CENTRALE LYON

École Centrale Lyon

ELC

Algorithmes collaboratifs

Rapport

Rapport de BE

Élèves :
Simon Hergott
Alexandre Louichon

Enseignant :
Alexandre Saïdi

7 avril 2025

Table des matières

1 Introduction

Le problème d'allocation de bus (Bus Allocation Problem, BAP) est un problème d'optimisation combinatoire crucial dans le domaine des transports publics, visant à assigner efficacement une flotte de bus à un ensemble de lignes de transport afin de minimiser les coûts opérationnels et d'améliorer le service aux usagers. Face à la complexité inhérente au BAP, notamment avec l'augmentation du nombre de lignes et de bus, les méthodes d'optimisation métaheuristiques se révèlent particulièrement efficaces pour obtenir des solutions fonctionnelles dans des délais raisonnables. Cette étude présente une implémentation du Bus Allocation Problem en utilisant une approche d'optimisation basée sur le système de colonies de fourmis (Ant Colony System, ACS), plus précisément une version multi-agents (mACS). L'objectif de ce travail est d'évaluer l'efficacité de cette approche pour résoudre le BAP et d'analyser les performances de l'implémentation développée.

Lien du projet : https://gitlab.ec-lyon.fr/shergott/elc-saidi-bap

2 Formalisation du problème

2.1 Compréhension

Il est important de formaliser les entrées et sorties du problème afin de pouvoir apporter une méthode de résolution sans ambiguités.

Quelles sont les données du système ? Le BAP étant un problème d'optimisation de lignes de bus, on a donc m lignes de bus et n noeuds (représentant des arrêts de bus). Ces noeuds sont reliés entre eux en formant des arcs, et la structure que nous avons choisi de prendre fait qu'un arc est un triplet :

$$(noeud1,\ noeud2,\ temps\ de\ parcours(noeud1,\ noeud2))$$

Pour simplifier les choses, on dit que le temps de parcours est équivalent à la distance euclidienne étant donné qu'on peut raisonnablement faire l'hypothèse que les fourmis se déplacent à vitesse égale et donc on obtient une relation bijective entre la distance et le temps.

Ainsi, on peut se permettre de faire l'amalgamme entre ces deux grandeurs, l'une illustre le propos (le temps de parcours est utile pour calculer l'efficacité du chemin parcouru puis l'average time of travel) et l'autre est utilisée dans le code pour l'implémentation concrète (les noeuds sont repérés par leurs coordonnées cartésiennes desquelles il est facile d'extirper la distance qui les sépare).

L'objectif du programme sera donc de parvenir, étant donné ce jeu de données, à trouver les chemins optimaux que doivent emprunter les lignes de bus afin de permettre de minimiser le temps de parcours moyen entre deux noeuds sélectionnés au hasard. On suppose qu'un changement de ligne pour se rendre à un endroit est immédiat (le service de bus est très fréquent) et n'entraine pas de délai. Il devra donc fournir un recouvrement (important) du graphe avec les lignes de bus puis optimiser ce recouvrement en vue de

cet objectif.

Plus précisément, on possède une fonction qui permet de calculer le temps de trajet moyen entre deux noeuds sélectionnés aléatoirement en calculant tous les temps de trajet de tous les noeuds dans le graphe de la manière suivante :

```
    def test_efficacite(self):
        # On veut deux choses:
        # - Pour tout couple de noeuds (arrêts) du graphe global, on veut pouvoir voyager de l'un à l'autre en prenant un (ou plusieurs) bus. Si c'est impossible (graphe disjoint), ca sert à rien de continuer
        # - On veut minimiser ce temps moyen de voyage entre deux noeuds pris au hasard

        # permet de vérifier que la solution des lignes de bus est bien couvrante du graphe global
        if not self.test_couverture_bus_2():
            return -1

        # Initialisation des compteurs
        total_time = 0
        num_pairs = 0

        # On parcourt tous les couples de noeuds du graphe global afin de calculer tous les temps de trajet (nécessite un graphe connexe et recouvert par les lignes de bus)
        for node1 in self.global_graph.nodes:
            for node2 in self.global_graph.nodes:
                if node1 != node2:
                    shortest_time = float('inf')
                    travel_time = self.global_graph.exists_path(node1, node2, self.lignes_bus)
                    if travel_time != -1: # exists_path renvoie -1 si le chemin n'existe pas
                        shortest_time = min(shortest_time, travel_time)
                    if shortest_time == float('inf'):
                        return -1  # Impossible de voyager entre deux noeuds, en théorie on a déjà vérifié ça avant
                    total_time += shortest_time
                    num_pairs += 1

        return total_time / num_pairs # utilisation tu temps de trajet moyen
```

