Rapport AI50

Rapport du projet Drone Delivery Solution, réalisé dans le cadre de l'UV AI50 de l'UTBM. Ce rapport détaille le problème, les contraintes ainsi que les solutions apportées.

Janvier 2024
UTBM AI50

Mickaël Martin
David Goblot
Thomas Berne
Kilian Mazereau
Tobias Fresard

Table des matières

Table des figures

1 Introduction

1.1 Description générale

Le domaine de la robotique connaît actuellement des évolutions technologiques majeures, tandis que les problématiques environnementales prennent une place croissante dans notre société. Dans ce contexte, l'utilisation de drones pour la livraison de colis dans les zones à forte densité de population se présente comme une méthode alternative aux modes de livraison traditionnels qui utilisent des énergies non renouvelables et contribuent à la congestion des routes déjà très fréquentées dans les villes.

Drone Delivery Solution, notre projet AI50 a pour objectif d'optimiser la gestion d'une flotte de drones chargés d'effectuer des livraisons au sein d'une zone urbaine complexe, en prenant en compte les contraintes liées à la portée de vol limitée et à la capacité de transport des drones. Le scénario de base envisagé suppose l'existence d'un entrepôt où les colis sont stockés, chaque colis étant associé à une destination de livraison spécifique. L'ensemble des drones est chargé de transporter ces colis depuis l'entrepôt jusqu'à leur destination respective.

1.2 Problème de la livraison de drone

1.2.1 Contraintes

Nous devons respecter plusieurs contraintes dans le cadre de ce projet, dont plusieurs concernant les capacités physiques des drones :

- La batterie d'un drone n'est pas infinie, il faut prendre en compte le fait que la batterie se décharge quand le drone est en vol et qu'il doit se recharger avant de tomber en dessous d'une limite critique lui permettant de réaliser la fin de son voyage plus le retour vers une borne de recharge se situant dans les dépôts.
- De plus, il faut prendre en compte que la consommation d'énergie augmente avec le poids du colis transporté par le drone.
- Pour une simulation semi-réaliste, les drones doivent se déplacer de manière convaincante dans l'environnement. Ils doivent donc éviter les collisions avec cet environnement, mais aussi entre eux.
- Notre drone simulé doit avoir des propriétés physiques réalistes afin que nos résultats soient transposables à une utilisation réelle des drones.
- Nous devons par ailleurs réduire au maximum le nombre de drones disponibles afin de ne pas trivialiser le problème.

1.2.2 Caractéristiques du drone

Pour avoir des valeurs réalistes dans nos différentes méthodes et respecter les contraintes émises plus haut, nous avons assigné aux drones des valeurs réaliste inspiré d'un drone de livraison existant réellement[1].
Voici les caractéristiques de nos drones :

- Vitesse = 8.5 m/s
- Poids = 4 kg
- Temps d'envol = 5 s
- Temps d'atterrissage = 5 s
- Batterie utilisé pour l'envol = 1 %

- Distance maximal = 15000 m
- Temps de chargement 0%-100% = 1h30

2 Périmètre fonctionnel de l'application

Afin de pouvoir comparer plusieurs approches de ce problème d'optimisation, nous avons pensé à deux approches temporelles :

2.1 Approche planifiée

Dans cette approche simplifiée, on se place dans le cas où notre flotte de drones devra livrer un nombre défini de colis qui sont au début de la simulation, tous déjà présents dans le dépôt. Le problème sera donc de trouver quelle est l'organisation la plus optimale pour livrer l'ensemble de ces drones dans un temps minimal.

2.2 Approche en runtime

Dans cette approche, nous considérons une flotte de drones constante qui est chargée de transporter des colis générés en temps réel. C'est-à-dire que les colis sont générés aléatoirement dans une zone délimitée à des horaires répondant à une loi de probabilité correspondant à la répartition des commandes de colis sur une journée. Les colis ainsi générés doivent être livrés le plus rapidement possible en respectant les contraintes énoncées plus haut.

2.3 Mise en place

Aussi, pour une simulation réaliste, nous avons réalisé un algorithme permettant la génération de colis correspondant à des statistiques réelles. Celles-ci ne sont évidemment pas uniformes, ce qui rajoute une contrainte supplémentaire à notre projet. De plus, nous avons dû générer également une ville, pour cela, on génère des maisons à des positions aléatoires comprises dans un cercle. Ces deux générations sont enregistrées dans un fichier CSV ce qui permettra d'utiliser les mêmes données pour les différentes implémentations et solutions et les comparer.

Nous avons également fait le choix de simplification, d'autoriser les micro-chargements de batterie qui sont normalement à proscrire, car abîmant rapidement la batterie[2].

Ensuite, nous avons dû faire le choix de la disposition des zones dépôt et du nombre d'entre elles.

2.4 Dépôt unique

Même si nous avions envisagé l'implémentation de plusieurs dépôts, nous sommes finalement partis sur un dépôt unique contenant tous les colis. Ce sera aussi l'unique point de rechargement pour les drone.

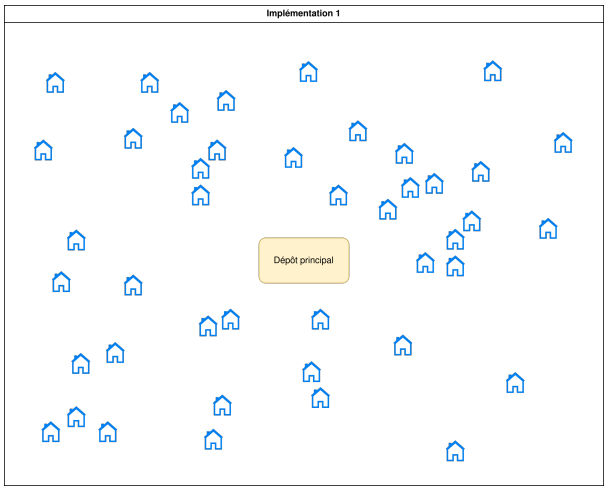


FIGURE 1 – Exemple avec un dépôt unique

3 Génération de données réalistes

3.1 Génération d'une ville

Pour que la simulation soit réaliste et puisse potentiellement être appliqué en par de véritables entreprise de livraison, il faut que les données de simulation soit réaliste. Notamment la génération de ville, avec une densité de population qui n'est pas uniforme. Pour cela, nous avons pris des données réaliste via un document de l'INSEE comprenant le nombre de ménage par carré de 200m sur 200m sur l'ensemble de la France. En prenant la ville de Grenoble, nous avons pu générer un fichier CSV avec des positions X et Y comme le montre de le tableau suivant :

