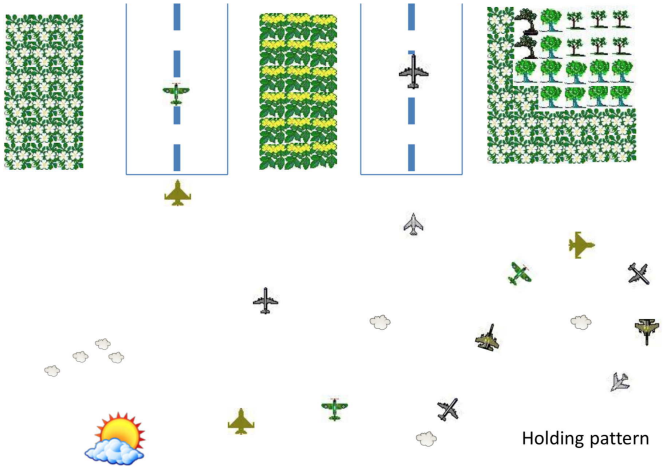
Projet SEQATA
SÉQuencement d'ATterrissage d'Avions

Cours SOD324-MH (2022-2023)

Agnès PLATEAU
Maurice DIAMANTINI et Natalia JORQUERA.

02/12/2022

Le problème d'atterrissage sur un aéroport



Pour SEQATA $\Rightarrow$ une seule piste d'atterrissage !

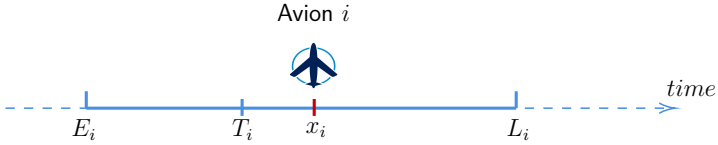
Le nom du problème est SEQATA pour Sequencement d'Atterr. d'Avion.
Le contexte du problème est le suivant :

- Un aéroport est constitué de plusieurs pistes d'atterrissage.
- Et on connait avec qq heures d'avance l'arrivée d'un ensemble d'avions.
- Chaque avion a une heure préférencielle d'atterrissage...
- mais c'est le contrôleur de l'aéroport qui doit définir sur **quelle piste** et à **quelle heure précise** chaque avion doit.
- Au besoin il est amené à demander à l'avion d'accélérer ou de retarder son atterrissage (par un détour).

- pour notre projet nous allons supposer que l'aéroport **n'a qu'une seule piste d'atterrissage**.

Next:
Le contrôleur var donc imposer les dates d'atterrissage x_i mais... Les dates d'atterrissage de chaque avion sont bornées par un intervalle $[E_i, L_i]$

Contraintes de fenêtre de temps de chaque avion



A : ensemble des n indices d'avion

E_i : earliest time

T_i : heure d'atterrissage préférée (target time)

L_i : latest time

x_i : heure d'atterrissage calculée

Date d'atterrissage de chaque avion bornée

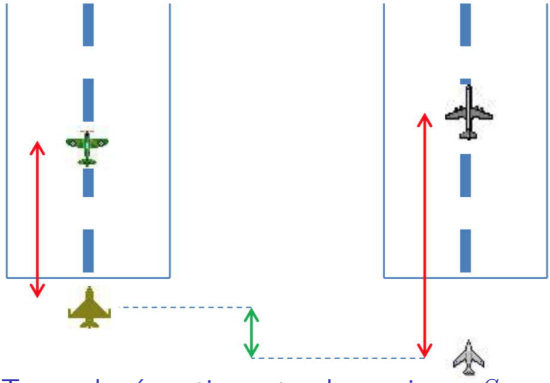
$$E_i \leq x_i \leq L_i \quad \forall i \in A$$

Le contrôleur va donc imposer les dates d'atterrissage x_i.
- Mais d'abord, sa date d'atterrissage effective est bornée par un intervalle :
  - au plus tôt E_i car sa vitesse est limitée
  - au plus tard L_i car même si on lui fait faire un détour ou des boucles d'attente, sa quantité de carburant est limitée !
- Ensuite, chaque avion a une heure préférée d'atterrissage T_i.

Si c'était la seule famille de contrainte, le problème serait facile car séparable... Mais il y a des **contraintes de couplage**...

