Challenge d'AG41

Projet d'AG41 – Méthode exacte

Belkacem Lahouel, Luc Cadoret

RESUME

UTBM, P2014. Rapport de projet d'AG41 : Optimisation et recherche opérationnelle. Etude de la tournée d'un voyageur de commerce, avec des contraintes de temps et de stockages.

Table des matières

Introduction

Dans le cadre de l'UV AG41 nous a été proposé un challenge d'optimisation des coûts. Ce challenge comporte un énoncé plutôt simple : des clients commandent des produits qui doivent arriver avant une certaine date. Le but est d'effectuer ces livraisons à l'aide d'un transporteur ayant une certaine capacité, tout en respectant les contraintes de temps.

Nous a été proposé deux approches du sujet : une méthode approchée, avec plus de paramètres au problème, ainsi qu'une méthode exacte. Nous nous sommes penchés sur cette dernière, et avons donc fait tout notre possible pour trouver une solution exacte aux instances du problème donné, et ce en le moins de temps possible.

Dans ce rapport, nous exposerons en premier lieu notre compréhension du problème et les contraintes de conception qu'il implique. Nous expliquerons ensuite la façon donc nous avons choisi d'implémenter la solution, et pour finir, nous expliquerons en détail le fonctionnement des outils de résolution, à savoir l'heuristique ainsi que le parcours de l'arbre des solutions avec élagage.

Problématique amenée par le sujet

L'énoncé peut paraître plutôt simple aux premiers abords, mais s'avère plus complexe quand on s'y penche un peu. En effet, il s'agit d'envoyer des lots dans un certain **ordre** aux clients, et ce en minimisant les coûts, mais il est très complexe **d'établir ces lots**, puisque toutes les dates de livraisons effectives de ces lots sont liées entre elles à cause des temps d'allers/retours chez le fournisseur. Par exemple, si je décide de livrer un produit à une date de 300, et qu'un autre doit être livré à la date 350, mais que l'aller-retour ne prends 100, on devra décaler le premier et le livrer finalement à 250. Il est aussi important de choisir la bonne date de départ, pour minimiser les coûts de stockage et donc les coûts totaux.

Le fait que toutes les dates et les lots soient fortement liés entre eux nous a amené à deux conclusions :

- Rechercher toutes les solutions sous forme d'arbres. De cette façon, nous pouvons rapidement élaguer les branches sans potentiel, mais aussi parcourir absolument toutes les possibilités de lots et d'ordre de livraisons pour chaque instance du problème. En effet, nous pensons qu'il n'existe pas de formule mathématique qui nous donne instantanément la solution au coût minimum, et qu'il faut donc regarder chaque solution possible.
- Lorsqu'on cherche une solution, mieux vaut le faire en partant du dernier lot livré (backtracking), puisque cela nous offre un point d'ancrage au niveau des dates : on sait que le dernier lot livré le sera à telle date, et on peut ainsi évaluer le coût de livraison des autres lots.

Nous avons donc tenté de construire une méthode de résolution à partir de cette analyse de la problématique.

Implémentation

Nous avons choisi d'implémenter notre algorithme grâce à un langage orienté objet, qui nous permettait de représenter plutôt intuitivement les paramètres de l'énoncé. Entre Java et C++, nous nous sommes penchés sur C++, qui offre une plus une meilleure performance aux dépens d'une implémentation plus difficile. Nous avons un petit regret pour notre choix, puisque le C++, de par son exigence, nous a parfois joué des tours. Il nécessite plus de connaissances du langage, et entraîne des problèmes de gestion mémoire (par exemple) non essentiels dans le cadre d'un tel projet.

Voici les principales structures de données présentes dans le programme :

Classe Probleme :

C'est la classe principale du programme. C'est cette classe qui regroupe toutes les données du problème.

Attributs :

- Capa : la capacité maximum du transporteur
- Eta : le coefficient de coût d'un trajet
- Clients : la liste de tous les clients du problème
- Produits : la liste de tous les jobs du problème
- bestSol : la meilleure solution trouvée au problème. Au début, il n'y en a pas. Une solution est une liste de Batches. Leur ordre dans la liste est l'ordre dans lequel il faut les envoyer.
- evalBestSol : l'évaluation de la meilleure solution trouvée

Méthodes :

- heuristique : créé une solution heuristique
- solve : trouve la solution exacte au problème

Il y a beaucoup d'autres méthodes que nous n'énumèrerons pas, puisqu'elles représentent des sous-méthodes nécessaires au bon fonctionnement de « heuristique » et de « solve ». Nous les cachons aux yeux de l'utilisateur en les mettant privées, ce dernier n'a besoin d'utiliser que l'interface de la classe.

