SOD 324 - Métaheuristiques
Projet SEQATA : SÉQuencement d'ATterrisage d'Avions

Robin LABBÉ, Ewan SEAN
Kévin ASSOBO

January 19, 2022

1 Introduction

1.1 Contexte Industriel

Ce projet traite du séquencement des avions lors de leur arrivée à l'aéroport, problème connu dans la littérature sous le nom de Aircraft Landing Problem. Lors de l'approche d'un aéoroport, chaque avion i dispose d'une date d'atterrissage privilégiée T_i, jugée idéale selon certains critères. En effet, atterrir plus tôt nécessiterait de dépasser sa vitesse de régime optimale conduisant à un surcoût de carburant. De même, retarder l'atterrissage pourrait se faire en ralentissant l'avion ou encore en lui imposant un détour, voire la réalisation d'un ou plusieurs tours d'attente. Un tel retard, outre les coûts induits par la surconsommation de carburant, peut conduire la compagnie aérienne à dédommager des clients qui auraient loupé une correspondance. Indépendamment de ces pénalités, la date d'atterrissage de chaque avion est bornée entre deux valeurs extrèmes E_i, liée à sa vitesse maximale, d'un côté et L_i, liée à sa réserve de carburant, de l'autre.

Par ailleurs les avions créent derrière eux des turbulences qui obligent à respecter une distance minimale entre deux atterrissages successifs sur une même piste, et dans une moindre mesure sur deux pistes voisines. Cette distance dépend des types d'avions concernés et n'est pas symétrique. En effet un petit avion devra attendre plus longtemps s'il est précédé d'un gros avion que l'inverse.

Compte tenu des caractéristiques de chaque avion, de leur date d'atterrissage souhaitée et des contraintes de sécurité, le problème ALP consiste finalement à :

- affecter chaque avion à une pistes d'atterrissage,
- définir l'ordre d'arrivée des avions sur chaque piste,
- pour un ordre donné d'avions, déterminer l'heure exacte d'atterrissage de chacun d'eux. Ce sous-problème est connu en ordonnancement sous le nom de sous-problème de timing (STP);

de façon à ce que le coût de pénalité global soit minimum. Pour de grosses instances, ces différentes phases sont généralement traitées séparément. En particulier pour une seule piste, il s'agît de définir un ordre d'avions, puis de calculer leurs dates d'atterrissage optimales.

Dans la littérature on considère généralement qu'une avance ou un retard engendre un coût linéaire en fonction de l'écartement à la date préférée d'atterrissage, et le sous-problème de timing resultant est souvent traité par programmation linéaire (PL). Cependant la modélisation avec une fonction de coût plus générale (convexe et linéaire par morceau) a été proposée et implantée de manière plus efficace que la PL.

Dans le cadre de ce projet, nous ne nous intéresserons qu'à une seule piste d'atterrissage, et le coût de pénalité sera linéaire en fonction de l'écart par rapport à la date souhaitée.

1.2 Formalisation du problème

Les données

Les temps sont exprimés en minutes entières. Les pénalités sont des flottants, les coûts seront donc également des flottants.

– n : nombre d'avions;
– $A = [1..n]$: ensemble des avions;
– E_i (entier) : heure d'atterrissage au plus tôt de l'avion i;
– T_i (entier) : heure préférée d'atterrissage de l'avion i (target) ;
– L_i (entier) : heure d'atterrissage au plus tard de l'avion i;
– ep_i (flottant) : pénalité unitaire d'avance (earliness) pour l'avion i;
– tp_i (flottant) : pénalité unitaire de retard (tardiness) pour l'avion i;
– S_{ij} (entier) : durée minimale de séparation entre les avions i et j quand i atterrit avant l'avion j;
– $S_{max} = \max_{i,j \in A} S_{ij}$

Les variables de décisions

– x_i : date d'atterrisage calculée pour l'avion i,
– c_i : coût de pénalité pour l'avion i :
$c_i(x_i) = ep_i (T_i - x_i)^+ + tp_i (x_i - T_i)^+$
avec $(X)^+$représentant la partie positive de X.

Les contraintes

– la date d'atterrissage de chaque avion est bornée :
$E_i \leq x_i \leq L_i \quad \forall i \in A$
– les délais de séparation doivent être respectés :
$x_j \geq x_i + S_{ij} \quad \forall i, j \in A^2$ tels que $i \neq j$ et $x_i < x_j$

2 Sous-problème de timing

On s'intéresse dans un premier temps à la résolution du sous-problème de timing pour un ordre fixé d'avion. La brique de code associé permettra de résoudre le sous problème de timing et donc de connaître la valeur associée à un ordre d'avion.