Pour cette fonction (qui est la quantité que l'on cherche à minimiser dans l'optimisation BAP), il est utile de s'intéresser au fonctionnement de la fonction exists_path(node1, node2, self.lignes_bus) pour mieux comprendre l'architecture de notre implémentation :

```python
    def exists_path(self, node1, node2, lignes_bus):
        # implémentataion d'un DFS pour vérifier si un chemin existe entre deux noeuds
        visited = set()
        stack = [node1]
        path_time = 0
        last_known_node = node1  # Dernier noeud connu avant de partir vers le voisin
        while stack:
            current_node = stack.pop()
            if current_node == node2:
                return path_time
            if current_node not in visited:
                visited.add(current_node)
                # Ajout des voisins du current_node dans la pile en vérifiant qu'ils sont accessibles par une ligne de bus
                # On ne peut pas utiliser self.get_neighbors(current_node) car on ne ferait pas la distinction entre les arêtes globales du graphe et celles couvertes par les lignes de bus
                neighbors = []
                for ligne_id, ligne_bus in lignes_bus.items():
                    if current_node in ligne_bus.get_nodes():
                        # On ajoute les voisins du noeud actuel dans la liste des voisins
                        for other_node in ligne_bus.get_nodes():
                            if other_node != current_node and other_node not in visited and other_node not in neighbors and ligne_bus.exists_arc(current_node, other_node):
                                neighbors.append(other_node)

                path_time += self.get_arc(current_node, last_known_node)[2]  # Ajoute le temps de trajet entre les deux noeuds
                last_known_node = current_node  # Met à jour le dernier noeud connu
                stack.extend(neighbors)
        return -1 # Retourne -1 si aucun chemin n'existe entre les deux noeuds (convention)
```

Cette fonction ressemble beaucoup à une fonction de parcours en profondeur (normal, c'en est un), sauf que l'on considère deux spécificités qui en font toute sa complexité : d'une part, on cherche à ne renvoyer non pas simplement un booléen qui dit s'il existe un chemin entre node1 et node2, mais également la durée du temps de trajet entre ces deux noeuds, que l'on calcule grâce à un compteur et à get_arc(last_known_node, current_node)[2] qui renvoie le 3e élément du tuple constitué de l'arc reliant node1 et node2 c'est à dire t_{ij}.

La deuxième complexité, plus subtile mais autrement plus importante, vient du fait que cette fonction se situe dans graph_global donc on pourrait être tentés d'accéder aux

voisins de current_node par la fonction get_neighbors(current_node). Cela fonctionnerait théoriquement mais ne prendrait pas en compte le fait que pour se déplacer d'un noeud à un autre, il faut... prendre un bus ! (c'est tout le principe de l'optimisation). On est donc obligés de faire une boucle dans toutes les lignes de bus, vérifier si current_node est traversée par la ligne de bus, trouver ses voisins, vérifier s'ils n'ont pas déjà été visités et s'ils ne se trouvent pas déjà dans la liste des voisins à visiter, puis enfin si les voisins en question sont accessibles depuis cette ligne de bus.

Une fois affranchis de ces subtilités, on obtient le travel_time (-1 s'il n'y a pas de trajet entre i et j, mesure de sécurité car le graphe est normalement connexe et recouvert par la solutions des lignes de bus), puis on renvoie le temps moyen en divisant ce temps de trajet total par le nombre de trajet qu'il représente.

2.2 Approche mACS

On construit m colonies de fourmis (ACS) indépendantes (ie : qui mettent à jour leurs phéromones séparément), chaque ACS représentant l'optimisation d'une ligne de bus. De plus, chaque ACS est constitué de r fourmis, donc à la fin de chaque itération du programme, on aura r solutions possibles de trajets entre les noeuds de départ et les noeuds d'arrivée pour chaque colonie, et chaque recouvrement ainsi obtenu fournit un temps de parcours moyen qui est ensuite comparé à celui des autres recouvrements pour savoir lequel est le meilleur.

3 Méthodologie pour la résolution

Le programme que nous proposons se divise en plusieurs parties :
— L'interface graphique
— La gestion des données du problème (classe GlobalGraph et BusGraph)
— La création des ACS adaptés à la situation (classe Ant_Colony et Ant)
— Le lancement de l'optimisation avec la classe optimizer

Nous avons souhaité diviser le code en modules afin de simplifier la gestion de ce dernier ainsi qu'améliorer la visibilité. Lorsque l'utilisateur souhaite résoudre le problème du BAP, il lui faut d'abord un graphe global sur lequel l'optimisation devra se faire ; Il peut soit le créer manuellement soit en ouvrir un grâce au bouton charger.

