]  # dictionary of nodes that have been
    visited
    to_explore = [[],[]] # heap of (distance, heuristic, label) tuples for all
     non-seen nodes

    #Initialize the heap with the source and target
    push(to_explore[0], (heuristic(source, target), 0,  source))
    push(to_explore[1], (heuristic(target, source), 0,  target))

    #Initialize the path with the source and target
    path[0][source] = [source]
    path[1][target] = [target]

    direction = 1 #Direction of the search, 0 is forward, 1 is backward

    while to_explore[0] and to_explore[1]:

        direction = 1 - direction #Switch direction

        #Pop the smallest distance node from the heap
        ( _ , dist, v) = pop(to_explore[direction])

        if v in out[direction]:
            continue

        out[direction][v] = dist

        #Check if the node has been visited by the other search
        if v in out[1-direction]:
            #If it has, we have found a shortest path
            return (out[0][v] + out[1][v], path[0][v] + list(reversed(path[1][
    v]))[1:])
```

```
89
90          for neighbor in neighbor_list[direction][v]:
91              if neighbor not in out[direction]:
92                  weight = weight_node(v, neighbor, out[direction], Graph,
    direction)
93                  #If the neighbor has not been visited, add it to the heap
94                  if neighbor not in seen[direction]:
95                      seen[direction][neighbor] = weight
96                      push(to_explore[direction], (weight + heuristic(neighbor,
    heuristic_target[direction]), weight, neighbor))
97                      path[direction][neighbor] = path[direction][v] + [neighbor
    ]
98                  #If the neighbor has been visited, but the new path is shorter
    , update the heap
99                  elif weight < seen[direction][neighbor]:
100                     seen[direction][neighbor] = weight
101                     push(to_explore[direction],  (weight + heuristic(neighbor,
     heuristic_target[direction]), weight, neighbor))
102                     path[direction][neighbor] = path[direction][v] + [neighbor
    ]
103
104     return (float('inf'), [])
105
106
107
108
109 def travel_time_route(u, v, graph):
110     """Return the travel time of a route."""
111     return graph[u][v][0]['travel_time'] if 'travel_time' in graph[u][v][0]
    else graph[u][v][0]["length"]/8.33 # 30 km/h
112
113 def weight_node(u, v, data, graph, direction):
114     """Return the weight of an edge."""
115     weight = data[u]
116     if direction == 0:
117         weight += travel_time_route(u, v, graph)
118     else:
119         weight += travel_time_route(v, u, graph)
120     return weight
```

C.2.3 christofides.py

```
1 from heapq import heappop, heappush
2 from networkx import Graph, max_weight_matching
3 from itertools import count
4 from copy import deepcopy
5 from collections import deque
6
7
8 def christofides(dictionary, weight="time"):
9     """Compute an approximation of the shortest path between all the nodes of
    the dictionary
10     Uses Christofides Algorithm to solve the traveling's salesman problem (TSP
    ).
11
12     Parameters:
13     ---------------
14     dictionary: python dictionary, complete graph
15     weight: weight used in the dictionary
16
17     Returns:
```