2.1 Modélisation

On modélise le problème de la manière suivante :

$$\min_{x_{ep_i}, x_{tp_i}} \quad \sum_{i=1}^{n} ep_i \, x_{ep_i} + tp_i \, x_{tp_i} \tag{1}$$

$$\text{s.c. :} \quad E_i \leq x_i \leq L_i \quad \forall i \in A \tag{2}$$

$$x_j \geq x_i + S_{ij} \quad \forall i, j \in A^2 \text{ tels que } i \neq j \text{ et } x_i < x_j \tag{3}$$

$$x_{ep_i} \geq T_i - x_i \quad \forall i \in A \tag{4}$$

$$x_{tp_i} \geq x_i - T_i \quad \forall i \in A \tag{5}$$

$$x_j > x_i \quad \forall (i, j) \in A^2 \text{ tels que } j > i \tag{6}$$

$$x_i \geq 0, \; x_{ep_i} \geq 0, \; x_{tp_i} \geq 0 \quad \forall i \in A \tag{7}$$

On a repris les contraintes explicitées précédemment et on a ajouté de nouvelles variables de décision x_{ep_i} et x_{tp_i}. Les contraintes (7) et (4) ainsi que la minimisation de (1) imposent $x_{ep_i} = (T_i - x_i)^+$. En

effet, si $T_i - x_i \geq 0$ la minimisation de (1) imposera de se placer à la borne (car $ep_i \geq 0$), i.e. dans le cas d'égalité $x_{ep_i} = T_i - x_i$. Sinon, si $T_i - x_i < 0$, la contrainte (7) et la minimisation de (1) impose $x_{ep_i} = 0$.

De même $x_{tp_i} = (x_i - T_i)^+$. On a donc égalité entre l'objectif de (1) et l'objectif défini initialement. Les contraintes conduisent donc à la résolution du problème initial.

Si l'inégalité triangulaire est vérifiée, on peut réduire le nombre de contraintes. En effet, on peut remplacer (3) par :

$$x_j \geq x_i + S_{ij}, \ \forall i, j \text{ tels que } \sum_{i=1}^{j-1} S_{ii+1} < S_{max} \tag{8}$$

On justifie cela en itérant l'inégalité (3) :

$$x_j \geq x_{j-1} + S_{j-1j} \geq x_{j-2} + S_{j-2j-1} + S_{j-1j} \geq ... \geq x_i + \sum_{i=1}^{j-1} S_{ii+1}$$

Donc en particulier si la somme ci-dessus dépasse S_{max}, la contrainte entre i et j est nécessairement vérifiée.

Dans la programmation du solveur, quelques lignes permettait de faire une distinction de cas lors de la résolution (entre les instances vérifiant l'inégalité (8) pour tout i, j et les autres). Mais en pratique on a commenté ces lignes et utilisé un solveur qui supposait l'inégalité vérifiée. La vérification prenait un peu de temps pour les grosses instances. Une amélioration pour ne pas avoir à faire le calcul à chaque résolution aurait été d'ajouter un attribut booléen `inequality` dans la classe `Instance` valant Vrai si (8) est vérifiée par l'instance et Faux sinon. Cela permet de ne faire la vérification qu'une seule fois lors de la création de l'instance. Toutes les instances supérieures à 9 vérifiaient $S_{ik} + S_{kj} \geq S_{max}$. Ce n'était pas le cas pour les instances 1 à 8, cependant on s'est focalisé sur les grosses instances d'où notre simplification abusive mais réfléchie du code qui consistait à ne vérifier que les contraintes entre les avions succesifs.

2.2 Résolution

La résolution du sous-problème de timing est implémentée dans le fichier `lp_timing_solver.jl`. On utilise le package `JuMP` pour modéliser facilement le problème. Le code associé est disponible en annexe.

