Projet de SOD321

Une Course D'Avion

Raphaël BERNAS & Maxime CORLAY

octobre 2024

Table des matières

1 Introduction

L'objectif de ce projet est la mise en application du cours SOD321 à un problème d'optimisation sur un parcours de graphe.

Le code implémenté pour ce projet peut être trouvé ici *(tous les résultats sont obtenus avec un CPU AMD Ryzen 7 5800H)* :

https://github.com/Raphael-Bernas/

Pour donner un peu plus de contexte à ce problème, nous imaginons une course d'avion. Les concurrents de cette course seront évalués par une fonction objectif (nous supposerons ici qu'ils cherchent à réduire cette fonction pour réussir la course). Cependant plusieurs conditions sont requises pour valider la course :

- **Contrainte de visites :** Les aviateurs devront passer par au moins A_{min} aérodromes.
- **Contrainte de distance :** Les avions ont une distance maximale R qu'ils peuvent parcourir sans se poser.
- **Contrainte de passage par des régions :** Les concurrents devront passer par au moins $N_{regions}$.
- *Contraintes sous-jacentes :* (ces contraintes sont implicites au problème) Les aviateurs doivent décoller et réatterrir. Leurs déplacement ne peuvent, bien évidemment pas, être discontinus dans le graphe des aéroports.

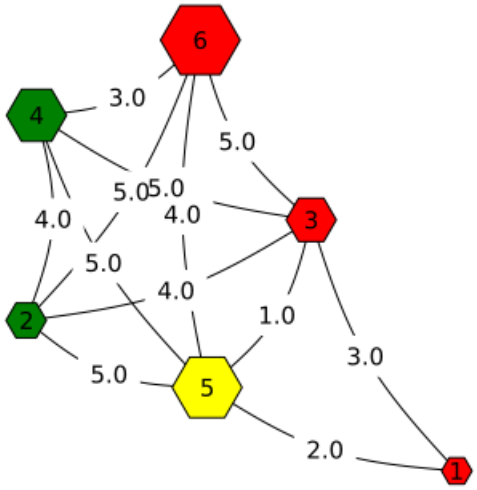

FIGURE 1 – Graphe simulant le problème d'aviation pour 6 aérodromes et 3 régions.

2 Modélisation du problème

Cas n°1 : on passe au plus une fois par chaque aérodrome :
Fonction objectif :

$$\min_{x_{ij}} \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} d_{ij} \tag{1}$$

On répartit les contraintes en **cinq** catégories :
— Contraintes de base
— Contraintes d'envol et d'atterrissage
— Contraintes de A_{min} et R
— Contraintes de connexité
— Contraintes de passage par **toutes** les régions

2.1 Contraintes de base

Contraintes sur la formulation du problème : présence/absence

(*contrainte présence-absence*) $x_{ij} \in \{0,1\}$ pour tous $i, j \in [|1,n|]$

2.2 Contraintes d'envol et d'atterrissage

Contrainte pour le départ du premier aérodrome et pour l'arrivée à la fin de course.

(*Partir de d*) $$\sum_{j=1}^{n} x_{dj} = 1$$

(*Arrive en f*) $$\sum_{i=1}^{n} x_{if} = 1$$

Pour tout $k \in [|1,n|] \setminus \{d; f\}$,

$$\sum_{i=1}^{n} x_{ik} \leq 1$$

La contrainte d'envol-atterrissage est une contrainte de **conservation du flot**

(*conservation du flot*) $$\sum_{j=1}^{n} x_{kj} = \sum_{i=1}^{n} x_{ik}$$

Dans le cas général où le point de départ est s et le point d'arrivée est t, toutes ces contraintes s'écrivent :

(*Partir de s*) $$\sum_{j=1}^{n} x_{sj} = 1$$

(*Arrivé en t*) $$\sum_{i=1}^{n} x_{it} = 1$$

Pour tout $k \in [|1,n|] \setminus \{s; t\}$,

$$\sum_{i=1}^{n} x_{ik} \leq 1$$

(*conservation du flot*) $$\sum_{j=1}^{n} x_{kj} = \sum_{i=1}^{n} x_{ik}$$

2.3 Contrainte A_{min} et R

Pour tout sommet i, $\sum_{j=1}^{n} x_{ij}$ représente le nombre de décollages depuis i (qui vaut 0 ou 1 dans notre cas, car il y a max un décollage par aérodrome). Donc $\sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij}$ représente la totalité des décollages. Si on lui ajoute 1, on obtient le nombre d'aérodrome parcourus, d'où la **contrainte A_{min}** :