```
18      an approximation of the shortest path between the dictionary's nodes
19      """
20      non_oriented_graph = oriented_to_non_oriented_graph(dictionary=dictionary,
        weight="time")
21      tree = prim_dictionnary(dict=non_oriented_graph, weight="time")
22      odd_graph = get_odd_graph(dictionary, tree)
23      min_weight_matchings = dic_min_weight_matching(odd_graph)
24      eulerian_graph = create_eulerian_graph(tree, min_weight_matchings)
25      eulerian_path = hierholzer_eulerian_circuit(eulerian_graph)
26      hamiltonian_path = shortcutting(eulerian_path)
27      new_path = reorder(hamiltonian_path, list(dictionary.keys())[0])
28      return new_path
29
30
31  def oriented_to_non_oriented_graph(dictionary, weight="time"):
32      """Compute the non oriented version of the dictionary, cleans the
        dictionary
33
34      Parameters:
35      --------------
36      dictionary: oriented graph like [start_node][end_node][weight] = weight
37      """
38      dictionary_copy = deepcopy(dictionary)
39      new_dictionary = {node: {} for node in dictionary_copy.keys()}
40      for start_node in dictionary_copy:
41          for end_node in dictionary_copy[start_node]:
42              if start_node == end_node:
43                  continue
44              w = (dictionary_copy[start_node][end_node][weight] +
        dictionary_copy[start_node][end_node][weight]) / 2
45              new_dictionary[start_node][end_node] = {weight : w}
46              new_dictionary[end_node][start_node] = {weight : w}
47              del dictionary_copy[end_node][start_node]
48      return new_dictionary
49
50
51  def prim_dictionnary(dict, weight="time"):
52      """Compute a the edges of a minimim spanning tree (mst) for a graph
53      Uses Prim's algorithm
54
55      Parameters:
56      ---------------
57      G: nx.Graph
58      weight: weight used in the graph
59
60      Returns:
61      An iterator containing the edges of the mst
62      """
63      push = heappush
64      pop = heappop
65      nodes = set(dict)
66      tree = {node : {} for node in nodes}
67      while nodes:
68          u = nodes.pop() #arbitrary node
69          heap = []
70          visited = {u}
71          for v, d in dict[u].items():
72              wt = d.get(weight, 1)
73              push(heap, (wt, u, v, d))
74          while nodes and heap:
75              _, u, v, d = pop(heap)
76              if v in visited or v not in nodes:
```

```
                continue
            tree[u][v] = d
            tree[v][u] = d
            visited.add(v)
            nodes.discard(v)
            for w, d2 in dict[v].items():
                if w in visited:
                    continue
                new_weight = d2.get(weight, 1)
                push(heap, (new_weight, v, w, d2))
    return tree


def get_odd_graph(dictionary, tree):
    """Get the vertices with odd degree of the tree from the main graph to get a new graph
     Parameters:
    ---------------
        dictionary: graph under dictionary form
        tree: minimum spanning tree

    Returns:
        I: dictionary containing the vertices of odd degree
    """
    I = deepcopy(dictionary)
    for node in tree:
        if not (len(tree[node]) % 2):
            for second_node in I[node]:
                I[second_node].pop(node)
            I.pop(node)
    return I


def dic_min_weight_matching(dictionary, weight="time"):
    """Calculate a minimum weight matching from a graph

     Parameters:
    ---------------
        dictionary: graph under dictionary form
        weight: weight used in the graph

    Returns:
        edges: dictionary containing the matchings

    """
    max_weight = 1 + max(dictionary[first_node][second_node][weight] for first_node in dictionary for second_node in dictionary[first_node])
    new_dic = deepcopy(dictionary)
    for first_node in dictionary:
        for second_node in dictionary[first_node]:
            new_dic[first_node][second_node][weight] = max_weight - dictionary[first_node][second_node][weight]

    InvG = Graph(new_dic)
    edges = {}
    matchings = max_weight_matching(InvG, maxcardinality=True, weight=weight)
    for matching in matchings:
        matching = list(matching)
        u = matching[0]
        v = matching[1]
        for node in matching:
            edges[node] = {}
```