3 DescentSolver

L'objectif du DescentSolver est de trouver une solution optimale (au moins localement) en s'intéressant aux voisins de la solution courante. A chaque fois qu'on rencontre un voisin qui représente une meilleure solution que la solution courante, on met à jour la solution courante puis réitère la recherche aléatoire. Les inconvénients de cette méthode sont multiples :

- On peut se retrouver bloquer dans un voisinage et donc n'obtenir qu'un optimum local.
- On peut, à l'inverse, choisir un voisinage trop grand ou inadapté auquel cas les tirages au sort successifs ne vont pas conduire à une amélioration de l'objectif.
- Il faut arrêter l'algorithme soit au bout d'un certains temps soit au bout d'un certain nombre d'échecs successifs. On peut donc arrêter l'algorithme alors qu'on aurait pu, en théorie, trouver une meilleure solution en laissant les itérations se poursuivre.
-

3.1 Stratégie et voisinages utilisés

Il faut donc trouver un voisinage qui représente un compromis en terme de taille. Notre stratégie était de partir d'une "bonne solution" et de chercher à l'améliorer en consultant aléatoirement ses voisins. Une bonne solution initiale est par exemple de partir du tri par ordre de target.

Une autre piste que nous avions envisagée était de partir d'une mauvaise solution puis de l'améliorer mais cela était très lent en particulier pour les grandes instances et ne semblait pas apporter grand chose. Par exemple, en partant de la solution triée selon l'inverse des target, on pouvait se retrouver bloqué dans une mauvaise solution. Dans le meilleur des cas on atteignait une solution optimale mais en un temps plus long qu'en partant du tri selon les targets. Il est aussi possible de trier les avions selon les bornes des contraintes, nous avons ajouté un nouveau type de tri selon la moyenne des deux bornes (i.e. selon $\frac{L_i+E_i}{2}$). Après des multiples essais, le meilleur choix nous a semblé être de partir d'une solution triée selon les T_i.

On a testé plusieurs voisinages :

- Les shifts, qui consistent à translater un avions à une autre place et à décaler ses nouveaux prédecesseurs/successeurs (respectivement pour un shift "à droite"/"à gauche"). Pour ce type de voisinage il était nécessaire que la distance du shift ne soient pas trop grande (sinon on détériore presque systématiquement la solution). L'avion à translater est tiré au sort de manière uniforme puis sa position cible est tirée au sort de manière uniforme en dessous d'une distance maximale par rapport à la position initiale.

- les swaps, qui consistent à inverser les positions de deux avions à une distance donnée. Un avion est tiré au sort de manière uniforme puis un de ses voisins) une distance inférieure à la distance maximale (2) est tiré au sort. On a également fait une variante ou la deuxième position est tirée au sort selon un loi binomiale centrée en la première. Cela permettait d'augmenter la distance maximale sans risquer d'explorer les voisins les plus lointains trop souvent.

- Le swap proportionel, qui consiste à effectuer le tirage au sort de la première position d'un swap en fonction du coût des avions (la probabilité de tirage correspond au rapport entre le coût de l'avion et le coût total). La deuxième position est tirée au sort comme lors d'un swap classique. Cette méthode permet d'améliorer rapidement une mauvaise solution mais est obsolète si on part d'une bonne solution. De plus, on est très vite bloqué dans une solution.

- Un mélange par bloc, qui consiste à tirer au sort un bloc d'avions d'une taille donnée puis à lui appliquer une permutation tirée au sort de manière uniforme dans l'ensemble des permutations possibles du bloc. L'idée était de trouver un voisinage qui permette de se sortir de certaines impasses en mélangeant tout un bloc. Néanmoins si le bloc est trop grand, le nombre de voisins potentiels l'est également, s'il est trop petit cette méthode équivaut à des swaps et des shifts.

- Enfin il est possible de combiner les opérations suscitées. Premièrement en tirant au sort l'opération à effectuer à chaque itération (cela équivaut à travailler sur l'union de deux voisinages). Deuxièmement en effectuant plusieurs opérations successives au sein d'une même itération (cela équivaut à agrandir le voisinage).

La meilleure méthode que nous avons trouvé combine les shifts et les swaps. En fonction d'un tirage au sort on effectue soit un shift, soit un swap, soit un shift et un swap, soit deux shifts, soit deux swaps.
Il est possible de choisir la méthode utilisée par le solveur grâce au symbole `:nbh`, par exemple avec la commande :

```
./bin/run.jl descent data/09.alp -n 1000 -nbh swap_and_shift -presort target
-dur 300
```

Les différents modes posibles sont : `binomial, shift, proportional, swap, bloc_shuffle, mbsb, swap_and_shift`.

La plupart des commandes correspondent aux opérations qui ont été décrites plus haut avec des valeurs prédéfinies pour les distances de swap, de shift etc. `mbsb` correspond à un mix entre un swap binomial (qui effectue des swap à une distance 2) et un mélange de bloc de taille 5. Un tirage au sort permet de savoir quelle méthode on utilise. On n'a pas vraiment utilisé cette méthode par la suite mais c'était un moyen d'illustrer la possibilité de faire des mix de voisinages.