$$(A_{min}) \qquad \sum_{i=1}^{n}\sum_{j=1}^{n} x_{ij} + 1 \geq A_{min}$$

De plus, les trajets sont $\leq R$, donc :

$$x_{ij} d_{ij} \leq R$$

pour tout i, j

2.4 Contrainte de connexité

Cette contrainte n'est pas évidente à modéliser. En effet, plusieurs propositions ont été faites pour proposer différents types de contraintes. Dans cette partie introductive, nous présentons une contrainte polynomiale : la **MTZ**. Exponentielle : la **DFJ**.

— Polynomiale (**contrainte MTZ** [1])
On ajoute des $u_i \in \mathbb{N}$ tels que pour tout $i, j \in [|1, n|]$:

$$u_j \geq u_i + 1 - n(1 - x_{ij})$$

Remarque : contrairement au cours et au papier, où on demandait que $i \neq s$ (i différent du point de départ) et $j \neq t$ (j différent du point d'arrivée), ici pas besoin de demander cela car on empêche TOUT cycle (donc on prend juste $i, j \in [|1, n|]$).

— Exponentielle (**contrainte DFJ** [2])
Pour tout S inclus dans V tel que $1 \leq |S| \leq |V| - 1$,

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1$$

2.5 Contraintes de passage par toutes les régions

Pour tout aérodrome $i \in [|1; n|]$, $regions[i] \in [|1; N_r|]$ désigne la région dans laquelle le $i^{ème}$ aérodrome se trouve. On veut que pour tout $r \in [|1; N_r|]$:

$$\sum_{i \,:\, region_i = r} \sum_{j \in [|1,n|} x_{ij} + x_{ji} \geq 1$$

3 Expérience et comparatif MTZ-DFJ

Maintenant, nous allons implémenter en $Julia$ notre problème et le résoudre en utilisant le package $Gurobi$.

1. MTZ = Miller-Tucker-Zemlin
2. DFJ = Dantzig-Fulkerson-Johnson

L'implémentation des deux modélisations va nous permettre de comparer l'efficacité de ces deux méthodes. Nous nous attendons à ce que pour de grandes valeurs de n (= nombre d'aéroports), les contraintes MTZ soient bien meilleures que les contraintes DFJ.

3.1 Implémentation

Le pseudo-code de l'algorithme de résolution que nous allons utiliser est décrit dans Algorithm 1.

3.2 Expérimentation pour $n = 6$

Résultats de la méthode DFJ : Voir Figure 2

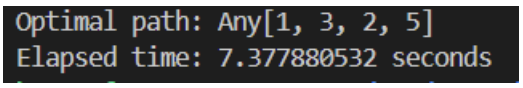

FIGURE 2 – Résultats de la méthode DFJ pour l'instance $n = 6$.

Résultats de la méthode MTZ : Voir Figure 3

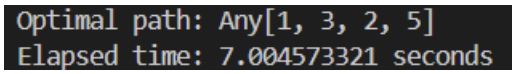

FIGURE 3 – Résultats de la méthode MTZ pour l'instance $n = 6$.

Algorithm 1: Résolution du problème d'aviation méthode DFJ/MTZ

Input : Nom du fichier de données `file`
Output : Chemin optimal

Initialisation :
Lire les données de l'instance depuis `file`
$n, d, f, Amin, Nr, R$, regions, coords, $D \leftarrow$ données de l'instance
Création du modèle d'optimisation :
Définir les variables de décision $x[i, j]$ binaires pour chaque paire (i, j)
Minimiser $\sum_{i=1}^{n} \sum_{j=1}^{n} D[i, j] \cdot x[i, j]$
Définition des contraintes :
Imposer que le départ soit de d et l'arrivée en f
for $k \in \{1, \ldots, n\} \setminus \{d, f\}$ **do**
    Limiter à un au plus le nombre de départs et arrivées de chaque nœud k
    Assurer le flux entrant égal au flux sortant pour chaque nœud k
Imposer que la somme des variables $x[i, j]$ soit au moins $Amin - 1$
for $r \in \{1, \ldots, Nr\}$ **do**
    Imposer qu'au moins une connexion passe par chaque région r
for $i, j \in \{1, \ldots, n\}$ **do**
    Imposer que $x[i, j] \cdot D[i, j] \leq R$
if ***DFJ*** **then**
    Ajout des contraintes DFJ :
    for *S sous-ensemble de nœuds où* $1 \leq |S| \leq n - 1$ **do**
        Limiter la somme des $x[i, j]$ pour $(i, j) \in S$ à $|S| - 1$
else
    Définition des variables MTZ :
    Définir les variables $u[i]$ entières pour chaque nœud $i \in \{1, \ldots, n\}$
    Ajout des contraintes MTZ :
    for $i \in \{1, \ldots, n\}$ **do**
        for $j \in \{1, \ldots, n\}$ **do**
            Ajouter la contrainte $u[j] \geq u[i] + 1 - n \cdot (1 - x[i, j])$
Résolution du modèle :
Résoudre le modèle d'optimisation
Récupération des résultats :
if *le modèle est optimal* **then**
    Extraire la solution $x[i, j]$
    Suivre le chemin optimal en partant de d jusqu'à f
    return *Chemin optimal*
else
    return *Aucune solution optimale trouvée*