```
136         w = dictionary[u][v][weight]
137         edges[v][u] = {weight : w}
138         edges[u][v] = {weight : w}
139     return edges
140 
141 
142 def create_eulerian_graph(tree, min_weight_matchings):
143     """ Adds the tree and minimum weight matchings results to get the eulerian
    graph
144 
145     Parameters:
146     ---------------
147         tree: minimum spanning tree
148         min_weight_matchings: dictionary containing the matchings
149 
150     Returns:
151         graph: dictionary containing the eulerian graph
152     """
153     graph = deepcopy(tree)
154     for node, value in min_weight_matchings.items():
155         graph[node].update(value)
156     return graph
157 
158 
159 def hierholzer_eulerian_circuit(eulerian_graph):
160     """Calculate an eulerian circuit in an eulerian graph
161 
162     Parameters:
163     ---------------
164         eulerian_graph: eulerian graph under dictionary form
165 
166     Returns:
167         path: list containing the circuit
168     """
169     dic = deepcopy(eulerian_graph)
170     first_vertex = list(dic.keys())[0]
171     vertex_stack = [first_vertex]
172     path = []
173     last_vertex = None
174     while vertex_stack:
175         current_vertex = vertex_stack[-1]
176         if dic[current_vertex]:
177             next_vertex, _ = list(dic[current_vertex].items())[0]
178             vertex_stack.append(next_vertex)
179             dic[current_vertex].pop(next_vertex)
180             dic[next_vertex].pop(current_vertex)
181         else:
182             if last_vertex is not None:
183                 path.append(last_vertex)
184             last_vertex = current_vertex
185             vertex_stack.pop()
186     path.append(current_vertex)
187     return  path
188 
189 
190 def shortcutting(path):
191     """Removes the nodes through which the path comes through twice
192     Parameters:
193     ---------------
194         path: list
195 
196     Returns:
```

```
197         nodes: list
198     """
199     nodes = [path[0]]
200     for node in path:
201         if node in nodes:
202             continue
203         nodes.append(node)
204     return nodes
205 
206 
207 def reorder(list, first):
208     """Rotate the list until the parameters "first" is in first position, then
        appends it to the list
209     Parameters:
210     ---------------
211         list: list
212         first: int
213 
214     Returns:
215         list: list
216     """
217     while(list[0] != first):
218         list.append(list.pop(0))
219     list.append(first)
220     return list
```

C.2.4 dijkstra.py

```
1 from heapq import heappop, heappush
2 
3 import networkx as nx
4 
5 
6 
7 def dijkstra(Graph, source, target):
8     """Find shortest weighted paths in G from source to target using Dijkstra'
      s algorithm.
9     Parameters:
10    -----------
11    G : NetworkX oriented graph
12    source : node
13       Starting node for path
14    target : node
15         Ending node for path
16    weight : string, optional (default='weight')
17         Edge data key corresponding to the edge weight
18    Returns:
19    --------
20    time : float
21        Shortest time from source to target.
22    path : list
23        List of nodes in a shortest path.
24    """
25 
26    if source == target:
27        return (0, [source])
28 
29    push = heappush
30    pop = heappop
31 
32    neighbor_list=[Graph._succ,Graph._pred]
```

```
33
34      #We are using the two way Dijkstra algorithm, so we need to keep track of
    the distances from both the source and the target
35      out = [{}, {}] #List of dictionaries, each dictionary contains the
    distance from the source/target to each node
36      path = [{},{}]  # dictionary of paths
37      seen = [{source : 0},{target : 0}]  # dictionary of nodes that have been
    visited
38      to_explore = [[],[]] # heap of (distance,label) tuples for all non-seen
    nodes
39
40      #Initialize the heap with the source and target
41      push(to_explore[0], (0, source))
42      push(to_explore[1], (0, target))
43
44      #Initialize the path with the source and target
45      path[0][source] = [source]
46      path[1][target] = [target]
47
48      direction = 1 #Direction of the search, 0 is forward, 1 is backward
49
50      while to_explore[0] and to_explore[1]:
51
52          direction = 1 - direction #Switch direction
53
54          #Pop the smallest distance node from the heap
55          (dist, v) = pop(to_explore[direction])
56
57          if v in out[direction]:
58              continue
59
60          out[direction][v] = dist
61
62          #Check if the node has been visited by the other search
63          if v in out[1-direction]:
64              #If it has, we have found a shortest path
65              return (out[0][v] + out[1][v], path[0][v] + list(reversed(path[1][
    v]))[1:])
66
67          for neighbor in neighbor_list[direction][v]:
68              if neighbor not in out[direction]:
69                  weight = weight_node(v, neighbor, out[direction], Graph,
    direction)
70                  #If the neighbor has not been visited, add it to the heap
71                  if neighbor not in seen[direction]:
72                      seen[direction][neighbor] = weight
73                      push(to_explore[direction], (weight, neighbor))
74                      path[direction][neighbor] = path[direction][v] + [neighbor
    ]
75                  #If the neighbor has been visited, but the new path is shorter
    , update the heap
76                  elif weight < seen[direction][neighbor]:
77                      seen[direction][neighbor] = weight
78                      push(to_explore[direction], (weight, neighbor))
79                      path[direction][neighbor] = path[direction][v] + [neighbor
    ]
80
81      return (float('inf'), [])
82
83
84  def travel_time_route(u, v, graph):
85      """Return the travel time of a route."""
```