Listing 1: swap_and_shift!

```
function swap_and_shift!(sol::Solution)
    p = rand() # tirage au sort

    if p<0.2
        random_swaper!(sol, 3, 1) # Un swap à une distance maximal de 3
    elseif p<0.4
        random_shifter!(sol, 3, 1)  # Un shift à une distance maximal de 3
    elseif p<0.6
        random_shifter!(sol, 2, 1, false) # false : on ne résoud pas inutilement le prob
        random_swaper!(sol, 2, 1) # Un shift puis un swap à une distance maximal de 2 ch
    elseif p<0.8
        random_swaper!(sol, 2, 2) # 2 swaps à une distance maximal de 2
    else
        random_shifter!(sol, 2, 2) # 2 shifts à une distance maximal de 2
    end
end
```

3.2 Performances

Pour les instances 1 à 8, on obtient assez rapidement les meilleurs résultats en utilisant notre meilleur voisinage. L'utilisation de DescentSolver nous a permis de voir quelles étaient les barrières rencontrées pour certaines instances et nous a poussé à adapter le voisinage à ces barrières. Par exemple pour l'instance 9, il fallait considérer des voisinages effectuant une composition des plusieurs permutations sans quoi on n'arrivait jamais à la meilleure solution en partant du presort par target. Le choix des paramètres (distance des shift/swap et probabilités) dans la fonction swap_and_shift avait pour but de permettre l'exploration de permutation assez complexe mais en essayant de privilégier les permutations les moins complexes via les probabilités et les distances choisies. On a obtenu des résultats relativement

	Temps	Meilleur objectif	Nombre d'itérations/s	Voisinage utilisé
09.alp	111.684 s	5611.7	171.77	swap_and_shift
10.alp	198.388	13409.99	119.155	swap_and_shift
11.alp	272.524	12418.32	97.779	swap_and_shift
12.alp	325.269	16265.84	70.634	swap_and_shift
13.alp	1200.436	37753.12	37.334	swap_and_shift

Table 1: Meilleurs résultats obtenus avec `DescentSolver`

proches des meilleurs solutions, le descent solver est cependant relativement instable en terme de performances. On ne trouve pas les mêmes résultats à chaque essai car l'amélioration suit un chemin différent à chaque fois qu'on relance le solveur. Une des choses importantes que l'on a apprise grâce à ce premier solveur est qu'il était parfois nécessaire d'effectuer des compositions de permutations pour pouvoir continuer d'améliorer la solution. Cela permet de déplacer globalement les contraintes (échanger les deux premiers avions peut avoir un impact sur les deux derniers).

4 SteepestSolver

Dans les 2 parties suivantes, il s'agit de définir précisément des voisinages et de les explorer intégralement, contrairement au `DescentSolver` où l'exploration du voisinage est aléatoire.

La *Steepest Descent* consiste à définir précisément un voisinage donné, l'explorer complètement et faire de la meilleure solution améliorante trouvée, si elle existe, la nouvelle solution courante. Si après parcours complet du voisinage on ne trouve pas de voisins améliorants, on a atteint un minimum local et l'algorithme s'arrête.

4.1 Classe Mutation et Voisinage

Afin de définir et d'explorer des voisinages plus facilement, nous avons défini une classe `Mutation` et une classe `Voisinage` qu'on a implémenté dans le fichier `/src/mutations.jl`.

Une `Mutation` représente une permutation en utilisant deux attributs `idx_1` et `idx_2` de vecteurs d'entiers. Par exemple, `idx_1 = [1,3]` et `idx_2 = [3,1]` signifie qu'on doit placer l'avion initialement à l'indice 1 à l'indice 3, et que l'avion à l'indice 3 se retrouve en 1. La classe contient une méthode `clean!` qui permet de rendre unique la `Mutation` associée à une permutation donnée (tri des indices et suppression des points fixes) et un opérateur `==` pour tester l'égalité entre deux mutations. On a également ajouté un attribut booléen `clean` pour connaître le statue d'une permutation (si elle est au format unique ou non).

La classe `Voisinage` contient un nom, un vecteur de `Mutation` et également un attribut booléen `clean` pour savoir si le voisinage a été traité par la méthode `clean!`, associée cette fois-ci à `Voisinage`, qui permet de supprimer tous les doublons d'un voisinage après les avoir mis au format unique. Les autres méthodes permettent de générer des voisinages donnés.

4.2 Voisinages définis

`n` est le nombre d'avions dans l'instance.