Par la suite, il entre le nombre de colonies souhaitées [m] (équivalent au nombre de lignes de bus du problème), et le programme initialise donc m ACS en prenant pour chaque colonie un noeud de départ aléatoire. Chaque ACS contient r fourmis prêtes à explorer le graphe. Cependant, il faut d'abord une solution globale recouvrante, sans quoi l'exploration des fourmis ne permettra pas d'atteindre les noeuds isolés (l'exploration des ACS se fait en fonction du sous-graphe formé par les noeuds et arêtes déjà visitées par la ligne de bus correspondante, ainsi que les voisins de tous ces noeuds).

Il y a donc un premier temps lors duquel un parcours en profondeur permet d'étendre les lignes de bus jusqu'à recouvrir entièrement le graphe. Une fois cela fait, chaque fourmi

va explorer ou exploiter les noeuds de son réseau (graphe de sa colonie donc sous-graphe global) jusqu'à atteindre son noeud objectif (fourni par sa colonie). Le choix des noeuds à explorer se fait en fonction des phéromones rencontrées : afin de garantir que les colonies de fourmis se partagent le graphe et ne le recouvrent pas entièrement chacunes, il faut pénaliser une trop grande expansion. Pour cela, nous avons introduit un paramètre *gamma* poussant une colonie de fourmis à ne pas réutiliser d'arcs utilisés par une colonie concurrente. On maximise alors l'efficacité, en ne couvrant que peu d'arcs par deux colonies simultanément.

```
def expansion(self, global_graph):
        visited_nodes = set(self.noeuds.keys())  # Ensemble des noeuds déjà visités
        # Parcourt les noeuds déjà visités pour trouver un voisin non visité
        for current_node in self.noeuds.keys():
            neighbors = global_graph.get_neighbors(current_node)  # Récupère les voisins du noeuds actuel dans le graphe global

            for neighbor in neighbors:
                # Vérifie que le voisin n'a pas été visité et qu'il existe un arc entre les deux noeuds
                if neighbor not in visited_nodes and global_graph.exists_arc(current_node, neighbor):
                    self.add_node(neighbor, *global_graph.nodes[neighbor])  # Ajoute le voisin au graphe du bus
                    self.add_arc(current_node, neighbor, global_graph.get_arc(current_node, neighbor)[2])  # Ajoute l'arc
                    return  # Arrête l'expansion après avoir ajouté un seul arc
```

Si la fourmi n'atteint pas le noeud cible (ie : elle atteint un cul-de-sac, elle ne déposera pas de phéromones sur son trajet). Dans l'hypothèse inverse, cependant, elle déposera une quantité de phéromone totale proportionnelle à la longueur du chemin qu'elle a parcouru (ratio entre la quantité totale de phéromone par fourmi et self.tps_de_trajet) puis un second ratio pour chaque portion de chemin afin de tenir compte de la longueur des arcs.

```
def mettre_a_jour_pheromones(self): # Version locale, après passage des fourmis
        # Calcule la qualité de la solution (inverse du temps de trajet)
        if self.tps_trajet > 0:
            ratio_pheromone = self.colonie.qte_pheromones / self.tps_trajet
            for i in range(len(self.path) - 1):
                n1 = self.path[i]
                n2 = self.path[i + 1]

                key = (min(n1, n2), max(n1, n2))
                qte = self.colonie.pheromones.get(key, 0.1) + ratio_pheromone * self.colonie.graph.get_arc(n1, n2)[2]
```

```
                self.colonie.pheromones[key] = qte

def update_pheromones(self): # Version globale, à chaque itération
        # Mise à jour des phéromones sur les arcs
        for key in self.pheromones.keys():
            self.pheromones[key] *= (1 - self.rho) if self.rho < 1 else 0.0
            ↪  # Évaporation des phéromones
```