```
    return graph[u][v][0]['travel_time'] if 'travel_time' in graph[u][v][0]
    else graph[u][v][0]["length"]/8.33 # 30 km/h

def weight_node(u, v, data, graph, direction):
    """Return the weight of an edge."""
    weight = data[u]
    if direction == 0:
        weight += travel_time_route(u, v, graph)
    else:
        weight += travel_time_route(v, u, graph)
    return weight
```

C.2.5 pairwise_exchange.py

```
import networkx as nx
import osmnx as ox
from itertools import permutations, combinations
from random import choice

from algorithms import dijkstra


def multiNodes_to_fullGraph(graph: nx.MultiDiGraph, multi_nodes: list[nx.nodes
    ], algo=dijkstra.dijkstra) -> nx.DiGraph:

    full_graph = nx.DiGraph()

    # get tuple of every possible combination, permutations allowed NOT
    duplicates
    paths_to_find = permutations(multi_nodes, 2)

    # add to graph every weighted edge and store path with it
    for edge in paths_to_find:

        weight, path_found = algo(graph, *edge)
        full_graph.add_weighted_edges_from([(*edge, weight)], path=path_found)

    return full_graph

def fullGraph_to_ringGraph(full_graph: nx.DiGraph) -> nx.DiGraph:
    ring_graph = nx.DiGraph()

    # randomly pick a node
    start_node = list(full_graph.nodes)[0]

    # get nearest node
    nearest_node, properties = min(full_graph[start_node].items(), key=lambda
    edge: edge[1]["time"])

    # add weight and node
    ring_graph.add_edge(start_node, nearest_node, weight = properties["time"])

    search_node = nearest_node

    while not all(node in ring_graph for node in full_graph.nodes):
        # get a sorted list of the nearest node
        sorted_nearest_nodes = sorted(full_graph[search_node].items(), key=
    lambda edge: edge[1]["time"])
        for nearest_node, properties in sorted_nearest_nodes:
            if nearest_node not in ring_graph:
```

```
                ring_graph.add_edge(search_node, nearest_node, weight =
    properties["time"])
                search_node = nearest_node
                break

    ring_graph.add_edge(search_node, start_node, weight=full_graph[start_node
    ][nearest_node]["time"])

    return ring_graph

def get_path_graphs(ring_graph: nx.DiGraph)-> list[nx.DiGraph]:
    path_graph_list = [component for component in nx.
    weakly_connected_components(ring_graph)]

    return path_graph_list

def exchange_nodes(full_graph: nx.DiGraph, ring_graph: nx.DiGraph, recursion:
    int):
    edges = list(ring_graph.edges)

    #get a random edge and convert to list
    switch_edge1 = list(edges.pop(edges.index(choice(edges))))
    switch_edge1.append(full_graph.get_edge_data(*switch_edge1, "time"))

    # remove neighbor edges of the nodes in switch_edge1
    edges = [edge for edge in edges if not(switch_edge1[0] in edge or
    switch_edge1[1] in edge)]

