ENSTA - année 2022-2023
Cours d'Optimisation Combinatoire SOD324 (RO-MH)

Projet SEQATA : SÉQuencement d'ATterrisage d'Avions [1]

02/12/2022 (version 1.3.0)

L'objectif de ce projet est de modéliser puis de résoudre un problème de séquencement des avions lors de leur arrivée à l'aéroport en mettant en œuvre concrètement les heuristiques/métaheuristiques présentées dans le cours. L'idée est d'implanter certains des algorithmes en les adaptant au mieux au problème : adaptation de la représentation informatique de la solution, des paramètres propres aux algorithmes mis en jeu, etc...

1 Description du problème

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 [2]. 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 [4, 3].

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 (figure 1).

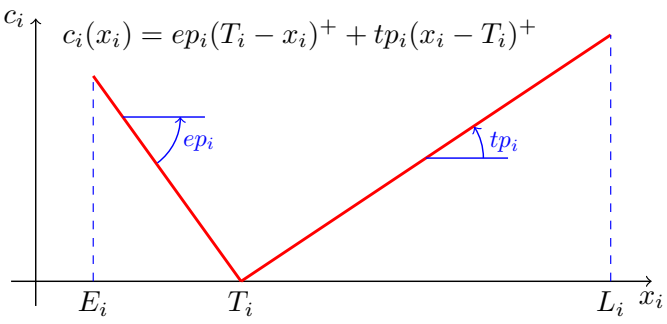

FIGURE 1 – fonction de coût de l'avion i

1. Agnès Plateau (CNAM/CEDRIC/OC), Maurice Diamantini (ENSTA Paris/UMA) et Natalia jorquera (ENSTA Paris/UMA).

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 ;

Par ailleurs, nous supposerons l'inégalité triangulaire vérifiée au niveau des temps de séparation, c'est-à-dire que : $S_{ik} + S_{kj} >= S_{ij} \ \forall i, j, k \in [1, n]^3 S_{ik} + S_{kj} >= S_{ij} \ \forall i, j, k \in \{1, ..., n\}^3$. Autrement dit, on ne peut pas diminuer le temps minimum de séparation entre deux avions i et j en en insérant un troisième k entre les deux !

Chaque instance sera intégralement décrite dans un unique fichier texte dont le format est décrit en Section 2.

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 $x_i < x_j$

L'objectif

Le problème consiste à définir une date x_i d'atterrissage pour chaque avion i en minimisant la somme des coûts de pénalité individuels c_i tout en respectant l'ensemble des contraintes.

2 Exemple d'instance, format des fichiers et outils

2.1 Les données d'entrée

Nous utiliserons deux jeux d'instances de référence pour le problème ALP [1]. Le premier jeu (airland 01 à 08) représente des instances de petites tailles, mais parfois pathologiques. Elles peuvent être traitées par une méthode exacte et les solutions sont connues (i.e. prouvées). Le second jeu (airland 09 à 13) propose des instances plus grosses et plus réalistes.

Ces instances sont également utilisées dans la littérature pour traiter une variante dynamique du problème ALP dans laquelle les avions apparaissent au fur et à mesure (paramètre `AppearingTime` de chaque avion). Cette variante peut entraîner une remise en cause de l'ordonnancement partiel déjà réalisé pour les avions dont la date d'atterrissage n'est pas trop proche dans le futur (paramètre global `freeze_time`). Pour notre problème, nous pourrons ignorer ces deux paramètres des fichiers d'instance. De plus par rapport au problème décrit, les temps de séparation sont indiqués par une matrice dont les indices représentent les types d'avion (relativement peu nombreux).

Les différentes instances du problème sont fournies sous la forme de fichiers textes au format suivant [2] :

```
# Aircraft Landing Problem instance (ALP) version 1.0

name alp_01_p10
nb_planes 10
nb_kinds 2
freeze_time 10

#    name  kind   at     E     T     L    ep    tp
plane  p1    1    55   130   156   560    10.0 10.0
plane  p2    1   121   196   259   745    10.0 10.0
plane  p3    2    15    90    99   511    30.0 30.0
plane  p4    2    22    97   107   522    30.0 30.0
plane  p5    2    36   111   124   556    30.0 30.0
plane  p6    2    46   121   136   577    30.0 30.0
plane  p7    2    50   125   139   578    30.0 30.0
plane  p8    2    52   127   141   574    30.0 30.0
plane  p9    2    61   136   151   592    30.0 30.0
plane p10    2    86   161   181   658    30.0 30.0

# Separation time : kind1 kind2 value
sep 1 1 3
sep 1 2 15
sep 2 1 15
sep 2 2 8
```