On observe en effet que la méthode MTZ est plus rapide que la méthode DFJ, cependant la différence est relativement faible par rapport à ce que l'on aurait pu attendre. Le chemin optimal obtenu est bien le même : $d = 1, 1 \rightarrow 3, 3 \rightarrow 2, 2 \rightarrow 5 = f$.

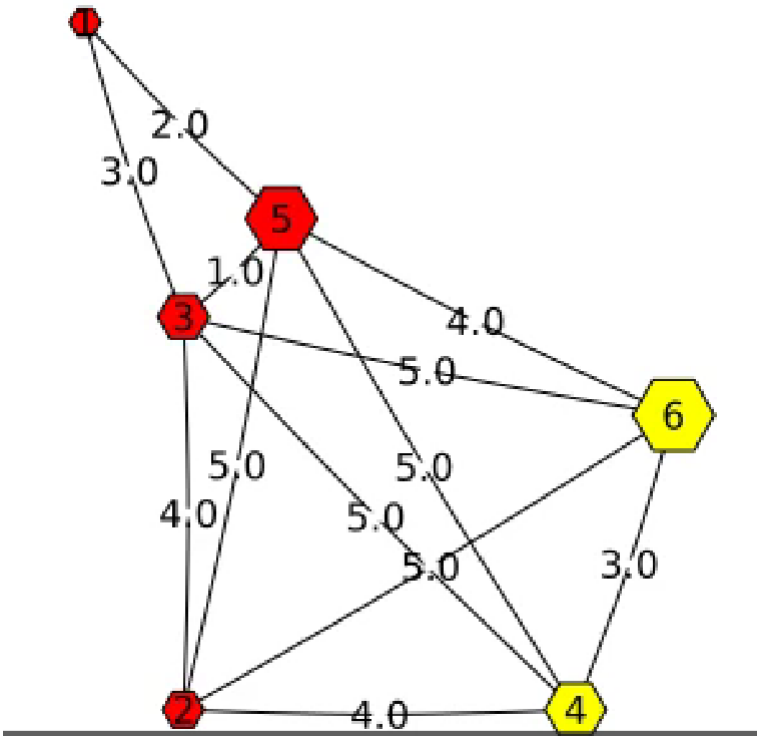

FIGURE 4 – Chemin parcouru par l'avion colorée en rouge (Vidéo de la traversée).

3.3 Expérimentation pour $n = 10$

Résultats de la méthode DFJ : Voir Figure 5

```
Optimal path: Any[1, 5, 4, 6]
Elapsed time: 7.688472285 seconds
```

FIGURE 5 – Résultats de la méthode DFJ pour l'instance $n = 10$.

Résultats de la méthode MTZ : Voir Figure 6
Pour voir la traversée en $n = 10$: (Vidéo de la traversée)
La différence se creuse entre MTZ et DFJ.

```
Optimal path: Any[1, 5, 4, 6]
Elapsed time: 6.77618621 seconds
```

FIGURE 6 – Résultats de la méthode MTZ pour l'instance $n = 10$.

3.4 Expérimentation pour $n = 20$

Résultats de la méthode DFJ : Voir Figure 7
Résultats de la méthode MTZ : Voir Figure 12
Pour voir la traversée en $n = 20$: (Vidéo de la traversée)
Cette fois-ci DFJ n'est plus capable de résoudre le problème. En effet, DFJ calcule 2^{20} contraintes ici...

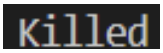

FIGURE 7 – Résultats de la méthode DFJ pour l'instance $n = 20$.