    #get a random edge and convert to list
    switch_edge2 = list(edges.pop(edges.index(choice(edges))))
    switch_edge2.append(full_graph.get_edge_data(*switch_edge2, "time"))
    ring_graph.remove_edges_from([switch_edge1, switch_edge2])

    # swap edges
    switch_edge1[1], switch_edge2[0] = switch_edge2[0], switch_edge1[1]

    # ring graph is currently composed of 2 path graphs
    list_path_graph_nodes = get_path_graphs(ring_graph)

    # create the 2 path graphs
    path_graph_A = ring_graph.copy()
    path_graph_A.remove_nodes_from(list_path_graph_nodes[0])
    path_graph_B = ring_graph.copy()
    path_graph_B.remove_nodes_from(list_path_graph_nodes[1])

    # find the path graph to reverse
    graph_reversed = None
    if switch_edge1[1] in path_graph_A:
        if path_graph_A.predecessors(switch_edge1[1]):
            path_graph_A = path_graph_A.reverse()
            graph_reversed = path_graph_A
        else:
            path_graph_B = path_graph_B.reverse()
            graph_reversed = path_graph_B
    else:
        if path_graph_B.predecessors(switch_edge1[1]):
            path_graph_B = path_graph_B.reverse()
            graph_reversed = path_graph_B
        else:
            path_graph_A = path_graph_A.reverse()
            graph_reversed = path_graph_A

```

```
    # update weights of reversed path
    weighted_edges = [(*edge, full_graph[edge[0]][edge[1]]["time"]) for edge
    in graph_reversed.edges]
    graph_reversed.add_weighted_edges_from(weighted_edges)

    # reconstruct graph
    ring_graph = nx.union(path_graph_A, path_graph_B)
    ring_graph.add_edges_from([switch_edge1, switch_edge2])

    print(f"recursion: {recursion} size: ", ring_graph.size(weight="time"))

    # return swapped graph
    if recursion == 1:
        return ring_graph

    new_ring_graph = exchange_nodes(full_graph, ring_graph.copy(), recursion -
    1)

    # return the smallest weighted paths
    if new_ring_graph.size(weight="time") < ring_graph.size(weight="time"):
        return new_ring_graph

    return ring_graph


def ring_graph_to_multinodes(ring_graph: nx.DiGraph, start_node):
    multinodes = [start_node]
    search_node = start_node

    for i in range(len(ring_graph.nodes)):
        for adj in ring_graph.adj[search_node]:
            multinodes.append(adj)
        search_node = adj
    return multinodes



def pairwise_exchange(dictionnary, multinodes: list[nx.nodes], recursion):
    full_graph = nx.DiGraph(dictionnary)
    ring_graph = fullGraph_to_ringGraph(full_graph)

    ring_graph = exchange_nodes(full_graph, ring_graph, recursion)

    multinodes = ring_graph_to_multinodes(ring_graph, start_node=multinodes
    [0])
    print(multinodes)
    return multinodes
```

C.3 graph_tools

C.3.1 ConstructGraph.py

```
from algorithms import dijkstra
#Construct a graph representation of the network of places to visited ready to
    be used by a TSP solver.

def construct_graph(graph, nodes, algorithm = dijkstra.dijkstra):
    """Construct a graph representation of the network of places to visited
    ready to be used by a TSP solver.
    The graph is represented as a dictionary of dictionaries. The keys are the
    nodes of the graph,
```

```
    and the values are dictionaries containing the time needed to travel between the node and its neighbors and
    the path to take to reach them.