Concernant le déplacement de la fourmi, on implémente une fonction choix_noeud et une fonction deplacement qui permettent de faire ce qu'elles disent faire, pour chaque fourmi de chaque colonie. Ce qui est caché dans l'implémentation de la mise à jour des phéromones, c'est qu'on prend également en compte la dissipation progressive des phéromones, mais ce pour l'entièreté des phéromones d'une colonie, donc ce n'est pas présent dans cette fonction propre à la fourmi, cela ne veut pas dire pour autant que ce n'est pas là tout court.

```
    def choix_noeud(self):
        voisins = self.voisins()
        if not voisins:
            return None

        q = random.random()
        if q <= self.colonie.q0:  # Exploitation
            max_val = -float('inf')
            next_node = None
            for node in voisins:
                visibility = 1.0 /
                ↪  self.colonie.graph.get_arc(self.noeud_actuel, node)[2]
                ↪  # Le poids représente le temps de trajet entre 2 noeuds
                pheromone =
                ↪  self.colonie.pheromones.get((min(self.noeud_actuel,
                ↪  node), max(self.noeud_actuel, node)), 0.1)

                # Ajout du facteur gamma pour pénaliser les arcs déjà
                ↪  utilisés par d'autres colonies
                gamma_penalty = 1.0
                for ligne_id, ligne_bus in self.colonie.lignes_bus.items():
                    if ligne_id != self.colonie.id_colony:  # Vérifie si la
                    ↪  ligne appartient à une autre colonie
                        if ligne_bus.exists_arc(self.noeud_actuel, node):
                            gamma_penalty += self.colonie.gamma  # Augmente
                            ↪  la pénalité pour chaque colonie étrangère
                            ↪  utilisant cet arc

                # Calcul de la valeur d'attractivité avec pénalité gamma
                val = (pheromone ** self.colonie.alpha * visibility **
                ↪  self.colonie.beta) / gamma_penalty
                if val > max_val:
                    max_val = val
                    next_node = node
```

```
            return next_node
        else:  # Exploration
            return random.choice(list(voisins))

    def deplacement(self):
        while not self.objectif and self.voisins():
            next_node = self.choix_noeud()
            if next_node is None:
                break

            temps_arc = self.colonie.graph.get_arc(self.noeud_actuel,
            ↪  next_node)[2] # temps == poids
            self.tps_trajet += temps_arc
            self.noeud_actuel = next_node
            self.visited.add(next_node)
            self.path.append(next_node)

            if next_node == self.noeud_cible:
                self.objectif = True

        if self.objectif:
            self.mettre_a_jour_pheromones()
```

Le code a l'air long mais il suffit de retenir que la fourmi ne peut choisir qu'entre deux comportements lorsqu'elle se trouve sur un noeud : l'exploration ou l'exploitation. L'exploration lui permet de choisir un noeud sur lequel se déplacer au hasard afin de lui permettre de peut-être trouver un meilleur chemin en "sortant des sentiers battus". Si son chemin s'avère moins bon, peu voire aucune autre fourmis ne l'empruntera et avec la dissipation des phéromones, il disparaîtra bien vite.

La fonction deplacement actualise les paramètres de la fourmi et lui permettent de progresser dans le sous graphe formé par les noeuds et les arcs couverts par sa colonie ie : sa ligne de bus. Pourquoi cela fonctionne ? Parce que lors de l'exploration, elle peut choisir un voisin du noeud sur lequel elle se trouve même si ce dernier ne se situe pas dans le sous-graphe (car la fonction get_neighbors() est dans GlobalGraph). Cela permet une bonne implémentation de l'exploration, tandis que l'exploitation est un algorithme de comparaison entre la visibilité d'un noeud et la quantité de phéromones qui se trouve sur l'arc pour s'y rendre, et ce pour tous les voisins du noeud actuel. Ici, le sous-graphe suffit puisque la fourmi ne regarde que les noeuds avec de la phéromone, et donc ce sont des noeuds pour lesquels les arêtes ont déjà été parcourues, ie : elles sont contenues dans le sous-graphe de la colonie.

Les colonies de fourmis sont paramétrées de manière à décrire leur comportement probabiliste : en modifiant ces paramètres, on aboutit à différent "caractères" de colonies, plus ou moins efficaces dans la résolution du problème et dépendant du graphe. On pourrait appliquer une méthode d'optimisation à ces caractères, pour essayer de trouver la meilleure colonie en général. Par amnque de temps, nous n'avons cependant pas fait cette étude.

Les résultats sont en accord avec la théorie : lors de la simulation d'un graphe modélisant une ville, on a une forte densité de lignes de bus sur les arrêts du centre, qui sont très desservis car statistiquement beaucoup plus utilisés, mais relativement peu en périphérie (voir par exemple 1)