Tous les fichiers d'instances respectent les règles de syntaxe suivantes :

— Toute ligne commençant par `#` est un commentaire.
— Chaque ligne significative commence par un mot clé (la commande) suivi de ses paramètres.
— La commande `name` indique le nom de l'instance. Celui-ci pourra être utilisé par exemple dans le nom du fichier de sortie.
— La commande `nb_planes` indique le nombre d'avions. En principe il correspond au nombre de lignes `plane`, mais il peut être modifié pour réduire la taille de l'instance (e.g passé en paramètre de votre programme).
— La commande `nb_kinds` définit le nombre de types d'avions différents et sera utilisée pour dimensionner la matrice de séparation.
— La commande `freeze_time` (inutilisé pour ce projet) indique la durée minimale en-dessous de laquelle la date d'atterrissage d'un avion déjà séquencé ne peut plus être remise en cause.
— La commande `plane` déclare les paramètres d'un nouvel avion :
  — `name` (String) : nom arbitraire. Dans nos instances elle seront de la forme `p1`, `p2`, ... ,
  — `kind` : type de l'avion,
  — `at` (Appearing Time) : inutile ici, réservé au problème dynamique,
  — `T` (entier) : heure d'atterrissage préférée,
  — `E` et `L` (entier) : heures d'atterrissage au plus tôt et au plus tard,
  — `ep` et `tp` : pénalités unitaires en cas d'avance ou de retard par rapport à la l'heure souhaitée.
— La commande `sep` indique la durée de séparation S_{ij} entre un avion i de type `kind1` suivi d'un avion j de type `kind2`.

2.2 Format d'une solution

Voici la solution correspondant à l'instance précédente. Tout ce qui suit un caractère `#` est un commentaire.

— la commande `name` rappelle le nom de l'instance traitée,
— la commande `timestamp` indique la date/heure à laquelle ce fichier a été généré, ce qui vous permet de vous y retrouver parmi vos différents essais,
— la commande `cost` résume le coût prétendu de cette solution,
— la commande `order` liste les avions dans l'ordre d'atterrissage,
— la commande `landing` fournit les paramètres d'atterrissage d'un avion. L'ordre des lignes `landing` n'est pas significatif : vous pouvez conserver l'ordre de l'instance : p1, p2, ...) mais cela vous simplifiera la vie de les afficher dans l'ordre de l'atterrissage.

2. Les instances d'origine ont été transcodées dans un nouveau format plus lisible et mieux adapté à une généralisation de la fonction de coût ([3]). Accessoirement la taille de l'instance de 500 avions passe de 830 Koctets dans l'ancien format à 27 Koctets dans ce nouveau format.

Pour chaque atterrissage seront indiqués le nom de l'avion, sa date réelle d'atterrissage (le x_i), l'écart par rapport à sa date souhaitée et son coût de pénalité. En commentaire *peuvent* être rappelés les caractéristiques de l'avion (pour vérification) mais surtout le temps de séparation réél (ainsi que le minimum requis entre parenthèses) par rapport à ses plus proches prédécesseurs. Ceci vous sera utile pour la mise au point et la validation de vos solutions.

```
name alp_01_p10
timestamp 2022-11-08T16:57:22.343
cost 700.0
order [p3,p4,p5,p6,p7,p8,p9,p1,p10,p2]

#      name  t  dt  cost  # comments
landing  p3  99  0   0.0  # E=90  T=99  L=511  ep=30.0 tp=30.0 sep -ok-
landing  p4 107  0   0.0  # E=97  T=107 L=522  ep=30.0 tp=30.0 sep 8(8) -ok-
landing  p5 119 -5 150.0  # E=111 T=124 L=556  ep=30.0 tp=30.0 sep 12(8) 20(8) -ok-
landing  p6 127 -9 270.0  # E=121 T=136 L=577  ep=30.0 tp=30.0 sep 8(8) 20(8) 28(8) -ok-
landing  p7 135 -4 120.0  # E=125 T=139 L=578  ep=30.0 tp=30.0 sep 8(8) 16(8) 28(8) -ok-
landing  p8 143  2  60.0  # E=127 T=141 L=574  ep=30.0 tp=30.0 sep 8(8) 16(8) 24(8) -ok-
landing  p9 151  0   0.0  # E=136 T=151 L=592  ep=30.0 tp=30.0 sep 8(8) 16(8) 24(8) -ok-
landing  p1 166 10 100.0  # E=130 T=156 L=560  ep=10.0 tp=10.0 sep 15(15) 23(15) 31(15) -ok-
landing p10 181  0   0.0  # E=161 T=181 L=658  ep=30.0 tp=30.0 sep 15(15) 30(8) 38(8) -ok-
landing  p2 259  0   0.0  # E=196 T=259 L=745  ep=10.0 tp=10.0 sep 78(15) 93(3) 108(15) -ok-
```