    Parameters:
    graph (networkx graph): the graph of the network
    nodes (list): the list of nodes to visit

    Returns:
    dict: the graph representation
    """

    #Create a graph with only the nodes to visit
    G = {node: {} for node in nodes}

    for start_node in nodes:
        for end_node in nodes:

            if start_node == end_node:
                continue

            #Find the shortest path between the two nodes
            time, path = algorithm(graph, start_node, end_node)

            if(time != float("inf")):
                #Add the path to the graph
                G[start_node][end_node] = {"time": time, "path": path}


    return G
```

C.3.2 TSP_solver.py

```
from graph_tools import ConstructGraph, input_generator
from algorithms import ant_colony, christofides, pairwise_exchange, astar, dijkstra
import osmnx as ox
import time as timestamp

def main_solver(nodes_to_visit, name_algorithm1 = "Dijkstra", name_algorithm2= "Christofides"):

    #select the algorithm to use
    algorithm1 = choose_algorithm(name_algorithm1)

    #Get the coordinates of the nodes
    nodesgeocode = NodesToCoordinates(nodes_to_visit)


    #Download the graph to run the algorithm on
    start = timestamp.time()
    nodes_to_visit, graph = graph_from_coordinates_array(nodesgeocode)
    end = timestamp.time()
    print("Time to download the graph: ", end - start)

    #Mesure the time to run the first algorithm
    start = timestamp.time()
    print("Start to create the graph with the algorithm: ", name_algorithm1, "")
    #Create a fully connected graph with only the nodes to visit with the algorithm1
```

```
25      ConnectedSimplifiedGraph = ConstructGraph.construct_graph(graph,
        nodes_to_visit, algorithm1)
26      end = timestamp.time()
27      print("Time to create a  the graph: ", end - start)
28
29      #If there is only two nodes, we don't need to run the TSP solver
30      if len(nodesgeocode) == 2:
31          path = ConnectedSimplifiedGraph[nodes_to_visit[0]][nodes_to_visit[1]][
        "path"]
32          time = ConnectedSimplifiedGraph[nodes_to_visit[0]][nodes_to_visit[1]][
        "time"]
33          return graph, path, time, [nodesgeocode[0],nodesgeocode[1]]
34
35      else:
36
37          solution_simplified_path = tsp_solver(nodes_to_visit,
        ConnectedSimplifiedGraph, name_algorithm2)
38          path, time = get_path_time(nodes_to_visit, ConnectedSimplifiedGraph,
        solution_simplified_path)
39          nodesgeocode = [nodesgeocode[nodes_to_visit.index(node)] for node in
        solution_simplified_path]
40          return graph, path, time, nodesgeocode
41
42  def tsp_solver(nodes, dictionnary, algorithm_name="Christofides"):
43      """Solve the TSP problem with the algorithm chosen.
44      :param graph: the graph of the network
45      :param nodes: the list of nodes to visit
46      :param dictionnary: the graph representation
47      :param algorithm_name: the name of the algorithm to use
48      :return: the path to take
49      """
50
51      #Solve the TSP problem with the algorithm2
52      start = timestamp.time()
53      if(algorithm_name == "Ant Algorithm"):
54          colony = ant_colony.ant_colony(dictionnary, nodes[0],n_ants=25, alpha
        =0.75, beta=3, rho=0.1, omega=50)
55          simplified_solution_path = colony.run()
56      elif algorithm_name == "Christofides":
57          simplified_solution_path = christofides.christofides(dictionnary)
58      elif algorithm_name == "Pairwise exchange":
59          simplified_solution_path = pairwise_exchange.pairwise_exchange(
        dictionnary, nodes, len(nodes))
60      else:
61          raise NameError("Unknown algorithm")
62      end = timestamp.time()
63
64      print("Time to solve the TSP problem: ", end - start)
65
66      return simplified_solution_path
67
68  def choose_algorithm(algorithm="Dijkstra"):
69      """Choose the algorithm 1 to use
70      :param algorithm: the name of the algorithm to use
71      :return: the function of the algorithm"""
72      algorithm_dictionary = {
73          "Dijkstra": dijkstra.dijkstra,
74          "A*": astar.astar
75      }
76      function = algorithm_dictionary.get(algorithm)
77      if function is None:
78          raise NameError("Unknown algorithm")
```