- `swap(n, shift)` : on inverse les positions de deux avions à une distance donnée `shift` (exactement). Il y a $\texttt{n} - \texttt{shift}$ voisins dans ce voisinage
- `shift(n, shift)` : on translate un avion à une distance donnée `shift` en décalant ses nouveaux prédécesseurs/successeurs. Il y a $(\texttt{n} - \texttt{shift}) * 2$ voisins, où le facteur 2 apparait puisqu'on peut shift à "gauche" ou à "droite"
- `compose_vois(vois1,vois2)` : on compose les permutations issus de deux voisinages. Si n_1 et n_2 sont les tailles respectives de `vois` et `vois2`, il y a environ $n_1 * n_2$ voisins dans ce voisinage (produit cartésien moins la suppression des doublons).

Dans la suite, on appelle "s_d" le voisinage obtenu en effectuant des swaps avec `shift=d`, "t_d" celui obtenu en effectuant des shifts, et "$vois_1\_vois_2$" une composition de deux voisinages.

4.3 Résultats

	Temps	Meilleur objectif	Nombre d'itérations/s	Voisinage utilisé
09.alp	16.289	5611.7	135.429	s_1, s_2, t_2, s_3
10.alp	121.581	12531.5	128.318	s_1, s_2, t_2, s_3
11.alp	152.472	12458.11	76.04	s_1, s_2, t_2, s_3
12.alp	212.734	16798.95	77.341	s_1, s_2, t_2, s_3
13.alp	607.896	40672.57	35.876	s_1, s_2, t_2, s_3

Table 2: Meilleurs résultats obtenus avec `SteepestSolver`

5 VNS

Pour palier au fait que le *Steepest Descent Solver* est sensible au voisinage seléctionné (il converge vers un minimum local de ce voisinage), on étudie dans cette partie l'approche de résolution par *recherche à voisinage variable*. Le principe de cette approche est de définir plusieurs voisinages et de les explorer en entier successivement jusqu'à trouver un voisin améliorant dans un des voisinages. La condition d'arrêt de l'algorithme est soit une contrainte temporelle (car une heuristique se doit de fournir une solution réalisable rapidement), soit quand la solution courante est un minimum local pour tous les voisinages choisis.

La qualité du résultat obtenu et la vitesse de résolution dépendent du choix des différents voisinages **ET** de leur ordre. Concernant l'ordre, il faut aller du voisinage le plus simple au plus complexe afin d'accélérer la vitesse de résolution (on n'explore les voisinages complexes seulement si on est bloqué dans un minimum local des voisinages simples). Concernant le choix des voisinages, il est nécéssaire d'avoir des voisinages simples pour garantir une convergence rapide en début d'exécution, et des voisinages plus complexes pour sortir d'éventuels minima locaux, tout en restant tout de même assez simples pour ne pas explorer un trop grand nombre de voisins.

Tout comme la *Steepest Descent*, on a le choix d'adopter une nouvelle solution courante dès que l'on trouve un voisin améliorant, ou explorer l'intégralité du voisinage et faire du meilleur voisin trouvé (s'il est améliorant) la nouvelle solution courante. Pour la VNS, on préfère en général explorer totalement le voisinage courant.

5.1 Voisinages successifs

Pour obtenir les résultats ci-dessous, nous avons décidé de parcourir successivement ces voisinages :

- s_1 : $n-1$ voisins
- s_2 : $n-2$ voisins
- t_2 : $2(n-2)$ voisins
- s_3 : $n-3$ voisins
- t_3 : $2(n-3)$ voisins
- $s_1\_s_1$: environ $(n-1)^2$ voisins, à éviter pour les très grandes instances.

L'idée est d'explorer d'abord des voisinages contenant des voisins proches de la solution courante (pas de grand changement dans l'ordre des avions, en effectuant un swap de deux avions à une distance de 1 ou 2 dans l'ordre d'arrivée de la solution courante), puis d'explorer des voisinages plus grands avec des voisins plus éloignées si on ne trouve pas de solution améliorantes (d'où le parcours des voisinages swap avant les voisinages shift).