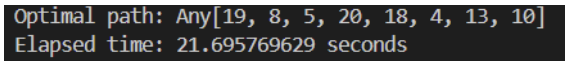

FIGURE 8 – Résultats de la méthode MTZ pour l'instance $n = 20$.

4 Optimisation de la méthode DFJ

Comme nous avons pu le voir précédemment, la méthode DFJ augmente exponentiellement la quantité de contraintes en fonction du nombre d'aérodromes n. Ainsi, il devient très vite impossible de résoudre le problème en utilisant cette méthode. Nous avons donc choisi de mettre en place deux solutions à ce problème :

— **Méthode par augmentation de contrainte** : Nous allons résoudre le problème dans un cas où les contraintes de connexité sont relâchées, puis en fonction de la solution retournée augmenter les contraintes du problème.
— **Méthode par séparation** : Nous allons passer par un problème intermédiaire de séparation afin de réduire la quantité de contraintes sur les x_{ij}.

4.1 La méthode par augmentation de contrainte

L'idée derrière cette méthode est relativement simple : D'abord, nous initialisons le modèle avec toutes les contraintes autre que celle de *connexité*. Ensuite on optimise ce modèle, on le résout. Enfin, on teste la solution pour vérifier qu'aucune contrainte DFJ n'a été violée. Si l'une des contraintes n'est pas respectée, on la rajoute au modèle et on reprend l'étape précédente. Cette approche garantit que seules les contraintes DFJ pertinentes sont ajoutées, rendant l'algorithme plus efficace.
Cela peut se formaliser en le pseudo-code : Algorithm 2

4.1.1 Comparaison DFJ/OptiDFJ pour $n = 16$

Résultats de la méthode DFJ : Voir Figure 9

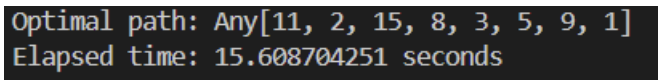

FIGURE 9 – Résultats de la méthode DFJ pour l'instance $n = 16$.

Résultats de la méthode OptiDFJ : Voir Figure 10
On voit très vite que cette méthode est bien plus performante que DFJ.

Algorithm 2: Résolution du problème d'aviation méthode OptiDFJ

Input : Nom du fichier de données `file`
Output : Chemin optimal

Initialisation :
Lire les données de l'instance depuis `file`
$n, d, f, Amin, Nr, R$, regions, coords, $D \leftarrow$ données de l'instance
Création du modèle d'optimisation :
Modèle issu de l'Algorithme 1 **sans les contraintes DFJ/MTZ.**
Ajout d'une contrainte pour éviter le sur-place : $\sum_{i=1}^{n} x[i,i] = 0$
Résolution du modèle :
Résoudre le modèle d'optimisation
while *Subset non vide* **do**
    Récupération des résultats :
    Extraire la solution $x[i,j]$
    Tester la violation des contraintes DFJ par la solution $x[i,j]$
    $Subset \leftarrow$ Premier sous-tour trouvé
    if *Subset non vide* **then**
        Ajouter une contrainte DFJ : $\sum_{i,j \in Subset} x[i,j] \leq |Subset| - 1$
Extraire la solution $x[i,j]$
Suivre le chemin optimal en partant de d jusqu'à f
return *Chemin optimal*

```
Optimal path: Any[11, 2, 15, 8, 3, 5, 9, 1]
Number of constraints added: 2
Elapsed time: 3.10988521 seconds
```

FIGURE 10 – Résultats de la méthode OptiDFJ pour l'instance $n = 16$.

4.1.2 Comparaison DFJ/OptiDFJ pour $n = 18$

Résultats de la méthode DFJ : Le processus a été interrompu car trop gourmand en ressources.

Résultats de la méthode OptiDFJ : Voir Figure 11
La méthode OptiDFJ fonctionne encore pour des instances où DFJ ne fonctionne plus.

```
Optimal path: Any[12, 7, 15, 2, 18, 16, 9, 5, 3]
Number of constraints added: 6
Elapsed time: 3.70402243 seconds
```

FIGURE 11 – Résultats de la méthode OptiDFJ pour l'instance $n = 18$.

4.1.3 Comparaison MTZ/OptiDFJ pour $n = 20$

Nous allons pour cette instance $n = 20$ (L'instance 20_2 pour changer) tester MTZ contre OptiDFJ car en effet DFJ ne converge plus.

