Rapport de Projet

Tom Simula
Daniel Azevedo Gomes
Sébastien Prud'homme Gateau

Université de Montpellier

Rappel des Consignes :

"*La compagnie aérienne GLIA-Airlines souhaite équiper sa flotte d'avions avec des séparateurs*
de classes, permettant ainsi des configurations et des tailles de classes différentes.
La compagnie propose des sièges dans les catégories "First Class", "Business Class" et "Economy
Class". De plus, elle offre la location de ses sièges à d'autres compagnies aériennes.
Cependant, les séparateurs sont coûteux et occupent un espace non négligeable à bord de l'avion.
L'objectif est de positionner n séparateurs dans un avion ayant une capacité de m blocs de sièges.
Il est impératif de ne pas installer de séparateurs au niveau des sorties de secours (au moins une sortie). De plus, il est essentiel que la classe à l'avant de l'avion dispose d'au moins deux blocs.
Afin de permettre des configurations avec des tailles de classes variées, il est nécessaire de garantir
que les distances entre les séparateurs sont différentes.

Question 1

• Proposez une solution sans Programmation par Contraintes et fournissez une implémentation en Java pour résoudre le problème des séparateurs de classes avec la fonction
dividers(n, m, exits).

Question 2

• Testez votre solution sur les instances fournies dans ce projet.

Question 3

(X, D, C).• Maintenant, modéliser dividers(n, m, exits) sous forme d'un réseau de contraintes

Question 4

• Mettez en œuvre votre réseau de contraintes sous Choco en utilisant les fichiers fournis dans ce projet tout en respectant les principes SOLID.

Question 5

• Testez la version Choco sur les instances fournies dans le projet.

Question 6

• Optimisez votre modèle en expliquant en détail les améliorations apportées, telles que l'utilisation de contraintes globales, l'élimination des symétries, et la réduction des contraintes redondantes.

ReadMe pour l'application :

Le code complet de l'application est disponible sur GitHub à cette adresse : https://github.com/TomSimula/CP_Airplane.git

Lors du lancement de l'application, veuillez éditer les options de lancement comme indiqué ci-dessous:

java LigtningAirlines.jar [-b] [-a] [-t <timeout in ms>] [-i <aircraft instance>] [-h]

-b,--basic	Chercher la première solution via un **algorithme itératif**. **Attention** : sur les instances fournies le temps de calcul est exponentiel, lancer le programme sur des instances supérieures à la troisième risque de prendre un temps conséquent
-a,--all	**chercher toutes les solutions** Lors de l'utilisation **avec le solveur Choco**, permet de lister toutes les solutions qui satisfont le problème et non pas la première trouvée.
-h,--help	Afficher ce message d'aide
-i,--instance <aircraft instance>	**Instance du Problème à résoudre** (de inst0 à inst10)
-t,--timeout <timeout in ms>	Set the timeout limit to the specified time
-c,--compare	search solution with different strategies and compare them

Exemple : recherche de la première solution de l'instance décrite dans le sujet avec Choco :
cocoAirlines -i inst0

Réponses aux questions :

1 et 2 : utiliser l'option '**-b**' et spécifier l'instance de test précédé de '**-i'**
4 et 5 : spécifier simplement l'instance de test précédé de '**-i'**

Question 3 :

Modélisation sous forme d'un réseau de contraintes (X,D,C)
X = {position des N dividers}
D = {[0, …, M]} où M est égal au nombre de blocs.
C = {
Pas séparateurs au niveau des sorties de secours,
La classe à l'avant de l'avion doit disposer d'au moins deux blocs,
Les distances entre les séparateurs doivent être toutes différentes.
}

Question 6 :

optimisations sur les contraintes :

- **Model.increasing** : Permet de chercher une solution contenant les valeurs de position des dividers dans l'ordre croissant => Cela évite à la fois d'avoir des dividers dans le désordre et d'avoir des redondances de solutions (divider 1 et 2 interchangés par exemple)

Recherche de stratégies :

- Les stratégies suivantes ont été testées :
  - *minDomLBSearch,*
  - *minDomUBSearch,*
  - *intVarSearchSearch*
  - *inputOrderLBSearch*
  - *inputOrderUBSearch,*
  - *activityBasedSearch,*
  - *randomSearch*

Nous avons pu observer une réduction générale du temps de recherche pour presque toutes les stratégies établies.

voici les résultats de ces stratégies selon différents critères (tests réalisés sur l'instance 4):

Stratégie	**temps d'exécution**	**nombre de backtracks**	**fails**	**nombre de noeuds**	**vitesse de parcours de noeuds** *(en noeuds par sec)*
par défaut	20.648 sec	594 813	297 390	297 423	14404.6 n/s
activityBasedSearch	8.929 sec	189 185	94 551	94 634	10598.1 n/s
inputOrderLBSearch	10.425 sec	207 709	103 838	103 871	9964 n/s
minDomLBSearch	11.853 sec	275 821	137 894	137 927	11.636 n/s
intVarSearchSearch	12.301 sec	184 903	92 435	92 468	7516.9 n/s
inputOrderUBSearch	15.083 sec	424 739	212 353	212 386	14080.9 n/s
minDomUBSearch	15.511 sec	444 123	222 045	222 078	14317.7 n/s
randomSearch	294.935 sec	5 488 341	2 744 154	2 744 187	9 304.4 n/s

On peut remarquer que seule la stratégie aléatoire à un effet négatif sur tous les critères.
Les autres ont l'effet inverse et sont toutes meilleures que la stratégie par défaut.
Cependant ces résultats peuvent varier selon l'instance, il sera donc nécessaire de tester les stratégies pour chaque cas si on tient à en déduire la plus efficace.