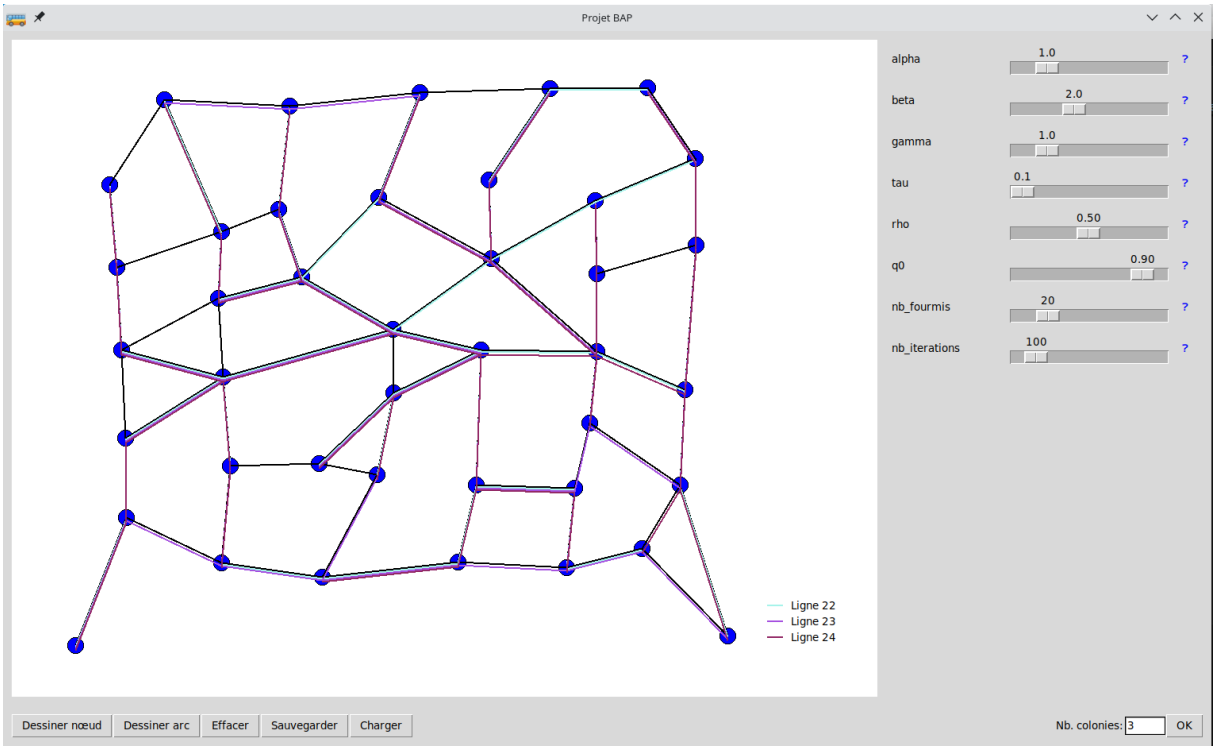

FIGURE 1 – Exemple d'exécution à 3 colonies sur un maillage dense type "ville"

4 Détails techniques

Le projet est disponible sur le gitlab de Centrale Lyon si vous y êtes autorié. M. Saïdi a été déclaré contributeur pour avoir accès au projet.

4.1 Lancement - Mode d'emploi

Pour lancer le projet, vous devez avoir Python, version 3 ou ultérieure. Téléchargez le projet, puis installez les dépendances avec

```
%> pip install -r requirements.txt
```

Vous pourrez ensuite lançer le fichier main.py à la racine du dossier, sans arguments. L'interface générale permet, au centre, de créer votre propre graphe. Si vous voulez aller vite, utilisez le bouton "charger" et sélectionnez le fichier JSON d'exemple fourni à la racine du dossier. Sinon, un clic de souris sur le canevas permet de placer des noeuds ou des arcs, selon le mode choisi par les boutons de la barre d'outils au dessous. Pour placer un arc, sélectionnez un noeud puis l'autre, et un arc non orienté sera créé. Vous avez la possibilité d'effacer la totalité du canevas pour recommencer, en utilisant le bouton dédié, ainsi qu'une fonctionnalité de sauvegarde/chargement du graphe pour exporter le graphe

créé au format JSON.

Les paramètres sont modifiables avec les sliders pour plus de facilités : pour lancer l'exécution, indiquez un nombre de colonies (c'est le nombre de lignes de bus) puis cliquez sur "OK".

