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

Robin LABBÉ, Ewan SEAN
Kévin ASSOBO

December 21, 2021

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 couts 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;

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.

2.2 Résolution

La résolution du sous-problème de timing est implémenté 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 Algorithmes de descente

3.1 Voisinnages envisagés et utilisés

3.2 Performances

4 VNS

5 Synthèse

5.1 Résultats

5.2 Critiques

5.3 Commandes à utiliser

6 Conclusion

7 Annexes

Listing 1: 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
@constraint(sv.model, [i in 1:n,j in 1:n; i<j],
    x[j] >= x[i] + get_sep(sol.inst, planes[i],planes[j]))

# Objectif
@objective(sv.model, Min ,
    sum(planes[i].ep * x_ep[i] + planes[i].tp * x_tp[i] for i in 1:n))


# 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
```