Next:
En effet, un avion génère des **turbulances derrière lui** en fonction de sa catégorie
... des temps de séparation entre avions

Contraintes de temps de séparation entre avions



Temps de séparation entre deux avions : S_{ij}

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

Avec S_{ij} : matrice carrée non symétrique.

(S_{ij}^{kl} si plusieurs pistes !)

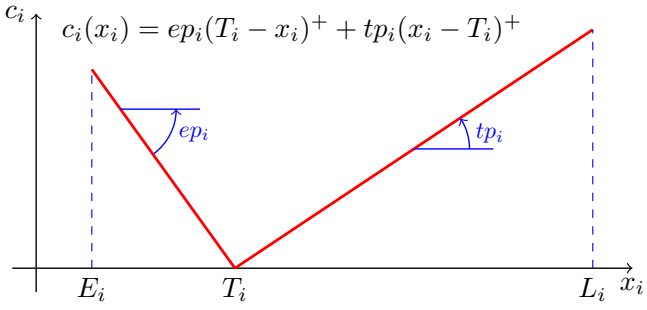
Un avion génère des **turbulances derrière lui** en fonction de sa catégorie :
- un gros porteur va laisser derrière lui une trainée importante qui aura d'autant plus d'effet que l'avion suivant est léger.
- on impose donc une durée de séparation minimum entre deux avions successifs atterrissant sur un même pistes
- mais aussi entre avions sur **deux pistes voisines (moindre mesure)**
- dépend du type des avions
- Ces délais de séparation sont définis par S_{ij} matrice non symétriques (e.g deltaplane derrière un gros porteur !)
- pour Seqata, on ne considère qu'une seule piste

Next:

Fonction de coût de chaque avion : pénalités

Coût de pénalité d'un avion et objectif

Fenêtre de temps pour chaque avion



Objectif du problème

$$\min_x \sum_{i \in A} c_i(x_i)$$

- On a vu que notre avion a une heure préférée d'atterrissage T_i entre E_i et L_i.
- Si on s'en écarte, on doit payer une pénalité.
- nous allons considèrer une penalités linéaire d'avance et de retard.
- mais en pratique, on peut avoir des fonctions de coût plus complexe : convexes linéaire par morceau, non convexes, ... (cf sujet).
- on pourrait aussi vouloir minimiser le plus grand écart par rapport au target sur l'ensemble des avions

Next:
Nous avons maintenant toutes les information sur le problème,
Examinons les données d'une instance...

Les données d'une instance

```
name alp_01_p10
nb_planes 10
nb_kinds 2
freeze_time 10 # inutile pour seqata (problème dynamique)

#    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
```


- Chaque ligne contient une information indépendante,
- Les lignes vides sont à ignorer,
- Les lignes commençant par # sont des commentaires à ignorer,
- la ligne commençant name fourni le nom de l'instance,
  (vont de 01 à 13, les 8 premières étant des instances jouets)
- les lignes `nb_planes` (deviner) et `nb_kinds` nbr de type d'avions différents,
  (`freeze_time` n'est pas utilisée)
- chaque ligne plane décrit un avion particulier (dont catégorie)
- La colonne `at` (appearing time) n'est pas utilisée
- Les lignes sep définissent des durées de sép. entre catégorie d'avions.
  Se lit :
  la durée de sép quand cat(i) est suivi par cat(j) vaut S_{ij}.

Next:
Résoudre cette instance consiste au final à générer un fichier
de la solution au format suivant...

Format de la solution et validateur 1/2

```
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  ...
landing   p4 107  0   0.0      # E=97  T=107 L=522  ...
landing   p5 119 -5 150.0      # E=111 T=124 L=556  ...
landing   p6 127 -9 270.0      # E=121 T=136 L=577  ...
landing   p7 135 -4 120.0      # E=125 T=139 L=578  ...
landing   p8 143  2  60.0      # E=127 T=141 L=574  ...
landing   p9 151  0   0.0      # E=136 T=151 L=592  ...
landing   p1 166 10 100.0      # E=130 T=156 L=560  ...
landing  p10 181  0   0.0      # E=161 T=181 L=658  ...
landing   p2 259  0   0.0      # E=196 T=259 L=745  ...
```

Détail des commentaires sur chaque ligne (facultatif)

- Les formats d'instance et de solutions sont imposés, (mais pas les commentaires)
- **Toute solution devra être acceptée par un validateur**

Next:

ZOOM de la solution pour deux avions p7 et p8

Format de la solution et validateur 2/2

```
...
cost 700.0
order [p3,p4,p5,p6,p7,p8,p9,p1,p10,p2]
...
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-
...
```

Détail des commentaires sur chaque ligne (facultatif)

- rappel des caractéristiques de l'avion,
- indique l'écart réel et minimum avec ses plus proches prédécesseurs,
- $\Rightarrow$ utile pour déboguer votre code !