Maison	X	Y
0	2459585.730814444	3984436.983484476
1	2459599.948431494	3984658.868017377
2	2459570.114416982	3984760.76513549
3	2459427.964746419	3984719.0962372837
4	2459430.0672668796	3984783.103779106
5	2459527.689261349	3984755.6272684466
6	2459424.059792156	3984780.0665168003
7	2459542.7617364004	3984627.1743303044
8	2459430.534525444	3984683.7184103183

TABLE 1 – Extrait de données de génération de ville

On a ensuite choisie arbitrairement, le point de stockage de colis affiché en rouge si-dessous :

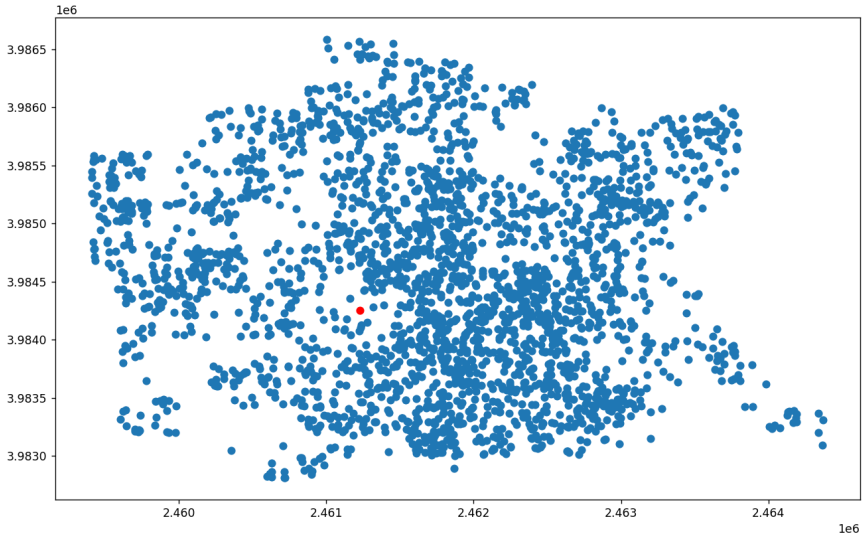


FIGURE 2 – Génération de la ville

3.2 Génération des commandes

Ensuite, le moment ou les personnes commandes leurs colis n'est pas linéaire non plus. Pour cela, nous avons également pris des données réelles. Sur l'image suivante, nous pouvons voir les heures durant lesquelles les gens commandes le plus :

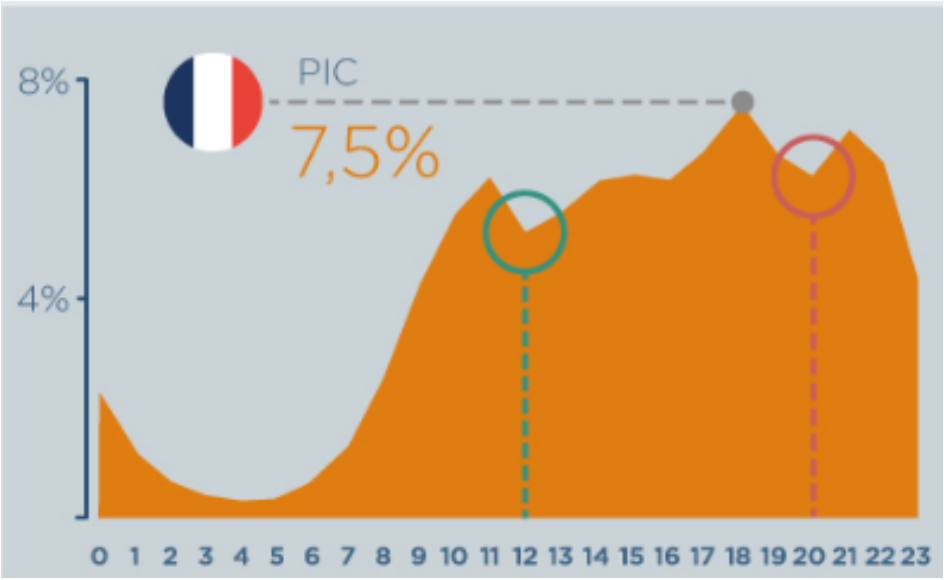


FIGURE 3 – Génération de la ville

Avec une interpolation linéaire, on déduit la probabilité pour chaque seconde d'avoir un colis commandé. On obtient un fichier comme ceci :

Commande	Heure	idMaison	Weight
0	09 :23 :05	1974	69
1	13 :05 :35	994	88
2	09 :20 :39	609	45
3	13 :36 :32	177	45
4	13 :10 :45	1084	38
5	19 :16 :08	570	71

TABLE 2 – Extrait de données de Commandes

Avec ces 2 fichiers (Ville et commande) basés sur des données réels, nous pouvons désormais faire des simulations réalistes.

3.3 Données taille réduite :

Il est nécessaire de générer des fichiers de données uniformes contenant un nombre modulable de commande afin de simplifier le processus de comparaison. Pour ce faire on prend uniformément des maisons dans le fichier contenant la ville de Grenoble afin de respecter la densité originale. On assigne ensuite un nombre de commande aléatoire entre 0 et 3 à chaque maison, ainsi qu'une heure et un poids comme dit plus haut. Ces fichiers vont être la base des comparaisons futures.