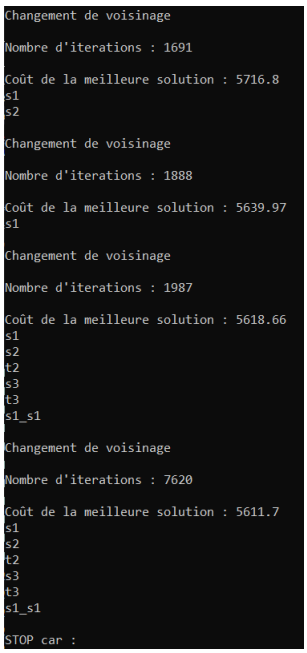
5.2 Résultats

Le *VnsSolver* permet de trouver très rapidement l'optimum des 7 premières instances, en partant de la solution fournie par l'heuristique gloutonne qui fixe l'ordre des avions dans l'ordre de leur target. T_i. Meilleurs résultats obtenus avec `DescentSolver`

	Temps	Meilleur objectif	Nombre d'itérations/s	Voisinage utilisé
09.alp	54s	5611.7	242.5	$s_1, s_2, t_2, s_3, t_3, s_1\_s_1$
10.alp	156s	12557.7	147.5	$s_1, s_2, t_2, s_3, t_3, s_1\_s_1$
11.alp	258s	12450.01	117.4	$s_1, s_2, t_2, s_3, t_3, s_1\_s_1$
12.alp	645s	16275.11	82.8	$s_1, s_2, t_2, s_3, t_3, s_1\_s_1$
13.alp	607s	39315.43	46.7	s_1, s_2, t_2, s_3, t_3

Table 3: Meilleurs résultats obtenus avec `VnsSolver`

On obtient des résultats proches des meilleures solutions connues (ou égal dans le cas de l'instance **09.alp**). Contrairement au *Descent Solver*, cette approche est déterministe et convergera toujours vers la même solution à condition initiale fixée (à part si l'on décide de se déplacer vers le premier voisin améliorant ET que l'on parcoure les voisins des voisinages définis dans un ordre aléatoire). De plus, l'algorithme s'arrête quand on atteint le minimum local de multiples voisinages, ce qui permet de palier au défaut du simple *Steepest Descent Solver*. En pratique, dans l'exécution de l'algorithme, la plupart des changements de solution courante s'effectuent en explorant seulement le premier voisinage (surtout au début). Le solveur se comporte donc comme celui de la Steepest Descent dans ces cas-là, mais occasionnellement l'algorithme se débloque en explorant les voisinages plus complexes/éloignées.



On peut voir dans cette fin de recherche de l'algorithme appliqué à **09.alp** comment la VNS passe d'une solution à une autre. En particulier, pour passer de la solution avec un coût de 5618 à celle de 5611, le voisin améliorant a été trouvé dans le dernier voisinage (composition de deux swap). Puis, cette dernière solution est un minimum local de chacun des 6 voisinages donc l'algorithme s'arrête.

6 Synthèse

6.1 Résultats

Le *Descent Solver* a pour avantage de pouvoir tester des voisins plus arbitrairement complexes (car on construit un voisin aléatoirement et on n'a pas besoin de parcourir le voisinage en entier), et il peut trouver des aussi bonnes voire meilleures solutions que le *VNS Solver*. Mais ses performances ne sont pas garanties et sont relativement instables. Il est donc préférable de lancer le solveur plusieurs fois et de garder la meilleure solution trouvée.

Le *Steepest Descent Solver* consiste à définir précisément un voisinage et de le parcourir en entier pour trouver de meilleures solutions. Deux stratégies sont possibles, changer de voisinage dès qu'on trouve un voisin améliorant ou parcourir tout le voisinage et garder la meilleure solution.

Dans le deuxième cas, si on prend on voisinage trop restreint on risque d'être très rapidement bloqué. A l'inverse, si on prend un voisinage trop vaste, on risque d'être lent dans la progression vers l'optimal car on parcourt l'intégralité du voisinage à chaque fois.

Pour ces deux raisons, il nous a semblés plus pertinent d'opter pour le premier cas. Changer de voisinage dès que l'on trouve un voisin améliorant permet de gagner du temps lorsqu'on travaille avec un gros voisinage. Le fait de parcourir tout le voisinage tant que l'on n'a pas trouvé un voisin améliorant minimise nos chance d'être bloqué lorsqu'on travaille avec un gros voisinage. Il nous a semblés pertinent de mélanger le voisinage à chaque fois qu'on change de voisin car cela évite de ré-étudier des voisins très proches à chaque fois qu'on effectue la boucle sur l'exploration des voisins. Cependant cela ajoute un caractère aléatoire à la résolution.