2.3 Prototype pour le projet

Bien que vous soyez en majorité formés en C++, vous pourrez développer votre code dans un langage efficace de votre choix (typiquement C++, Julia, Java, C, Rust, Fortran). Mais les formats d'entrée et de sortie décrits ci-dessus ne sont pas une option et devront impérativement être respectés.

Cependant afin de vous faire gagner du temps en vous concentrant sur la partie optimisation du problème, il vous sera fourni un prototype de code `proto_seqata` écrit en Julia. Cette archive est un projet opérationnel dans le sens où il fonctionne correctement à l'ensta, s'exécute et gère les options de la ligne de commande en proposant une aide sur son utilisation. En l'état, il vous permet de lire un fichier d'instance passé en paramètre, de le résoudre à l'aide d'une méthode stupide et d'en enregistrer une solution valide dans un fichier au format standard. Il est organisé pour pouvoir utiliser plusieurs méthodes (recuit, descente, ...) en s'appuyant sur plusieurs algorithmes de résolution du *sous-problème de timing* (seul `EarliestTimingSolver` est opérationnel). De plus il vous permet de choisir le solveur PL externe (CPLEX, CLP ou GLPK sont disponibles à l'ENSTA).

Même si le but du cours est la recherche opérationnelle et non pas la programmation, vous avez fortement intérêt à vous inspirer de l'organisation de ce code (e.g si vous voulez coder en C++) ou à vous l'approprier (en Julia) pour être capable de l'adapter et le compléter pour votre propre besoin. En effet, il suit un modèle d'organisation relativement générique, extensible et souple pour résoudre différents types de problèmes d'optimisation.

L'utilisation de ce prototype et son organisation sont décrites dans les fichiers de sous-répertoire `docs/src/` de l'archive du projet.

2.4 Validation des solutions

Ce prototype intègre également un **validateur** qui vous permettra de vérifier la conformité de vos solutions. Outre sa fonctionnalité de validation, vous pourrez aussi l'utiliser comme exemple de manipulation d'un objet `Solution` depuis votre code Julia. Les lignes suivantes permettent de résoudre (en supposant que votre action `descent` soit opérationnelle !) puis de valider la solution générée :

```
./bin/run.jl descent data/09.alp  -n 1000
=> action descent : création d'un fichier solution dans ./_tmp/
./bin/run.jl validate data/09.alp _tmp/alp_09_p100=5676.96.sol
=> affiche : "Solution correcte de coût : 5676.96"
```

Précisons qu'***une solution non acceptée par le validateur ne sera pas prise en compte pour vos résultats***.

3 Travail demandé

Le travail demandé se décompose en plusieurs questions, les premières servant de base aux suivantes qui sont relativement indépendantes.

3.1 Brique « exacte » de résolution du *sous-problème de timing* (STP)

Nous avons vu que le *sous-problème de timing* (STP) consiste, pour un ordre fixé des avions, à calculer leur date d'atterrissage précis en respectant les contraintes du problème tout en minimisant de coût global de la solution. Le prototype fournit un exemple de « TimingSolver » dont le principe consiste à placer les avions le plus tôt possible, ce qui conduit à une solution valide mais loin d'une solution optimale du STP !

Sur le modèle de la classe `EarliestTimingSolver` [3] fourni dans le prototype, créer une *classe* `XxxTimingSolver` dont le but est de lire une solution partielle définie par un ordre donné d'avions, de calculer leur heure d'atterrissage optimale et d'enregistrer la solution complète. Vous pourrez utiliser la Programmation Linéaire pour résoudre ce problème [4] [5]. Dans ce cas le solveur sera appelé `LpTimingSolver`.