4 Solutions réalisées

4.1 Solveur optimal

La première solution envisagé pour résoudre ce problème utilise un solveur. Cette méthode nous permet de fournir une solution optimale à laquelle nous pouvons comparer les résultats des autres méthodes. Cependant, l'ajout d'un trop grand nombre de contraintes, et de paramètres, peut rendre le processus de résolution trop complexe et entraîner des temps d'exécution plus longs. Il est donc nécessaire de trouver une modélisation qui soit à la fois proche du problème de base et à la fois simple pour garantir une efficacité globale.

4.1.1 Modèle

Pour le solveur nous avons décidé d'utiliser GLPK, un kit de programmation linéaire très complet et facile à prendre en main qui permet de modéliser et résoudre des modèles mathématiques linéaires. Nous avons donc modélisé le problème de livraison par drone comme ceci :

Tout d'abord les paramètres fixes, que l'on peut changer rapidement dans le fichier DATA pour modifier les données du problèmes.

nb_p : Nombre de colis
nb_d : Nombre de drones
nb_h : Nombre de maisons
autonomy_max : Autonomie maximale des drones
recharge_time : Temps de recharge des drones
delivery_time{i} : Temps nécessaire pour livrer la maison i
orders{i} : Liste des commandes des colis

Ensuite les variables,

1. $x(p, d, h)$ qui est une variable binaire avec $x(p, d, h) == 1$ si le paquet p est livré par le drone d à la maison h.
2. $t(d)$ qui représente le temps total des livraisons du drone d.
3. $time\_max$ qui est la variable que l'on va essayer de réduire et qui représente $max(t)$

$$\text{var } x\{p \in 1..nb\_p, d \in 1..nb\_d, h \in 1..nb\_h\}, \text{ binary};$$
$$\text{var } t\{d \in 1..nb\_d\}, \text{ integer};$$
$$\text{var time\_max, integer};$$

L'objectif du modèle est donc de minimiser le temps maximal pris par un drone pour livrer ses colis, formulé comme :

$$\text{minimize } f : \text{time\_max};$$

Les contraintes du modèle sont définies comme suit, avec dans l'ordre :

1. Vérification que le paquet n'est bien livré qu'une seule fois.

2. Vérification que le paquet est bien livré au bon endroit.
3. $t(d)$ est égal a la somme de tous les voyages et rechargement du drone d.
   On multiplie par 2 car il faut compter l'aller et le retour.
4. Vérification que les voyages sont réalisables avec la capacité du drone.
5. Vérification que $time\_max$ est bien égal à $max(t)$

package_delivered_once {p in 1..nb_p} :

$$\sum_{d \in 1..nb\_d, h \in 1..nb\_h} x[p, d, h] = 1;$$

package_well_delivered {p in 1..nb_p} :

$$\sum_{d \in 1..nb\_d} x[p, d, orders[p]] = 1;$$

time_spent_flying_and_charging {d in 1..nb_d} :

$$t[d] = \sum_{p \in 1..nb\_p, h \in 1..nb\_h} ((delivery\_time[h] \cdot x[p, d, h]) \cdot (1 + recharge\_time)) * 2;$$

autonomy_not_exceeded {d in 1..nb_d, p in 1..nb_p, h in 1..nb_h} :

$$(delivery\_time[h] \cdot x[p, d, h]) \leq \text{autonomy\_max};$$

check_time_max {d in 1..nb_d} :

$\text{time\_max} \geq t[d]$; # Où time_max = max($t$)

Comme on peut le constater, l'élaboration d'un modèle linéaire entraîne beaucoup de simplification. On ne peut notamment pas intégrer une variable véritablement temporelle, on doit donc la simuler avec $t(d)$ qui suit le temps de livraison de chaque drone. De même pour la gestion de la batterie, qui est intégrée directement dans la variable $t(d)$ avec l'approximation qu'un drone va toujours se recharger jusqu'à 100% de batterie et que la décharge est linéairement proportionnelle à la distance parcouru par le drone.*

4.1.2 Données

Les données transmises aux modèles sont tirées des données réalistes décrites plus haut, voici un exemple avec 30 paquets, 15 maisons (placées entre 500m et 2km du dépôt) et 7 drones.

```
drones :

param nb_p := 30;
param nb_d := 7;
param nb_h := 15;

param autonomy_max := 1764;
param recharge_time := 5;

param delivery_time :=
1          85
2         119
...
14         65
15        104;

param orders :=
1          1
2          9
...
29         5
30         9;
```

On note donc que les distances sont représentées par le temps que prend le drone pour parcourir la distance dépôt-maison.
Dans ce projet, nous avons fait le choix de lancer le solveur avec des données allant de 10 à 50 colis (de 10 en 10), et de 1 à 10 drone. Cela nous permet de réaliser des visualisations pertinentes tout en gardant le temps d'exécution dans une limite raisonnable.

4.1.3 Exécution

Avec les différents paramètres de lancement du solveur glpsol de GLPK, nous pouvons définir une limite de temps de calcul, élément nécessaire car l'espace de recherche devient extrêmement grand lorsque qu'on augmente le nombre de drone, de colis et de maison.
Nous pouvons aussi enregistrer les résultats dans des fichiers .log. Cela va nous permettre de réaliser des visualisations et ainsi comparer les différentes approches entre elles.
Nous allons donc utiliser la commande suivante :

```
$ glpsol --model solver_drone.mod --data solver_drone_data_p_d.dat --output
  solver_drone_solution_p_d.log --log solver_drone_cmd_output_p_d.log
  --tmlim (p*5+d*5)
```

Avec p le nombre de colis, d le nombre de drone et le temps d'exécution accordé une fonction de p et d

On enregistre donc chaque combinaison dans deux fichiers de résultat, l'un avec la solution complète donnée par glpsol (*solver_drone_solution_p_d.log*).
L'autre avec l'enregistrement de l'affichage terminal (*solver_drone_cmd_output_p_d.log*).