La *VNS* consiste à changer de voisinage quand on est bloqués dans un voisinage donné. L'algorithme s'arrête quand on se retrouve dans un minimum local de tous les voisinages choisis : cela permet de palier au défaut du simple Steepest Descent Solver qui s'arrête souvent prématurément en fonction du voisinage choisi. Son principal avantage comparé au *Descent Solver* est que ses performances sont stables, grâce à son caractère déterministe, et généralement bonnes grâce au parcours exhaustif des multiples voisinages. En revanche, son execution est donc plus lente.

6.2 Critiques

L'archive était très facile à comprendre et à manipuler. Nous avons pu nous approprier rapidement les concepts vus en cours grâce à ce projet. Nous aurions peut-être pu améliorer notre code en utilisant mieux certaines fonctionnalités de l'archive (notamment la verbosité). Nous n'avons pas réussi à utiliser la machine virtuelle de l'ENSTA pour faire nos calculs à distance (la commande `julia` n'était pas reconnu lorsque nous avons essayé), ce qui nous a fait perdre beaucoup de temps et nous a empêché de tester nos algorithmes sur de plus grands voisinages. Pour certains solveurs on aurait pu implémenter plus de choix via les des arguments passés dans le terminal (par exemple choisir quelles permutations sont dans les différents voisinages de vns et steepest en passant `--nbh s1+s2+t2` en commande).

Il aurait été intéressant de tester d'autres métaheuristiques sur ce problème. Le recuit simulé aurait pu être pertinent car lorsqu'on bloquait dans un voisinage le déteriorer un peu aurait peut-être permis d'améliorer ensuite la solution trouvée. A l'inverse il semblait difficile de travailler avec des familles de solutions étant donné la taille de certaines instances.

6.3 Commandes à utiliser

Pour faire appel au descent solver avec le voisinage mixant des shift et des swap aléatoire :

```
./bin/run.jl descent data/09.alp -n 10000 --presort target --dur 300 --nbh
swap_and_shift
```

Pour faire appel au steepest solver avec l'option qui permet de changer de voisin dès qu'on en a trouvé un améliorant :

```
julia ./bin/run.jl steepest data/05.alp -n 10000 --presort target --dur 200
--fb
```

Pour faire appel au VNS solver (l'option –fb et le choix du voisinage n'a pas été implémenté à la date de rendu de ce rapport) :

```
julia ./bin/run.jl vns data/05.alp --presort target --dur 300
```

7 Conclusion

Ce projet nous a permis de mettre en place des métaheuristiques de manière très appliquée. Le format et la structure de l'archive nous a permis de nous concentrer sur la stratégie plus que sur la programmation. Nous avons réussi à résoudre certaines des instances mais cela a nécessité d'adapter notre stratégie au fur et à mesure des résultats.