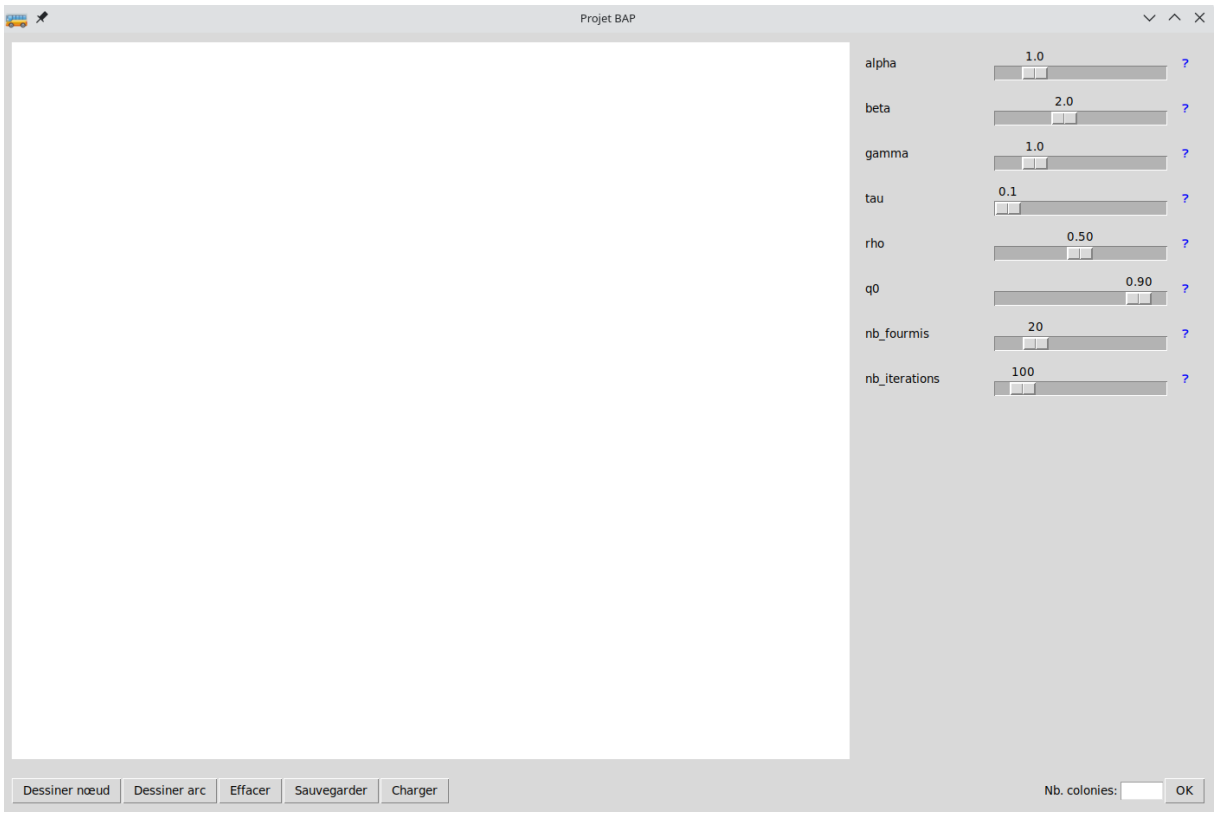

FIGURE 2 – Interface logicielle à l'ouverture

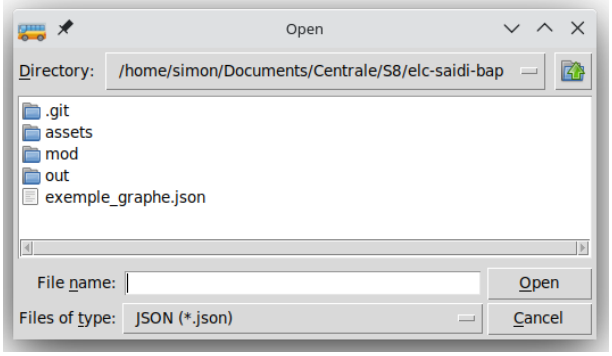

FIGURE 3 – Boite de dialogue "Charger un graphe"

À droite, les paramètres de l'optimisation par ACS sont visibles : des tooltips sont à votre disposition en cas de doute. Notez que nous avons ajouté un paramètre gamma, correspondant à l'incitation à ne pas utiliser des routes portant les phéromones de colonies concurrentes : ce paramètre sert à assurer un partage du graphe sans trop de chevauchements entre les lignes de bus.