4.1.4 Exemple de résultat

Nous prenons ici les résultats de glpsol avec 10 colis et 4 drones.
Avec de simple commande d'affichage, nous obtenons un résumé des livraison sur le terminal :

```
The package 1 is delivered by the drone 2 to the house 1
The package 2 is delivered by the drone 2 to the house 1
The package 3 is delivered by the drone 2 to the house 1
The package 4 is delivered by the drone 3 to the house 2
The package 5 is delivered by the drone 4 to the house 2
The package 6 is delivered by the drone 4 to the house 2
The package 7 is delivered by the drone 2 to the house 3
The package 8 is delivered by the drone 4 to the house 4
The package 9 is delivered by the drone 3 to the house 6
The package 10 is delivered by the drone 4 to the house 7
The package 11 is delivered by the drone 3 to the house 8
The package 12 is delivered by the drone 3 to the house 8
The package 13 is delivered by the drone 1 to the house 9
The package 14 is delivered by the drone 1 to the house 10
The package 15 is delivered by the drone 1 to the house 10
```

Également la solution final, la solution optimale théorique, et le pourcentage de différence entre les deux :

```
+165221: mip =    2.848000000e+03 >=    2.819000000e+03    1.0% (10927; 27167)
TIME LIMIT EXCEEDED; SEARCH TERMINATED
Time used:    25.0 secs
Memory used: 19.4 Mb (20313034 bytes))
```

Après 25 secondes d'exécution (temps limite), nous obtenons donc un *time_max* de 2848 secondes, soit un écart d'environ 1% à la solution optimale théorique.

Le fichier *solver_drone_solution_10_4.log* nous permet d'obtenir la solution complète et notamment le temps de livraison et de rechargement de chaque drone :

$t[1]$	$*2840$
$t[2]$	$*2796$
$t[3]$	$*2792$
$t[4]$	$*2848$
time_max	$*2848$

C'est aussi un moyen de noter l'efficacité du solveur, car les drones ont bien des temps de livraisons similaires, ce qui montre qu'ils sont tous utilisés équitablement et de façon optimale.

4.1.5 Pistes d'améliorations

La première piste d'amélioration évidente est la modification du calcul du temps afin de ne pas recharger à 100% un drone qui revient au dépôt.

L'exploration et l'expérimentation des différents paramètre de résolution de glpsol pourrait aussi amener à de meilleurs résultats.
Enfin l'implémentation GLPK ne prend pas en compte le poids du colis dans son calcul de batterie et/ou de vitesse, qu'il faudrait ajouter avec une variable supplémentaire à ajouter dans les données.

4.2 Recuit simulé

On a également implémenté l'algorithme d'optimisation du recuit simulé afin d'obtenir une solution à comparer dans le cas où la méthode de livraison des colis serait au jour par jour et non en runtime.

On considère que le cycle de vie d'un colis commence par l'attente d'un drone libre, puis l'attente possible de la charge de ce drone afin d'effectuer la livraison, et enfin sa livraison effective. Le temps du retour du drone n'est pas pris en compte, car il est déjà pris en compte dans le temps d'attente de la libération d'un drone.

A partir de ces spécifications, nous avons pu mettre en place une implémentation en python du recuit simulé pour résoudre le problème. On retrouve les classes suivantes et leurs fonctions :

4.2.1 SimulatedAnnealing class

Cette classe est responsable d'appliquer l'algorithme du recuit simulé avec un refroidissement exponentiel. Son constructeur prend en paramètre la valeur initiale pour la température, par défaut équivalent à 1000°. Il prend aussi en paramètre le facteur de refroidissement, généralement entre 0.9 et 0.99, c'est lui qui dicte à quelle vitesse la température décroît.

La méthode 'solve' utilise l'algorithme de recuit simulé pour optimiser un problème spécifique donnée en paramètre. Elle effectue plusieurs type de recherche pour explorer et exploiter l'espace de recherche au mieux.

- Recherche en profondeur : Cette recherche vise à exploiter la solution actuelle pour l'améliorer. Elle s'appuie sur des heuristiques (qui dépendent du problème à résoudre) que nous détaillerons plus tard.

- Recherche locale : Avec cette recherche, on explore l'espace proche de la solution actuelle.

- Recherche aléatoire : La recherche aléatoire vise à explorer au hasard l'espace de recherche, elle permet d'éviter à l'algorithme de rester bloqué si jamais sa solution de départ était mauvaise.

4.2.2 Order class

Cette classe représente une commande avec des attributs tels que la destination, la distance de livraison, l'heure de la commande et le poids du colis à livrer.

4.2.3 Drone class

Elle est responsable de modéliser un drone avec des caractéristiques telles que la vitesse, le poids, le temps de décollage/atterrissage, et la capacité de la batterie.

On y retrouve aussi les méthodes permettant de calculer le temps nécessaire pour charger la batterie et le temps nécessaire pour effectuer une livraison.

```python
def battery_needed(self, order : Order):
    # distance max with with this particular package
    distance_max_with_package = Drone.distance_max*(Drone.weight+order.weight)/
        Drone.weight
    percentage_battery_needed = order.distance*100/distance_max_with_package +
        order.distance*100/Drone.distance_max
    # takeoff and landing
    percentage_battery_needed += (Drone.weight+order.weight)*Drone.
        takeoff_ratio + Drone.takeoff_battery_loss
    # return percentage rounded to the superior integer
    return int(percentage_battery_needed+0.99)
```

FIGURE 4 – Calcul de la batterie nécessaire

4.2.4 Action class

La classe Action est un composant de la solution, elle représente une action qu'un drone doit effectuer, définie par la quantité de batterie à charger et la commande à livrer.

4.2.5 DeliveryProblem class

Elle modélise le problème de livraison de colis, avec des drones, des commandes, et des méthodes pour générer, évaluer et modifier des solutions notamment en créant des voisins de la solution actuelle. Ces voisins sont obtenus en modifiant ou en intervertissant certains paramètres de la solution actuelle comme la batterie ou en assignant une action à un autre drone.

```python
else : # Modifying battery of an action
        drone_index = np.random.randint(0, self.nb_drones)
        while(len(neighbor[drone_index]) == 0) :
            drone_index = np.random.randint(0, self.nb_drones)
        action_index = np.random.randint(0, len(neighbor[drone_index]))
        neighbor[drone_index][action_index].battery_to_charge += np.random.
            randint(-10,0)

    return self.rectify(neighbor)
```

FIGURE 5 – Modification de la batterie pour créer un voisin

Cette classe permet également de corriger des solutions irréalisable, pouvant survenir lors de la génération des voisins. Pour rectifier ces erreurs, la méthode met à jour les valeurs de batterie à recharger afin que le drone puisse effectuer la livraison.