Vous pourrez tester cette brique *de timing* en passant sur la ligne de commande le fichier d'instance traité et soit un fichier de solution partiel (contenant la ligne `order`), soit directement l'ordre des avions (contenu dans la ligne `order`) en utilisant l'option `--planes`. Par exemple, la commande suivante permet à partir d'un fichier correspondant à la plus petite instance et de l'ordre optimal des avions de reconstruire la solution optimale (connue) :

```
./bin/run.jl timing --planes p3,p4,p5,p8,p6,p7,p9,p1,p10,p2 data/01.alp
=> reconstruit la solution optimale à partir de l'ordre des avions
```

Le rapport devra décrire en détail le modèle ou l'algorithme utilisé et justifier son optimalité. Vous préciserez également les performances de cette brique en nombre de résolutions par seconde pour chacune des instances proposées (en particulier les instances 09 et 13).

Question subsidiaire Le prototype vous propose une action `dmip` pour la résolution exacte du problème (que nous appellerons "approche frontale") via une formulation PLNE avec temps discrétisé. Cette méthode qui est implantée dans la classe `MipDiscretSolver` permet de résoudre de toutes petites instances [6] pour une fonction de pénalité arbitraires des avions.
Cette méthode est utilisable par la commande :

```
./bin/run.jl dmip data/01.alp  # -x cbc par défaut
./bin/run.jl dmip data/01.alp -x gurobi
```

En exploitant la fonction de coût simplifiée du problème Seqata, proposer une extension de votre modèle PLNE de la brique STP pour résoudre le problème de façon exacte. Pour cela vous créerez une nouvelle classe `MipSolver` en vous inspirant de la classe `MipDiscretSolver`.

3.2 Briques d'exploration locale (`ExploreSolver` et `DescentSolver`)

Les méthodes d'exploration de voisinages sont à la base de la plupart des méta-heuristiques et vous en aurez besoin pour la question suivante. Afin d'alléger le travail à réaliser et de vous guider dans l'organisation du code, le prototype contient déjà des méthodes implantant quelques opérateurs de voisinage paramétrables (dans la classe `Solution`), une méthode d'exploration aléatoire (classe `ExploreSolver`) ainsi que le squelette d'un solveur réalisant une descente.

3. Le terme *classe* est utilisé dans un sens d'action du programme et regroupe à la fois le type et les méthodes spécialisées pour manipuler ce type. Il regroupe aussi la partie du programme principal destinée à l'exploiter ou à le tester.

4. D'autres méthodes plus efficaces mais plus lourdes que la Programmation Linéaire peuvent être utilisées comme la Programmation Dynamique.

5. Un exemple d'utilisation de la bibliothèque de modélisation mathématique `JuMP.jl` est proposé dans la classe `MipDiscretSolver`. Celle-ci permet de résoudre de manière exacte (pour les très petites instances) une version en temps discrétisé du problème initial en utilisant la PLNE.

6. De plus, autant le solveur gratuit Cbc est équivalent en performance pour la résolution de la brique STP de la première question, autant la différence de performance est énorme entre les solveurs commerciaux et gratuits pour la résolution par l'approche frontale en temps disrétisé (nombre de variable binaire énorme).

La première étape de cette question consiste donc à compléter la classe `DescentSolver` de façon à rendre opérationnelle la commande de test suivante [7] :

```
./bin/run.jl descent -n 100  --presort target  data/05.alp  # => solution de coût 3100
```

Pour cela, vous pourrez vous inspirer de la classe `ExploreSolver` qui accepte systématiquement son nouveau voisin obtenu par une inversion de deux avions arbitraires. [8].