```
79      print("TSP algorithm:", algorithm)
80      return function
81
82  def get_path_time(nodes, dictionnary, simplified_path):
83      """
84      This function takes a list of nodes and returns the path and the time to
        go through it
85      :param nodes: list of nodes
86      :param dictionnary: dictionnary of the graph
87      :param simplified_path: list of nodes
88      :return: path and time"""
89      time = 0
90      path = [nodes[0]]
91      for i in range(len(simplified_path)-1):
92          path += dictionnary[simplified_path[i]][simplified_path[i+1]]["path"
        ][1:]
93          time +=  dictionnary[simplified_path[i]][simplified_path[i+1]]["time"]
94      return path, time
95
96  def NodesToCoordinates(NodesName):
97      """
98      This function takes a list of nodes and returns a list of coordinates
99      :param NodesName: list of nodes
100     :return: list of coordinates
101     """
102
103     geocode_list = []
104
105     for NodeName in NodesName:
106         geocode_list.append(ox.geocode(NodeName))
107
108     return geocode_list
109
110 def graph_from_coordinates_array(coordinates_array, simplify=True,
        network_type='drive'):
111     """Create a where the list of coordiante is in the graph
112     :param coordinates_array: list of coordinates
113     :param simplify: boolean to simplify the graph
114     :param network_type: type of network
115     :return: list of nodes and graph
116     """
117     #If there is the same adress, we remove the duplicates
118     coordinates_array = list(dict.fromkeys(coordinates_array))
119
120     if len(coordinates_array) == 0:
121         raise ValueError("The list of coordinates is empty")
122     if len(coordinates_array) == 1:
123         raise ValueError("The list of coordinates contains only one element")
124
125
126     minlat, maxlat, minlon, maxlon = coordinates_to_bounds(coordinates_array)
127
128     graph  = ox.graph_from_bbox(maxlat,minlat,maxlon,minlon, simplify=simplify
        , network_type=network_type, truncate_by_edge=True)
129     graph = ox.add_edge_speeds(graph)
130     graph = ox.add_edge_travel_times(graph)
131     graph = ox.utils_graph.get_largest_component(graph, strongly=True)
132
133     nodes = []
134     for latitude, longitude in coordinates_array:
135         nodes.append(ox.nearest_nodes(graph, float(longitude), float(latitude)
        ))
```

```
136
137     return nodes, graph
138
139 def coordinates_to_bounds(nodesgeocode):
140     """Get the bounds of the coordinates and add a margin
141     :param coordinates_array: list of coordinates
142     :return: bounds
143     """
144     minlat = min([float(latitude) for latitude, _ in nodesgeocode])
145     maxlat = max([float(latitude) for latitude, _ in nodesgeocode])
146     minlon = min([float(longitude) for _, longitude in nodesgeocode])
147     maxlon = max([float(longitude) for _, longitude in nodesgeocode])
148     print(minlat, maxlat, minlon, maxlon)
149     #Padding to get a bigger area
150     padding = 0.1 * (maxlat - minlat)
151     minlat -= padding
152     maxlat += padding
153     padding = 0.1 * (maxlon - minlon)
154     minlon -= padding
155     maxlon += padding
156     #if the area is to linear, add padding to the other axis. This is to avoid
        the problem of the graph being a line
157     if maxlat - minlat < 0.5 * (maxlon - minlon):
158         padding = 0.5 * (maxlon - minlon) - (maxlat - minlat)
159         minlat -= padding / 2
160         maxlat += padding / 2
161     elif maxlon - minlon < 0.5 * (maxlat - minlat):
162         padding = 0.5 * (maxlat - minlat) - (maxlon - minlon)
163         minlon -= padding / 2
164         maxlon += padding / 2
165
166     print(minlat, maxlat, minlon, maxlon)
167     return minlat, maxlat, minlon, maxlon
```