Résultats de la méthode MTZ : Voir Figure 12

Résultats de la méthode OptiDFJ : Voir Figure 13
On voit que MTZ reste vraiment la méthode la plus efficace et de loin.

```
Optimal path: Any[1, 15, 12, 17, 18, 13, 14, 10]
Elapsed time: 8.375667469 seconds
```

FIGURE 12 – Résultats de la méthode MTZ pour l'instance $n = 20$.

```
Optimal path: Any[1, 15, 12, 17, 18, 13, 14, 10]
Number of constraints added: 147
Elapsed time: 112.905345346 seconds
```

FIGURE 13 – Résultats de la méthode OptiDFJ pour l'instance $n = 20$.

La méthode OptiDFJ a beaucoup d'avantages, notamment le fait qu'en fonction des conditions du problème, elle peut converger plus rapidement que les autres méthodes (même MTZ). Cependant sa complexité au pire en temps est toujours $O(2^n)$.

4.2 La méthode par séparation

4.2.1 Le Problème de Séparation

Comme nous l'avons vu précédemment, en pratique, il est computationnellement infaisable d'inclure toutes les contraintes possibles en raison du nombre exponentiel de sous-ensembles. La méthode alors présentée auparavant consistait en l'ajout de contraintes de manière itérative si celle-ci était nécessaire. Cependant la méthode d'identification de ces contraintes violées nécessite un parcours complet de la solution et des sous-ensembles DFJ. Nous introduisons alors la méthode par résolution du problème de séparation.

Le problème de séparation est résolu de manière répétée dans le cadre d'un algorithme de résolution. Étant donnée une solution du problème relâché (qui peut potentiellement inclure des sous-tours), le problème de séparation vérifie si un sous-ensemble S d'aérodromes forme un circuit fermé. Si un tel sous-ensemble est trouvé, cela implique une violation des contraintes DFJ, ce qui signifie que des contraintes supplémentaires doivent être ajoutées pour éliminer le sous-tour. Cette approche est fortement similaire à l'approche présentée au-dessus mais en théorie pour des grandes valeurs de n, elle devrait se montrer meilleure.

4.2.2 Formulation Mathématique du Problème de Séparation

Soient x^*_{ij} les valeurs actuelles des variables de décision obtenues à partir de la solution relâchée. Le problème de séparation introduit des variables auxiliaires w_{ij} et z_i pour déterminer s'il existe un sous-ensemble $S \subset \{1,2,\ldots,n\}$ qui forme un sous-tour. Le problème peut être formulé comme suit :

$$\text{Maximiser} \sum_{i=1}^{n} w_{ij} x^*_{ij} - \sum_{i=1}^{m} z_i + 1$$

sous les contraintes :

$$\begin{cases} \sum_{i=1}^{n} z_i \geq 1, & \text{(au moins un nœud dans le sous-ensemble)} \\ w_{ij} \geq z_i + z_j - 1, & \text{(contraintes de liaison)} \\ w_{ij} \leq z_i, & \\ w_{ij} \leq z_j, & \\ z_i \in \{0,1\}, & \text{(contrainte binaire sur la sélection des nœuds)} \\ w_{ij} \geq 0. & \text{(contrainte de non-négativité)} \end{cases}$$

Dans cette formulation :
- z_i est une variable binaire qui indique si le nœud i est inclus dans le sous-ensemble S.
- w_{ij} représente si une arête (i, j) se trouve dans le sous-ensemble.

En maximisant la fonction objectif, nous tentons de détecter le sous-ensemble S dont la contrainte DFJ est la plus violée. Si une violation est détectée, la contrainte correspondante est ajoutée au modèle principal du problème pour empêcher ce sous-tour de se reproduire. Ce processus itératif continue jusqu'à ce qu'aucun sous-tour ne soit détecté (ce qui correspond à une solution optimale ≤ 0), garantissant ainsi une solution optimale du problème sans sous-tours.

4.2.3 Comparaison SepDFJ/OptiDFJ pour $n = 10$

Résultats de la méthode OptiDFJ : Voir Figure 14

```
Optimal path: Any[1, 5, 4, 6]
Number of constraints added: 1
Elapsed time: 7.635248359 seconds
```

FIGURE 14 – Résultats de la méthode OptiDFJ pour l'instance $n = 10$.

Résultats de la méthode SepDFJ : Voir Figure 15
On observe que la méthode de séparation est moins rapide que la méthode d'ajout (avec reconnaissance de sous-tour).

```
Optimal path: Any[1, 5, 4, 6]
Elapsed time: 8.95856567 seconds
```

FIGURE 15 – Résultats de la méthode SepDFJ pour l'instance $n = 10$.