Tout d'abord, ce format est valide (commentaires sur deux lignes),

Next:
Deux approches de résolution possibles (frontal ou par niveau)

Deux approches de résolution possibles

Approche exacte du problème complet **Taille limitée**

- Pour chaque avion, chercher sa date d'atterrissage,
- Plusieurs modélisations exactes possibles par PLNE,
- $\Rightarrow$ valable pour pour les petites instances,
- $\Rightarrow$ voir exemple en PLNE dans le proto fourni.

Décomposition en deux niveaux successifs **Pour Seqata !**

- **étape 1** : chercher un ordre optimal d'atterrissage des avions
  $\Rightarrow$ définir une *permutation* des avions,
- **étape 2** : Pour un ordre d'avions donné, chercher la date précise d'atterrissage des avions
  $\Rightarrow$ pb connu sous le nom *Sous Problème de Timing* (STP).

Le proto fourni un exemple de solveur PLNE exact pas pas efficace pour notre problème car il résoud le problème avec une fonction de coût arbitaire (e.g non lineaire, non continu, ...)

Next:
travail demandé en **quatres phases dont deux briques de base**

Travail demandé 1/4 : briques de bases

Brique exacte pour le **Sous-Problème de Timing (= STP)**

- ordre des avions fixé ⇒ trouver la date x_i de chaque avion,
- **solution algorithmique** : Programmation Dynamique (`DpTimingSolver`) **non triviale pour être efficace !**
  ⇒ aucune dépendance externe.
- **le plus simple : Programmation Linéaire**, (`LpTimingSolver`) **conseillé !**
  ⇒ nécessite solveur externe `CPLEX`, `Gurobi`, CLP, ...
  ⇒ vous disposerez d'un exemple de solveur avec JuMP : `EarliestTimingSolver`

- On s'intéresse à la résolution exacte de ce sous-problème.
- Elle devra être répétée de nombreuses fois.
- un `EarliestTimingSolver` est disponible :
  Mais celui-ci place les avions au plus tôt (dans l'ordre imposé)
  ⇒ solution sous-optimal pour le critère initial,
  ⇒ mais valide si faisable

Next: Seconde brique de base : une descente aléatoire...

Travail demandé 2/4 : briques de bases

Implanter une **méthode de descente aléatoire**

- définir/tester/choisir un voisinage couvrant,
- tirage aléatoire et acceptation éventuelle d'**un** voisin,
- compromis *largeur voisinage* vs *rapidité convergence*,
- créer une classe `DescentSolver` : **Facile** :
  $\Rightarrow$ un squelette du `DescentSolver` existe déjà !
  $\Rightarrow$ un `ExploreSolver` complet existe déjà !

- L'`ExploreSolver` effectue un déplacement systématique de voisin aléatoire en voisin aléatoire.
  - Il se contente de mémoriser toute solution améliorante rencontrée.
  - Il permet de résoudre la plus petite instance (avec un peu de patience)
- Le `DescentSolver` est presque complet, ... mais il faut bien lire le code !

Next: 3^e phase : une *Steepest Descent*
Cette descente aléatoire permet d'explorer des voisinages à la fois large (car...) et complexe (car...) ...