4.2.6 Améliorations possibles

- On pourrait améliorer le recuit simulé en ajoutant une heuristique qui inciterait le programme à ne pas réutiliser le même drone plusieurs fois à la suite : on pourrait par exemple donner une valeur d'utilisation à un drone qui s'il vient à être utilisé plusieurs fois en peu de temps, augmenterait et diminuerait ainsi la probabilité qu'il soit choisi par le programme comme livreur suivant.

- Une autre façon d'améliorer le recuit simulé serait d'intégrer une part d'erreur fictive des drones telles que des pannes.

- La gestion de la batterie des drones pourrait être améliorée en affectant aux drones ayant peu de batterie des livraisons proches.

4.3 Système Multi-agents

4.3.1 Introduction

L'une des approches envisagées pour notre projet repose sur l'utilisation d'un système multi-agents. Ce concept repose sur la collaboration intelligente de plusieurs entités autonomes, appelées agents, qui interagissent pour atteindre des objectifs communs et qui est utilisé dans le domaine du transport [3].

L'idée fondamentale derrière le système multi-agents est de permettre une prise de décision distribuée, adaptative et en temps réel. Chaque agent est capable de percevoir son environnement, de prendre des décisions autonomes basées sur ces informations, et de coopérer avec d'autres agents pour optimiser le processus de livraison. Cette approche présente des avantages significatifs en termes d'efficacité opérationnelle, de résilience et d'adaptabilité aux changements dynamiques dans l'environnement.

4.3.2 Diagrammes de conception

Lors de la conceptualisation du programme, nous avons créé deux diagrammes : un diagramme de classes ainsi qu'un diagramme de séquences.