Le voisinage idéal est généralement le résultat d'un compromis entre la *rapidité de convergence* et la *qualité de la solution* obtenue. Votre travail pour cette question consistera à proposer un voisinage (qui peut être un tirage aléatoire entre plusieurs voisinages élémentaires prédéfinis) permettant de résoudre au mieux les 5 plus grosses instances (09 à 13) en 10 minutes maximum sur la machine de référence. Pour cela vous pourrez exploiter l'option `--duration` alias `--dur` intégrée au prototype de la classe `DescentSolver` :

```
./bin/run.jl descent --dur 600  data/09.alp
```

3.3 Approche globale par *Steepest Descent*

Dans la question précédente, il s'agissait d'explorer des opérateurs de voisinage avec tirage aléatoire du voisin à tester. Cette technique permet d'explorer des voisinages à la fois relativement larges (en nombre de voisins) et complexes à construire (réunion de voisinage élémentaires, ...) mais sans garantie de convergence vers le minimum local [9].

Le principe de l'algorithme de *Steepest Descent* consiste à définir précisément un voisinage (i.e ensemble de voisins) et à en garantir l'exploration complète. L'algorithme s'arrête quand le minimum local est atteint. Une des difficultés consiste à définir implicitement ou explicitement cet ensemble d'une manière suffisamment générique pour pouvoir tester facilement différents opérateurs de voisinage. Par ailleurs, différentes stratégies sont possibles (voir le cours) : acceptation de la première solution améliorante [10] sans attendre l'exploration complète, pré-construction dans un tableau de l'ensemble des mouvements à explorer, ...

Votre travail consiste à créer un nouveau solveur `SteepestSolver` (action `steepest`) en vous inspirant du `DescentSolver` de la question précédente. Le rapport détaillera les différents voisinages étudiés avec pour chacun d'eux sa taille, la durée de calcul et la valeur du minimum local obtenu sur l'ensemble des instances. En particulier vous comparerez les perforances des résultats par rapport à la descente simple de la question précédente.

3.4 Approche globale à voisinage variable (VNS)

Vous avez vu que l'algorithme de descente pouvait fournir de bonnes solutions, mais que la qualité du résultat et la vitesse de résolution est très sensible au choix du voisinage utilisé pour l'exploration. En particulier la descente précédente ne permet pas de jouer sur les crières d'intensification de de diversification. En utilisant le résultat du cours, proposer une approche de résolution globale par *recherche à voisinage variable* (VNS). Sur le modèle de la classe `DescentSolver`, créer une classe `VnsSolver` et ajouter le code nécessaire pour l'intéger dans votre projet (options, require, ...). Vous pourrez choisir la variante de l'algorithne VNS la mieux adaptée au problème.

Le rapport détaillera et justifiera les choix effectués et les résultats obtenus sur l'ensemble des instances.

7. Celle-ci effectue une descente à partir du résultat du glouton standard (tri sur l'attribut target) avec comme critère d'arrêt un maximum de 100 itérations non améliorantes.

8. Cette métaheuristique est certes stupide, mais contrairement à une descente, elle garantit grace à son opérateur de voisinage « couvrant » de résoudre la plus petit instance à l'optimum (avec un peu de patience...)

9. Ce minimum local est associé non seulement à la solution initial et à l'opérateur de voisinage adopté, mais également aux tirages aléatoires des voisins acceptés au cours de la descente (contrairement à une *Steepest Descent* pure).

10. Si vous voulez tester cette alternative, vous pouvez prévoir une nouvelle option booléenne `--first_best` alias `--fb` qui vous permet, si elle est active, d'accepter la première solution améliorante rencontrée dans un voisinage.

3.5 Approche par décomposition en tranches (optionnelle)

L'objectif de cette dernière question optionnelle[11] comporte à la fois une partie principale théorique suivie si possible d'une implantation informatique. Elle consiste à décomposer le problème initial en tranche dans le sens ou l'on se contente de résoudre un sous-problème défini par un sous-ensemble des avions de l'instance complète.

Partie théorique

- Observer les fichiers d'une solution relativement bonne obtenue pour les grosses instances, en particulier observer le sens des décallages des dates d'atterrissage par rapport à leurs dates souhaitées (`target`), ainsi que la marge éventuelle sur la contrainte de séparation entre un avion et ses voisins immédiats.
- Soit une tranche d'avion définie par l'intervalle $[idx_{first}, idx_{last}]$ dans l'ordre de la solution courante. Déterminer les conditions à respecter sur les avions situés en idx_{first} et en idx_{last} pour que la résolution du *sous-problème de timing* de la solution courante sur la tranche $[idx_{first}, idx_{last}]$ soit équivalente à sa résolution sur l'intervalle $[1, n]$.
- Définir en pseudo-code une méthode de la classe `Solution` qui retourne une partition des sous-listes d'avions respectant les conditions de séparation ci-dessus. Au pire des cas, cette partition sera réduite à un singleton constitué de tous les avions dans l'ordre de la solution courante !