Travail demandé 3/4 : *Steepest Descent*

Principe

1. à chaque itération, **tester tous les voisins**,
2. adopter le meilleur voisin et passer à l'itération suivante,
3. on s'arrête quand il n'y a plus de voisin améliorant,
   $\Rightarrow$ **déterministe** pour une solution initiale donnée.
4. $\Rightarrow$ **créer nouvelle classe** `SteepestSolver`,

Stratégies et variantes

- quel opérateur de voisinage utiliser ?
- accepter le premier voisin améliorant rencontré
  $\Rightarrow$ passer à l'itération suivante **sans tout explorer**,
  $\Rightarrow$ l'exploration ne sera complète que dans le minimum local.
  $\Rightarrow$ **ajoute de l'aléa** selon ordre d'exploration.
- voisinage implicite (définir et appliquer chaque mouvement),
- voisinage explicite (préconstruire le vecteur des mouvements),
  $\Rightarrow$ **combinaisons de voisinages possibles** mais **plus difficile**,

La descente aléatoire précédente permettait d'explorer des voisinages à la fois large (car on en explore qu'un seul élément !)
et complexe (mélange probabiliste de plusieurs voisinages).
Mais celle-ci ne garantit pas l'atteinte d'un minimum local.

La *Steepest descent* garantit l'atteinte d'un minimum local car elle explore le voisinage complet.
Par contre le voisinage ne doit pas être trop grand (sinon c'est trop long !), ni trop complexe (car difficile d'être exhaustif sans être redondant).

Next: La 4^e phase : VNS
méthode à voisinage variable...

Travail demandé 4/4 : métaheuristique

Méthode à voisinage variable (VNS)

- but : implanter une méta-heuristique complète,
- ⇒ **créer nouvelle classe `VnsSolver`,**
- en exploitant les briques précédentes,
- explorer les variantes de VNS présentées en cours,
- objectif : meilleure solution possible en 1 heure,
- liberté et créativité !

Challenge !

1/2 point de bonus par record battu pour chacune des 5 grosses instances [1]

1. limité à 1.5 points

La difficulté du VNS :
- choisir de petits voisinages au départ !

Il y a aussi le travail 5/4 :
- décomposition en tranches temporelles
- optionnel mais OBLIGATOIRE SI il y a des quadrinômes

Si vous voulez coder ce projet en C++ , il faudra développer :
- les structures de données (Instance, Plane, Solution, ...),
- une "classe" pour chaque solveur avec leur méthode solve!,
- une gestion des options de la ligne de commande,
- choisir entre plusieurs solveurs,
- la lecture du fichier fichier d'instance passé en paramètre,
- construire et exécuter le solveur choisi,
- en extraire le résultats,
- enregistrer la solution.

Next: Heureusement, vous disposerez d'un **prototype de code en Julia** opérationnel.

Un prototype de programme Julia **est fourni**

AV : Le proto est **opérationnel !**

- Projet pré-organisé et fonctionnel.
- Structure de *classes* (Instance, Planes, Solution ...).
- Gestion des options déjà faite (personnalisable).
- Lecture des fichiers d'instance faite.
- Plusieurs solveurs déjà implantés (EarliestTimingSolver, ExploreSolver, CarloSolver).

INC : Le proto aide beaucoup **mais...**

- Pas mal de code à lire pour le proto.
- Voir fichier `presentation_proto_seqata.md` pour les conseils...
- `https://sod324.minisme.fr`
- `https://sod324.minisme.fr/seqata_docs/`

Next: Calendrier des retours.

Calendrier des retours

- **Avant le V/09/12/2022** *(le plus tôt possible)*
  - envoyer votre **formation de trinôme**,
  - créer un dépot **privé** sous git (github, gitlab, ...)
  - $\Rightarrow$ vous recevez le code `proto_seqata_p2023`,
  - $\Rightarrow$ à recopier sous le nom `seqata_gN` (pour le groupe N),
  - $\Rightarrow$ valider votre installation de Julia,
  - $\Rightarrow$ explorer les fonctionnalités du proto.
- **Pour V/16/12/2022** vous retourner :
  - l'url de votre **dépot git du projet avec journal**,
  - avec brique exacte STP.
  - avec `DescentSolver` (premier jet),
- **Pour V/06/01/2023** (séance de suivi de projet)
  - retour `SteepestSolver` opérationnel ou en cours,
  - prérapport avec au moins la description de la brique STP ;
- **Pour L/23/01/2023 rapport final**
  avec synthèse préliminaire des résultats.
- **Pour V/27/01/2023** (jour de l'examen)
  code définitif et annexe éventuelle au rapport.

- LEUR DEMANDE DE PRÉVOIR UN JOURNAL DE VOTRE TRAVAIL PAR PERSONNE DANS DÉPOT GIT !
- se mettre rapidement à git
- s'approprier rapidement le code du proto