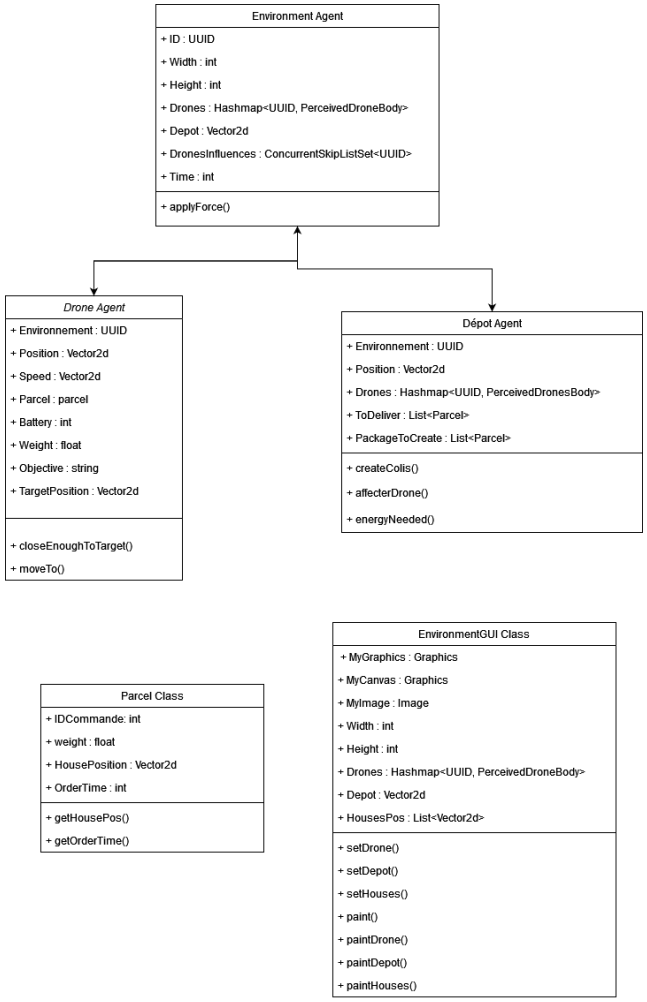


FIGURE 6 – Diagramme de Classe

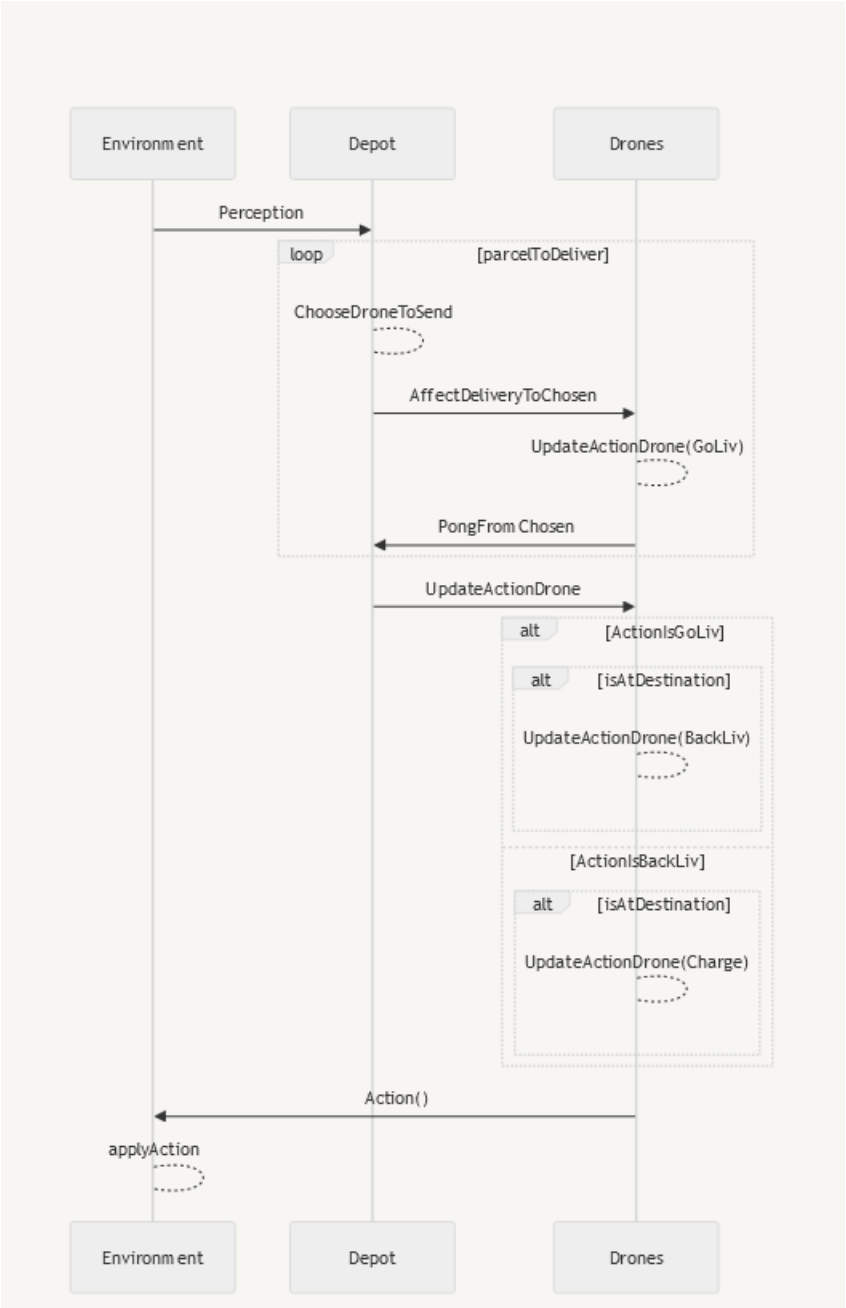


FIGURE 7 – Diagramme de Séquence

4.3.3 PEAS

Pour nous aider dans le dévellopement du système multi-agents, nous avons également fait un diagramme PEAS :

Environnement	Performance	Coût	Actionneurs	Capteurs
— Totalement observable — Stochastique — Séquentiel — Dynamique — Continue	— Satisfaire toutes les maisons le plus rapidement possible — Écart max entre le temps de livraison théorique et le temps de livraison effectif	Énergie	— Prendre un colis — Bouger / Voler — Déposer un colis — se charger	— Accéléromètre — GPS — Chronomètre

Au vu de ces paramètres, nous avons opter pour approche Reactive Agent.

4.3.4 Fonctionnement général du code

Ainsi, nous avons pu développer une solution basée sur un système multi-agent Sarl dont voici le fonctionnement principal :

1. Plusieurs agents aux rôles différents : Environnement, Dépot et Drones. L'environnement tient le dépôt informé des positions de tous les drones ainsi que leurs états respectifs. Ces informations prises en compte, l'agent de dépôt choisira le meilleur drone pour la livraison à venir (si livraison il y a) et lui affectera la tâche de livraison. Les drones compareront alors leurs objectifs à réaliser avec leur état actuel et décideront de la marche à suivre pour demander à l'environnement de mettre leur position à jour en fonction.
2. Le dépôt choisira comme meilleur drone de livraison un drone disponible, dont la batterie est suffisante et si possible la plus faible pour la livraison afin de laisser d'autres drones charger pour des livraisons plus lointaines.
3. Les drones peuvent avoir trois états différents : Charge, GoLiv (en chemin pour la livraison) et BackLiv (en retour de la livraison). Ces états sont mis à jour par le dépot pour les drones disponibles en chargement, et par les drones eux-mêmes pour ceux qui seraient indisponibles, en déplacement. Exemple : un drone dans l'état GoLiv passera en l'état BackLiv si son colis à été livré lors de cette perception.
4. Par soucis de réalisme, la batterie de chaque drone est mise à jour à chaque nouvelle perception. Une charge complète permet de voler 30 minutes en vitesse de croisière et un chargement sera considéré comme complet après une heure et demie de charge si vide initialement.
5. Un paquet 'Parcel' (car Package est déjà défini et ne peut pas être utilisé comme variable) est défini par la position de la maison de destination, un poids de 0.1 à 2kg, un temps -heure de la journée où il à été commandé- et un Id de commande. Le paquet ou Parcel est considéré comme livré si la position du drone est égale à la position de la maison de destination.
6. L'affichage en temps réel du déroulement des actions est pris en charge par le fichier 'EnvironmentGUI' qui actualisera l'affichage à chaque nouvelle perception.
7. La liste des colis à livrer est récupérée d'un fichier Csv qui contient des données réalistes.

4.3.5 Améliorations

Pour améliorer la solution obtenue par le système multi-agent, il serait intéressant de le faire fonctionner pour une période de temps supérieure à une journée (une semaine ou plus). Dans ce cas de figure, le principe du programme en temps réel prendrait plus de sens.

5 Visualisation

Pour les deux approches planifiées, nous avons réalisé une visualisation en utilisant la bibliothèque Python Tkinter. En structurant chaque drone et maison par le biais de classes distinctes, cependant pour faciliter l'affichage un redimensionnement a été nécessaire.