Partie pratique

- Implanter une méthode de la classe `Solution` qui retourne une partition des avions respectant les conditions de séparation précédentes.
- Ajouter un nouveau constructeur d'Instance pour créer une instance partielle à partir de l'instance originale complète et d'un sous-ensemble d'avions.
- Résoudre cette instance partielle soit à l'aide de votre meilleure méta-heuristique, soit en utilisant le solveur exact fourni dans le prototype et accessible via l'action `dmip` (méthode PLNE frontale résolvant le problème en temps discrétisé[12]).
- Créer un nouveau solveur global `SliceSolver` (action `slice`) exploitant la statégie décrite ci-dessus qui :
  - effectue une première descente rapide pour disposer d'une solution de départ de qualité correcte,
  - extraire les sous-instances de la solution obtenue par la partition décrite précédement et résoudre séparément chaque tranches par votre meilleur solveur,
  - recontruire la solution finale par concaténation des solutions des instances partielles et l'enregistrer.

3.6 Travail à réaliser et évaluation

3.6.1 Le programme

Le programme pourra être réalisé dans le langage de votre choix bien que C++ ou Julia soit fortement conseillé. L'utilisation du prototype Julia n'est pas une obligation, mais vous devrez impérativement respecter les spécifications suivantes :

- **spec1** : le répertoire du projet (et donc l'archive compressée) sera nommé en fonction de votre numéro de groupe qui vous sera communiqué ultérieurement, par exemple `seqata_g02` pour le deuxième groupe. Il contiendra un fichier README précisant la manière de compiler et d'utiliser le projet **en ligne de commande**.
- **spec2** : il devra être utilisable en ligne de commande (non interactive) depuis la machine salle du réseau d'enseignement de l'ENSTA et gérer (au minimun) les paramèters suivants :

  ```
  ./bin/run.jl <action> [options] <chemin_instance> [<chemin_solution>]
  ```

  Avec
  `<action>` : une des actions parmi `timing`, `descent`, `steepest`, `vns`.

11. Sauf pour les quadrinômes pour lesquels cette question est obligatoire.
12. Cette approche qui crée une variable binaire pour chaque date et chaque avion s'affranchit de la contrainte de convexité mais est limitée à de très petites instances.

`<chemin_instance>` : chemin d'accès au fichier d'instance,
`<chemin_solution>` : chemin vers le fichier solution à enregistrer.
Si le chemin de la solution est omis, une solution avec un nom par défaut sera générée **dans le répertoire courant** [13] (et surtout pas dans le répertoire contenant les fichiers d'instance !).

Par défaut, votre programme n'exploite qu'un seul cœur de votre ordinateur alors que les algorithmes sont en partie séparables et se prêtent donc particulièrement bien à la **parallélisation du code**. **Si vous maitrisez déjà ce domaine**, vous pouvez exploiter les fonctionnalités de parallélisme du langage utilisé pour améliorer l'efficacité de vos solveurs.

3.6.2 Organisation du travail

Compte tenu des diverses occasions d'isolement (vacances, reconfinement, ...), vous serez probablement amenés à collaborer entre vous à distance. Par conséquent chaque groupe (trinôme) partagera un dépôt git **privé** (github, gitlab ou autre). Vous prendrez garde à ce que la branche *master* (ou *main*) du dépôt partagé contienne en permanence une version fonctionnelle de votre code (cette branche ne doit pas être un brouillon !). Vous ajouterez à la racine du projet un fichier `hist.md` *résumant* les contributions datées de chaque membre du groupe. Les retours du code à l'encadrement se feront alors simplement par la communication de l'url du dépôt git.

3.6.3 Le rapport final

Le rapport final devra traiter en détail les questions suivantes :
— la brique exacte de résolution du *sous-problème de timing* et ses performances,
— les voisinages adoptés pour vos algorithmes de descente (aléatoire et *steepest*),
— les choix adoptés et le fonctionnement de votre algorithme à voisinage variable,
— éventuellement la partie théorique de l'approche par décomposition en tranches.