8 Annexes

Listing 2: lp_timing_solver.jl

```
export LpTimingSolver, symbol, solve!
using JuMP
# Déclaration des packages utilisés dans ce fichier
# certains sont déjà chargés dans le fichier usings.jl

"""
    LpTimingSolver

Résoud du Sous-Problème de Timing par programmation linéaire.

Ce solveur résoud le  sous-problème de timing consistant à trouver les dates
optimales d'atterrissage des avions à ordre fixé.
Par rapport aux autres solvers (e.g DescentSolver, AnnealingSolver, ...), il
ne contient pas d'attribut bestsol
```

```
VERSION -t lp4 renommé en Lp : modèle diam version sans réoptimisation (coldrestart)
  - modèle diam simplifié par rapport à lp3 : sans réoptimisation (coldrestart)
  - pas de réoptimisation : on recrée le modèle à chaque nouvelle permu
    d'avion dans le solver
  - seules les contraintes de séparation nécessaires à la permu sont créées
  - gestion de l'option --assume_trineq true|false (true par défaut, cf lp1)
  - contraintes de coût simple (diam) : une seule variable de coût par avion
    plus un contrainte par segment :
       cost[i] >= tout_segment[i]
"""
mutable struct LpTimingSolver
    inst::Instance
    # Les attributs spécifiques au modèle
    model::Model  # Le modèle MIP
    x         # vecteur des variables d'atterrissage
    cost      # variable du coût de la solution
    costs     # variables du coût de chaque avion

    nb_calls::Int    # POUR FAIRE VOS MESURES DE PERFORMANCE !
    nb_infeasable::Int

    # Le constructeur
    function LpTimingSolver(inst::Instance)
        this = new()

        this.inst = inst

        # Création et configuration du modèle selon le solveur externe sélectionné
        this.model = new_lp_model() # SERA  R E G  N  R   DANS CHAQUE solve!()

        this.nb_calls = 0
        this.nb_infeasable = 0

        return this
    end
end

# Permettre de retrouver le nom de notre XxxxTimingSolver à partir de l'objet
function symbol(sv::LpTimingSolver)
    return :lp
end

function solve!(sv::LpTimingSolver, sol::Solution)

    #error("\n\nMéthode solve!(sv::LpTimingSolver, ...) non implanté : AU BOULOT :-)\n\n

    sv.nb_calls += 1

    #
    # 1. Création du modèle spécifiquement pour cet ordre d'avion de cette solution
    #


    sv.model = new_lp_model()
```

```
#    COMPL TER : variables ? contraintes ? objectif ?

n = sv.inst.nb_planes
planes = sol.planes

# Variables
@variable(sv.model, x[1:n] >= 0) # temps d'arrivée
@variable(sv.model, x_ep[1:n] >= 0) # variable pour un avion en avance sur sa target
@variable(sv.model, x_tp[1:n] >= 0) # variable pour un avion en retard sur sa target

# Contraintes

# Respect des contraintes d'heure d'arrivée
@constraint(sv.model, [i in 1:n], x[i] >= planes[i].lb)
@constraint(sv.model, [i in 1:n], x[i] <= planes[i].ub)

# Contraintes sur les valeurs x_ep et x_tp
# La minimisation de l'objectif entrainera x_ep > 0 ==> x_tp ==0
# et planes[i].target - x[i] >= 0 ==> x_ep[i] == planes[i].target - x[i]
@constraint(sv.model, [i in 1:n], x_ep[i] >= planes[i].target - x[i])

# Et réciproquement
@constraint(sv.model, [i in 1:n], x_tp[i] >= -(planes[i].target - x[i]))

# Il faut également respecter les délais entre deux arrivées successives

# On a commenté la partie du code qui permettait d'élaguer certaines contraintes
# en supposant Sik + Skj > Smax, pour tout k,i,j
# c'est le cas uniquement pour les grosses instances (celles qui nous intéressent le

# S = [get_sep(sol.inst, planes[i],planes[j]) for i in 1:n, j in 1:n]
# Smax = maximum(S)
# SS = [S[i,k]+S[k,j] for i in 1:n, j in 1:n, k in 1:n]
# if minimum(SS) >= Smax

@constraint(sv.model, [i in 1:n-1], x[i+1] >= x[i] + get_sep(sol.inst, planes[i],pla
# else
    # @constraint(sv.model, [i in 1:n,j in 1:n; i<j], x[j] >= x[i] + get_sep(sol.ins
# end
# Objectif
@objective(sv.model, Min ,sum(planes[i].ep * x_ep[i] + planes[i].tp * x_tp[i] for i


# 2. résolution du problème à permu d'avion fixée
#
JuMP.optimize!(sv.model)

# 3. Test de la validité du résultat et mise à jour de la solution
if JuMP.termination_status(sv.model) == MOI.OPTIMAL
    # tout va bien, on peut exploiter le résultat

    # 4. Extraction des valeurs des variables d'atterrissage
    sv.x = value.(x)
    sv.cost = objective_value(sv.model)
    sv.costs = [0.0 for i in 1:n]
    for i in 1:n
```

```
            sv.costs[i] = planes[i].ep*value.(x_ep[i]) + planes[i].tp*value.(x_tp[i])
        end

        # ATTENTION : les tableaux x et costs sont dans l'ordre de
        # l'instance et non pas de la solution !
        for (i, p) in enumerate(sol.planes)
            sol.x[i] = round(Int, value(sv.x[i]))
            # Cet arrondi permet d'utiliser le solver linéaire Tulip qui utilise
            # une méthode de points intérieurs et qui contrairement au simplexe,
            # ne donne pas nécessairement une solution située sur un point extrème
            # du polytope des contraintes.
            # Cela est possible car pour toutes les instances, les pénalités
            # unitaire sont exprimées en centièmes.
            # sol.costs[i] = value(sv.costs[p.id])
            sol.costs[i] = round(value(sv.costs[p.id]), digits=2)
        end
        prec = Args.get(:cost_precision)
        sol.cost = round(value(sv.cost), digits = prec)

    else
        # La solution du solver est invalide : on utilise le placement au plus
        # tôt de façon à disposer malgré tout d'un coût pénalisé afin de pouvoir
        # continuer la recherche heuristique de solutions.
        sv.nb_infeasable += 1
        solve_to_earliest!(sol)
    end

    # println("END solve!(LpTimingSolver, sol)")
end
```