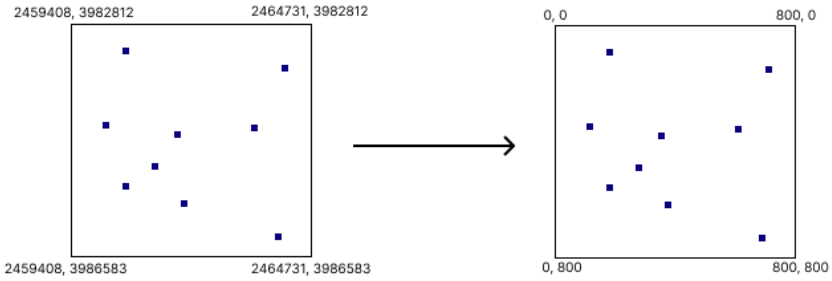


FIGURE 8 – Redimensionnment

Ensuite en exploitant les fonctionnalités offertes par Tkinter, nous avons obtenu le résultat suivant :

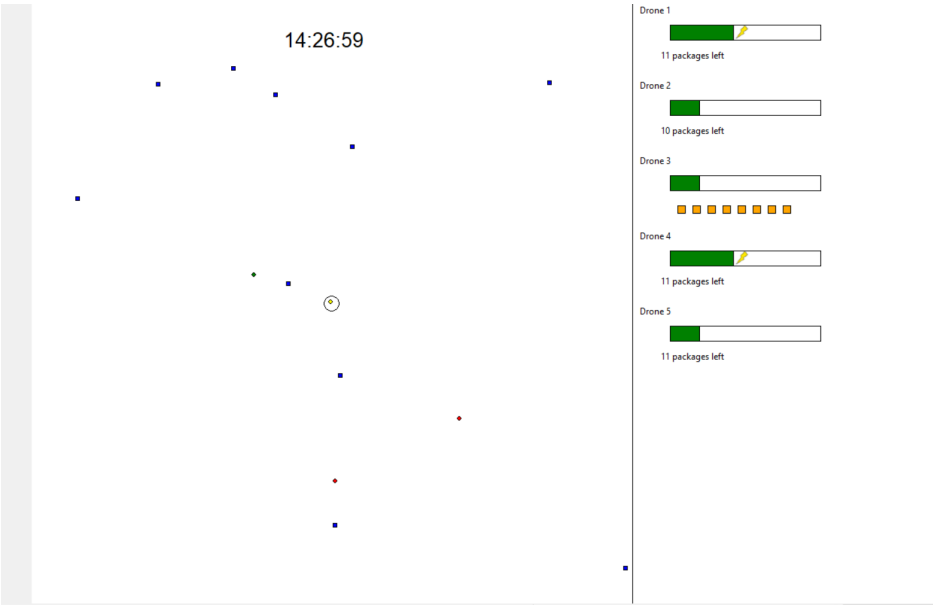


FIGURE 9 – Exemple de visualisation avec un nombre réduit de maison

Seulement même si la base est la même, certaines adaptations ont dû être apporté pour le GLPK et le recuit simulé.

5.1 Visualisation GLPK

Étant donnée que la solution GLPK est calculée en partant du principe que le drône doit être charger intégralement avant de repartir du dépôt, il est possible à partir de la solution retournée par le solveur d'optimiser la solution, en gardant la répartition et l'ordre mais en faisant en sorte que le drône se charge uniquement de ce qu'il a

besoin et uniquement lorsque c'est nécessaire (avec une une marge de 20%). Cette optimisation a donc directement été faite dans la visualisation. La batterie nécessaire est déduit via des coefficients liés au changement d'échelle.

```
1               distance = norm(self.objectif[0] - self.depot[0],self.objectif[1] -
                    self.depot[1])*6.2
2               batteryNeeded = (distance/150)*2
```

FIGURE 10 – Calcul de la batterie nécessaire

5.2 Visualisation Recuit simulé

Pour le recuit simulé, la solution retourné donne la batterie nécessaire pour le drône puisse repartir du dépôt, c'est donc une fonctionnalité qui a du être implémenter en plus par rapport à la solution donnée par le solveur GLPK. Pour cela, chaque drône a une liste contenant la batterie nécessaire pour chaque colis. On peut ensuite vérifier que le drône a assez de batterie avec le code suivant :

```
1       if self.battery <= self.listBattery[self.numberPackage - len(self.
            listPackage)] and self.atSpawn():
2               self.charge()
```

FIGURE 11 – Vérification de la batterie

5.3 Visualisation SARL

Puisque SARL est en temps réel, une visualisation différente a été réalisé, elle basé sur l'exemple boids et a été adapté à nos besoins.

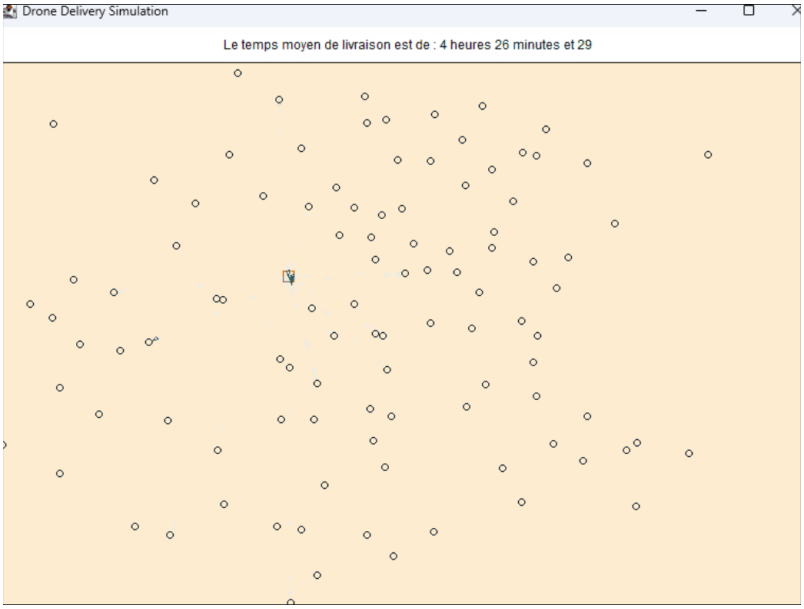


FIGURE 12 – Visualisation SARL

6 Résultats et comparaison

6.1 Analyse

Grâce aux différents solutions implémentées, on peut tester sur un même ensemble de données l'efficacité de la solution. Ainsi, on peut obtenir des graphiques qui représentent facilement la courbe d'efficacité.

Ce graphique est obtenu avec les données du solveur exact Glpk. On observe que les données sont cohérentes dans l'ensemble et une ligne stable semble se distinguer et former une moyenne commune entre toutes les solutions.

La limite pour avoir une solution satisfaisante en termes de temps de livraison peut donc être calculée comme un rapport moyen entre les nombres de drones et le nombre de colis, approximée à 10 colis par drone.