En plus de la description propre à chaque méthode, le rapport devra fournir un tableau récapitulatif des résultats obtenus sur *tous* les scénarios de test proposés (quitte à laisser des cases vides pour les instances non traitées) ainsi qu'une analyse critique des résultats en proposant des axes d'amélioration (autres méthodes, hybridations, …). Vous préciserez également pour chaque instance les réglages de vos algorithmes (commandes et options à taper, ...) pour l'obtention des résultats fournis.

L'évaluation finale du projet tiendra compte :
— du programme (résultats obtenu, facilité d'utilisation et qualité de l'implantation) fourni le jour de la soutenance ;
— du travail et des résultats obtenus sur chacune des parties détaillées précédemment et évaluées indépendamment ;
— de la qualité globale du rapport incluant l'analyse critique ;
— de la qualité de la présentation orale du projet (clarté et esprit de synthèse).

3.6.4 Calendrier

— Pour le **vendredi 9 décembre 2022**, vous nous communiquez la composition de votre groupe (typiquement un trinôme). Le prototype de projet en Julia vous sera alors fourni par email.
— Pour le **vendredi 16 décembre 2022**, chaque groupe devra remettre (par couriel à l'encadrement) le lien vers le dépôt git du programme répondant aux questions 3.1 et 3.2. Votre code sera donc capable d'exécuter les commandes suivantes [14] :

```
./bin/run.jl carlo --itermax 1  data/01.alp
=> trouve l'optimum de l'instance 01.alp de coût 700
./bin/run.jl descent -n 50  --presort target  data/05.alp
=> trouve l'optimum de l'instance 05.alp de coût 3100
```

13. Cependant, le prototype fourni par défaut les solutions sous un nom automatiquement déterminé en fonction du nom de l'instance et du coût de cette solution. Le fichier est enregistré dans le sous-répertoire `_tmp/` du projet. Ce répertoire est modifiable par l'option `--outdir` alias `-d`

14. La première commande exécute une méthode de Monte-Carlo consistant à itérer la construction d'une liste d'avions dans un ordre arbitraire, puis à résoudre le *Sous-Problème de Timing*. Dans le cas particulier (d'intérêt limité pour une méthode de Monte-Carlo !) dans lequel on ne veut qu'une seule itération, on choisit alors l'ordre le plus prometteur en triant les avions sur leur attribut T_i.

Si vous utilisez le prototype Julia fourni, profitez-en pour vous appropriez le code en en maitrisant l'organisation. Si vous souhaitez utiliser un autre langage (C++, Java) vous devrez recoder une version simplifiée du prototype en en respectant les fonctionnalités (gestion des options, ...).

— Pour le **vendredi 6 janvier 2022**, chaque groupe devra remettre par courriel le lien du dépôt git du programme permettant de résoudre les instances proposées par l'algorithme `SteepestSolver` ainsi qu'un prérapport décrivant les briques nécessaires à sa construction.

— Pour le **lundi 23 janvier 2022**, chaque groupe devra remettre par courriel le rapport final avec le tableau de synthèse préliminaire des solutions obtenues.

— Pour le **vendredi 27 janvier 2022**, jour de soutenance, vous remettrez par courriel une archive du code définitif. Une annexe comportant des corrections éventuelles ou des résultats complémentaires pourra être remise également. Le corps du rapport final fourni ne doit pas être modifié ; toutes les modifications, ajouts et évolutions devront apparaître en annexe.

Références

[1] John E Beasley. "Obtaining test problems via internet". In : *Journal of Global Optimization* 8.4 (1996), p. 429-433. URL : `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/airlandinfo.html`.

[2] John E Beasley et al. "Scheduling aircraft landings—the static case". In : *Transportation science* 34.2 (mai 2000), p. 180-197. DOI : `10.1287/trsc.34.2.180.12302`.

[3] Maurice Diamantini, Alain Faye et Julien Khamphousone. "Calcul des dates d'atterrissage d'une séquence d'avions pour des fonctions de coût convexes et affines par morceaux". In : ROADEF'2019 - Le Havre, fév. 2019. URL : `https://hal.archives-ouvertes.fr/hal-02354405`.

[4] Alain Faye. "A quadratic time algorithm for computing the optimal landing times of a fixed sequence of planes". In : *European Journal of Operational Research* 270.3 (nov. 2018), p. 1148-1157. DOI : `10.1016/j.ejor.2018.04.021`.