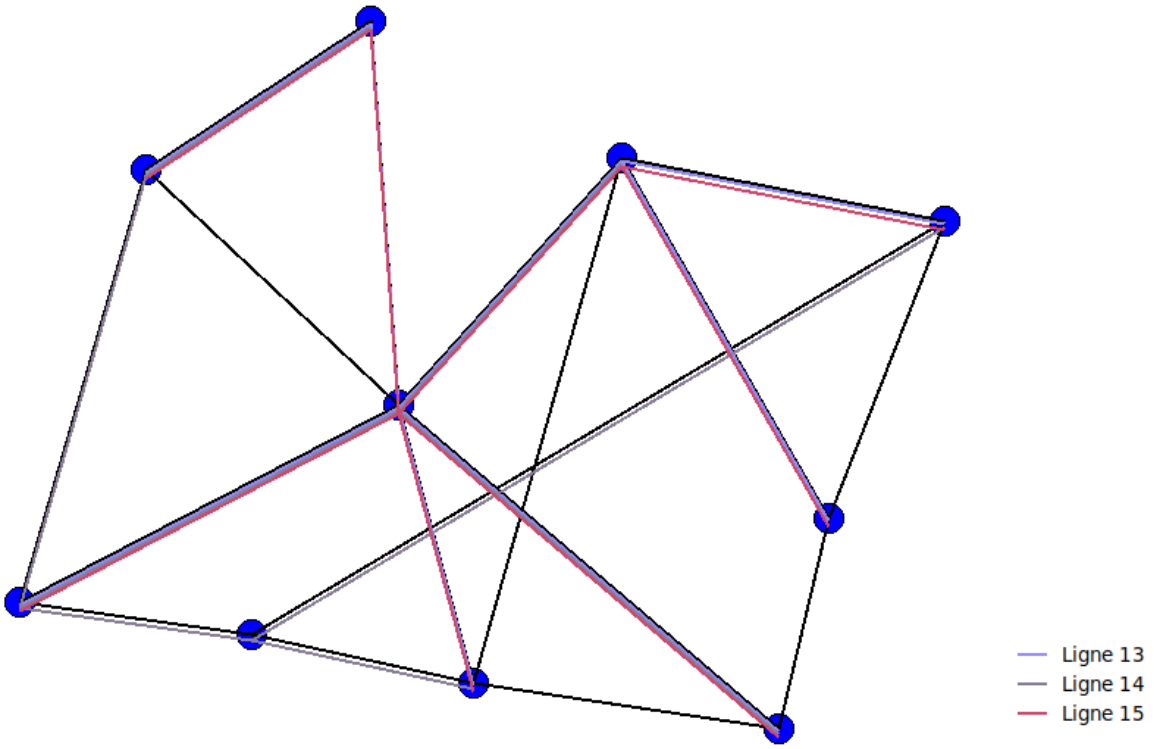

FIGURE 4 – Exemple d'exécution sur le graphe donné sur GitLab

4.2 Architecture générale

Le programme est articulé autour de quelques classes, contenues dans le package *mod* :

- Application : Fichier principal, lance l'interface et sa logique associée
- Interface : Fichier de création de l'interface
- LogiqueMetier : fichier contrôleur de l'interface, dérivé du design pattern *Model - View - ViewModel*. Sa logique n'est pas fondamentalement intéressante, il s'agit en grande partie d'outils techniques pour la manipulation des divers composants.
- BusGraph et GlobalGraph : classes principalement dédiées au stockage des données dynamiques du problèmes, représentation finale des objets de l'ACS.
- ACS : contient les classes dédiées à l'optimisation par colonies de fourmis. Ces classes formalisent la logique mathématique de la résolution par colonies.
- Optimizer : représente le problème d'optimisation. C'est une classe tampon pour diviser les tâche, qui aurait très bien pu être intégrée dans la logique métier. Cependant, nous avons jugé préférable de laisser à LogiqueMétier le soin de gérer l'interface, et Optimizer la gestion du problème. Cette classe implémente le design pattern Singleton, afin de ne pouvoir être instanciée qu'une seule et unique fois durant toute l'exécution pour éviter les conflits.
- IdCooker : Classe implémentant le design pattern Singleton (voire Factory), permettant d'attribuer un identifiant unique aux objets - utile lorsqu'il s'agit de les comparer entre eux et/ou de les représenter.