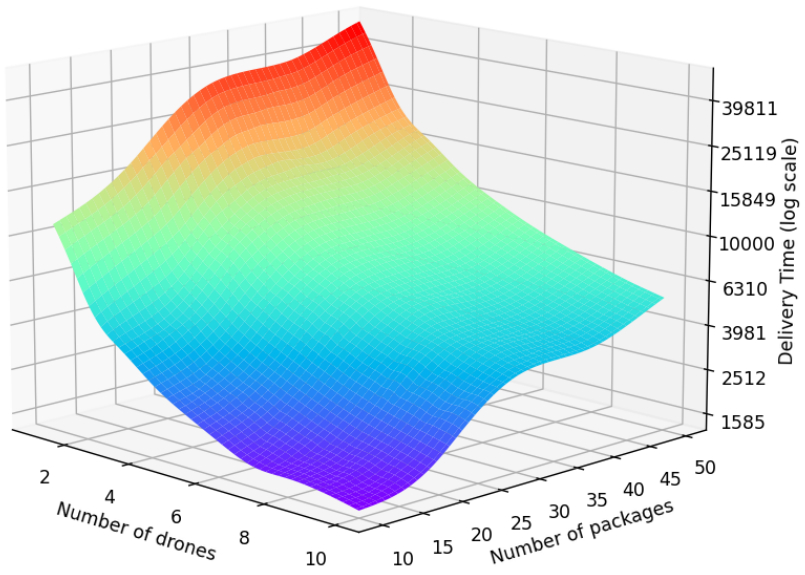


FIGURE 13 – Visualisation du temps de livraison obtenu grâce à la méthode Glpk en fonction du nombre de drones et de colis

De la même façon que pour le premier graphique, avec ce graphique tiré des données du recuit simulé, on observe une moyenne facilement observable et une cohérence des résultats à une exception près. La limite de satisfaction peut être ici approximée à 8 colis par drone.

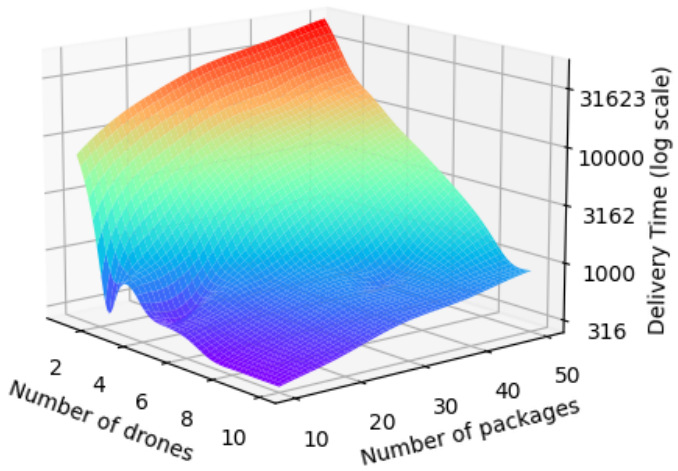
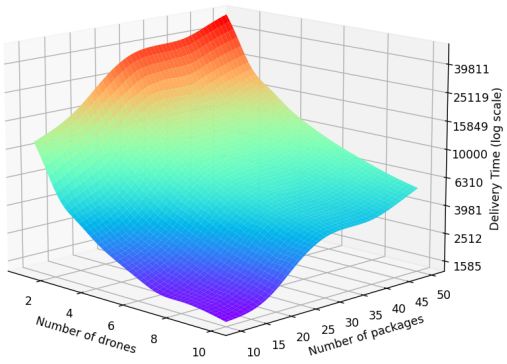
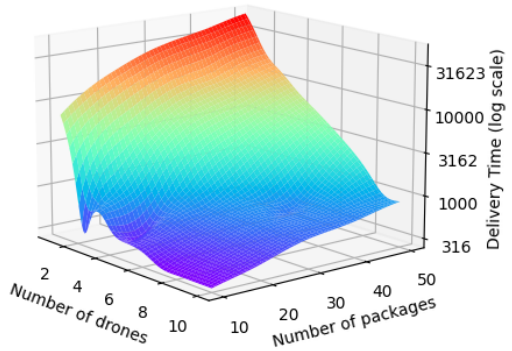


FIGURE 14 – Visualisation du temps de livraison obtenu grâce à la méthode du recuit simulé en fonction du nombre de drones et de colis

6.2 Comparaison



(a) Glpk



(b) Recuit simulé

On remarque plus de variation dans le recuit simulé lié à l'aléatoire de l'algorithme. Le GLPK est plus régulier lorsque le nombre de colis est augmenté ou le nombre de drône diminue.

7 Difficultés rencontrées

Lors de ce projet nous avons rencontré de nombreuses difficultés :

— SARL et GLPK, deux technologies très présentes dans le projet, sont des technologies nouvelles pour nous. De plus nous avons du apprendre à utiliser GLPK sans cours car la programmation linéaire de modèle mathématiques n'est pas au programme du bloc AI, ce qui a demandé une bonne période d'apprentissage.

— GLPK notamment à entrainé beaucoup de simplification car il nécessite la modélisation d'un problème linéaire. Nous avons par exemple eu du mal à implémenter la prise en charge de la batterie, et le résultat ne nous convient pas parfaitement.

— Les différents algorithmes doivent utiliser les mêmes données pour pouvoir être comparer, l'echelles de ces dernières à donc été revu à la baisse afin que les solutions de GLPK soit générées dans un temps convenable.

— Ce projet à été réalisé lors de notre dernier semestre d'étude, tous les membres du groupe ont donc du effectuer des recherches de stages et d'hébergement ainsi que des projets hors emploi du temps en parallèle des cours. Cela a entraîner un semestre très chargé personnellement et académiquement.

Références

[1] Société DPD. L'innovation au service des populations difficiles d'accès. *dpd.com*, 2023.

[2] Youngmin Choi and Paul Schonfeld. Optimization of multi-package drone deliveries considering battery capacity. *ResearchGate*, 2017.

[3] Shushman Choudhury. Multi agent allocation transit. Technical report, 2020-11-17.