En fait, pour des instances plus élevées telles que $n = 20$: La méthode de séparation prend beaucoup plus de temps que la méthode OptiDFJ. [3]

3. La méthode de séparation ne converge toujours pas après plusieurs minutes alors que OptiDFJ converge en moins d'une minute.

5 Conclusion

5.0.1 Tableau de résultats (test sur les instances n_1)

Nous ne présentons pas dans ces tableaux les résultats issus de SepDFJ et OptiDFJ étant donné que ces deux méthodes ne fonctionnent pas par résolution unique d'un problème par Branch-and-Cut.

n =	6	8	10	16	18	20	40	80
MTZ	7.0	8.0	8.0	12.0	8.11	36.3	67.62	X
DFJ	7.0	8.0	8.0	12.0	X	X	X	X

TABLE 1 – Pour chaque instance et chaque formulation, la valeur de la borne par relaxation continue à la racine.

n =	6	8	10	16	18	20	40	80
MTZ	1	1	1	1	1	99016	6172	X
DFJ	1	1	1	1	X	X	X	X

TABLE 2 – Pour chaque instance et chaque formulation, nombre de noeuds explorés.

n =	6	8	10	16	18	20	40	80
MTZ	15	24	34	51	50	47	113	X
DFJ	5	10	12	76	X	X	X	X

TABLE 3 – Pour chaque instance et chaque formulation, nombre d'itérations.

5.1 Finalement

La première mention du problème du voyageur de commerce (TSP = Traveling Salesman Problem) dans un formalisme mathématique remonte aux années 1930 par Karl Menger. Le problème étudié dans ce rapport est un dérivé du TSP. Cependant ces deux problèmes ont en commun les fameuses contraintes de connexité. Historiquement, la première formulation de ces contraintes est la formulation DFJ en 1954 par George Dantzig, Ray Fulkerson, et Selmer Johnson [4]. En effet, cette formulation est relativement intuitive mais elle a le (grand) défaut d'être exponentielle en fonction de n. Le problème va finalement passer de l'ensemble NP à l'ensemble P en 1960 lorsque Harvey Miller, Warren Tucker, et Richard Zemlin vont publier un article [5] dans lequel ils introduisent des contraintes polynomiales : les contraintes MTZ. Dès lors, de nombreuses méthodes d'optimisation du traitement des contraintes seront introduites (comme certaines vues ici) cependant en terme de complexité au pire en temps il apparaîtra assez vite que MTZ obtient les meilleurs résultats. Ce rapport va dans ce sens : Nous observons que pour certains cas spécifiques où le problème est résolu sans grand nombre de sous-tours à restreindre et pour des n petits, il existe des méthodes dérivées de DFJ qui ont la capacité de dépasser MTZ (en vitesse de computation) **cependant** de manière générale et surtout dans les pires cas, MTZ (et ses dérivés) reste la formulation la plus performante.

4. Dans "Solution of a Large-Scale Traveling-Salesman Problem".
5. Article : "Integer Programming Formulation of Traveling Salesman Problems".

5.2 Ouverture

Nous avons étudié dans ce rapport différentes méthodes mises en place pour modéliser et résoudre le célèbre problème du voyageur du commerce. Cependant, nous nous sommes majoritairement penchés sur des améliorations possibles de la méthode DFJ, cela ne signifie pas que la méthode MTZ ne possède aucune variante. En effet, il en existe de nombreuses, chacune ayant des avantages et des inconvénients. Mais aucune des méthodes se basant principalement sur un algorithme de Branch-and-Cut ne surperforme par rapport à MTZ. Sur notre répertoire GitHub nous avons mis une ébauche de code pour un problème équivalent au MTZ par conservation de flot [6] qui est un exemple de ces variantes.

Nous joignons à la conclusion de ce rapport un lien vers une vidéo de la traversée d'avion pour une instance de $n = 50$ obtenue par la (incroyablement performante) méthode MTZ.

Lien de la vidéo.

6. Le grand avantage de la méthode de flot est au niveau de la relaxation continue. En effet, cette méthode impose plus de contraintes sur les variables introduites (qui ne sont plus des u_i mais des $flow_{ij}$) permettant d'atteindre de meilleurs bornes. Cependant elle est plus gourmande en mémoire (de l'ordre de n^2 alors que MTZ est en n). Voir le fichier $solver\_flow.jl$