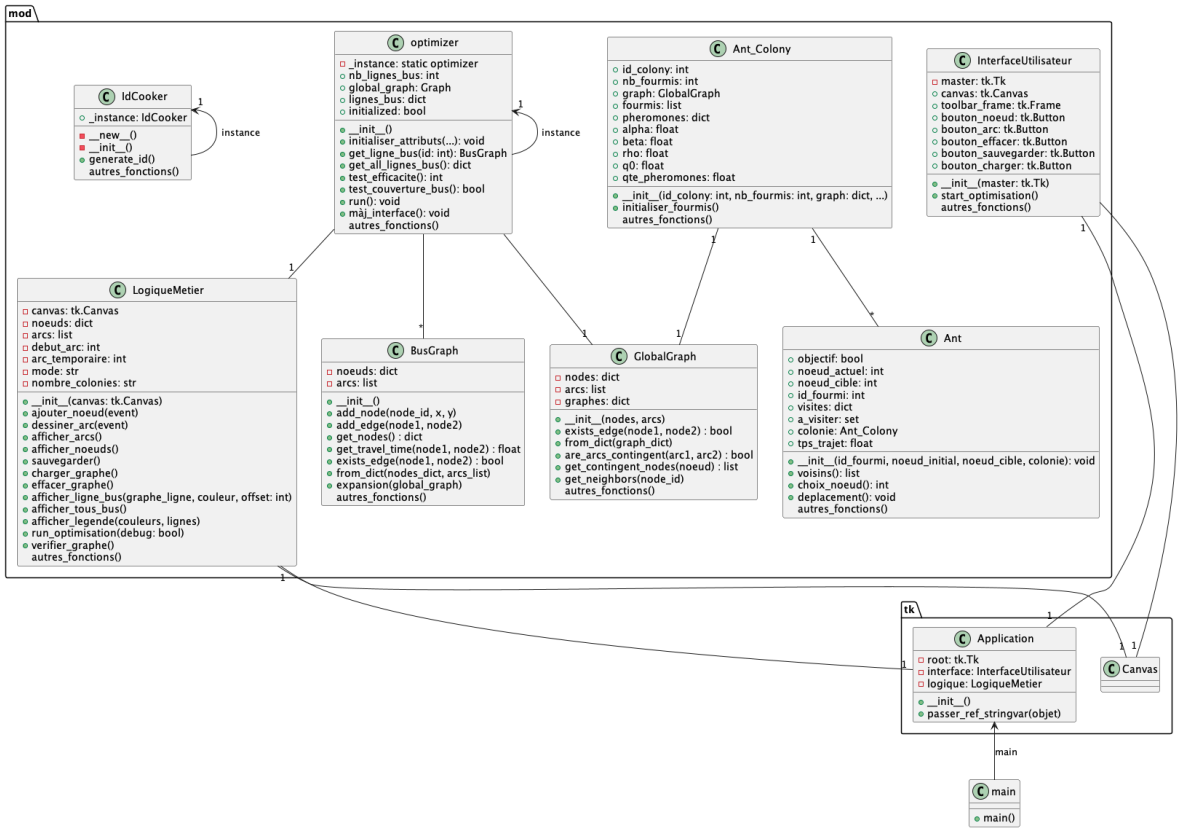

FIGURE 5 – Diagramme UML global du projet

4.3 Détails

4.3.1 Interface

Des tooltips ont été développés pour rendre l'interface plus naturelle. Ces éléments n'étaient malheureusement pas inclus dans TkInter (ou alors, bien cachés !) :

```
class Tooltip:
    def __init__(self, widget, text):
        self.widget = widget
        [...]

    def show_tooltip(self, event=None):
        if self.tooltip_window:
            return
        x, y, _, _ = self.widget.bbox("insert")
        x += self.widget.winfo_rootx() + 25
        y += self.widget.winfo_rooty() + 25
        self.tooltip_window = tw = tk.Toplevel(self.widget)
        tw.wm_overrideredirect(True)
        tw.wm_geometry(f"+{x}+{y}")
        label = tk.Label(tw, text=self.text, justify="left",
                         background="#ffffe0", relief="solid",
                         ↪  borderwidth=1,
```

```
                            font=("tahoma", "8", "normal"))
        label.pack(ipadx=1)

    def hide_tooltip(self, event=None):
        if self.tooltip_window:
            self.tooltip_window.destroy()
            self.tooltip_window = None
```

4.3.2 Identifiants

Nous avons utilisé des identifiants constamments uniques dans ce projet afin de le rendre plus robuste en cas de modifications ultérieures. Pour cela, nous avons implémenté une classe permettant de générer et retracer des identifiants uniques (un compteur), instanciable une seule et unique fois : le design pattern Singleton.

5 Conclusion

L'Ant Colony Optimization (ACO) est un choix naturel pour résoudre des problèmes complexes comme le BAP, en raison de sa capacité à explorer efficacement un grand espace de solutions combinatoires, à équilibrer exploration et exploitation, et à s'adapter à des environnements dynamiques. Cependant, d'autres techniques d'optimisation comme la recherche tabou, les algorithmes génétiques, ou même la programmation linéaire peuvent également être efficaces, en fonction de la structure du problème et des besoins spécifiques (temps de calcul, qualité de la solution, etc.).

Le choix entre ACO et d'autres méthodes dépendra des spécificités du problème, comme la taille du graphe, les contraintes à respecter, et la nature des objectifs d'optimisation (par exemple, trouver une solution optimale ou une bonne solution dans un délai raisonnable).

En ouverture, il serait possible d'optimiser les paramètres de l'optilisation par colonie avec un algorithme génétique, en prenant l'ensemble des paramètres comme phénotype pour chaque individu-colonie du problème.