Classe Batch

Un batch est un lot de produits. Quels batchs envoyer et dans quel ordre, c'est ce qu'on va essayer de trouver lorsque nous nous attaquerons à la résolution de ce problème.

Attributs

- produits : la liste de produits qui sont inclus dans le batch.
- Date_livraison : la date à laquelle un batch a été livré. Cette date ne peut être trouvée qu'après avoir trouvé la solution complète dans laquelle ce batch est inclus.
- Cout_st_cour : utilisé ponctuellement pour connaitre le coût de stockage du batch
- dateGlobale : date due globale du batch. Elle correspond à la date due minimum parmi les dates dues des produits contenus dans le batch.

Tout comme la classe Probleme, il y a quelques méthodes pas forcément pertinentes à détailler.

Classe Produit

Attributs

- i : le numéro du produit (son ordre dans la liste de jobs)
- date : la date maximale à laquelle il doit être livré
- client : le client à qui est destiné ce produit

Classe Client

Attributs

- h : le numéro du client
- dist : la distance entre lui et l'entrepôt
- cost : le coefficient de stockage chez ce client

Nous ne parlerons pas de la classe Parser, chargée de découper le fichier de données, qui ne constitue qu'un outil non important dans le cadre de la résolution de ce problème.

Fonctionnement de l'heuristique

La méthode de résolution choisie est un arbre avec élagage. Pour que cette méthode soit la plus efficace possible, il convient de trouver une bonne heuristique, qui nous permettra d'élaguer rapidement les branches inutiles.

Construction des lots

La première étape de l'heuristique consiste à construire des lots de produits fixes. On va chercher à regrouper ensemble les produits ayant une **date due proche**. Voici la méthode employée pour construire ces lots :

- Tant que tous les produits n'ont pas étés placés dans des lots :
    - Prendre le produit ayant la date due la plus élevée parmi les produits restants, et créer un nouveau lot avec.
    - Tant que le lot ne dépasse pas la capacité du transporteur et qu'il reste des produits pour ce client :
        - Prendre un produit pour le même client que le premier ajouté au lot.
        - Si leur différence de date due ne permet pas de faire un aller-retour, l'insérer dans le lot.
    - Fin tant que
- Fin tant que

Test des permutations

Après avoir construit des lots, la seconde étape de l'heuristique consiste à prendre ces derniers, et à effectuer une recherche dans un arbre avec élagage dessus, pour trouver leur ordre optimal. Cette méthode peut paraître un peu forte pour une simple heuristique, mais il se trouve que l'arbre est finalement parcouru très rapidement. En général, pour les instances données, le nombre de lots trouvés est de 5/6, ce qui constitue à peu près 120 solutions à calculer (5!), ce qui se fait en quelques secondes tout au plus. Cependant, il est vrai que pour des instances générant plus de lots, un travail d'optimisation serait à faire pour ne pas consacrer trop de temps à l'heuristique.

Cette méthode heuristique, bien que plutôt simple et intuitive, nous donne finalement des résultats très convaincants :

	Heuristique	**Exacte**	**Différence**
10n3cl	1273,89	1211,1	62,79
10n4cl	1876,07	1874,25	1,82
15n2cl	2754,92	2606,25	148,67
15n3cl	1999,98	1999,98	0

Résolution exacte

Pour trouver la solution optimale, nous avions en premier lieu pensé à ne construire une solution sous forme de liste de produits. Nous testerions d'envoyer les produits un par un au client, et les regrouperions dans le cas où deux produits étaient envoyés à suivre au même client. Ce raisonnement a rapidement été abandonné, puisqu'il représentait un nombre de solutions possible de n! (soit plus d'un billion pour n=15). Nous nous sommes finalement penchés sur énumération des possibilités pertinentes.

Construction des batches

Pour trouver la solution optimale aux instances données, nous avons opté pour un branch'n'cut. Pour réduire au maximum le nombre de possibilités parcourues et optimiser le temps, nous avons décidé de construire avant de faire une recherche dans l'arbre la liste des seuls batches pertinents.

Première étape : trouver les permutations de produits (sans répétion)

Pour trouver tous les batches pertinents, il faut déjà trouver toutes les permunations sans répétitions pour chaque client. Par exemple, pour un client ayant commandé les produits 1,2 et 3, et pour un transporteur de capacité maximum de 3, on aura comme batches possibles :

[1], [2], [3], [1,2], [1,3], [2,3] et [1,2,3].

Le fait de ne pas avoir de répétions fait qu'on n'aura pas de batches comme [2,1,3], [3,2,1] etc… qui sont au final équivalents entre eux.

Seconde étape : supprimer les batches inutiles

Puisque les produits sont ordonnés par date due croissante, on aura 1 < 2 < 3 (au niveau de leurs dates). On peut donc affirmer que des batches tels que [1,3] est inutile, puisque son coût sera au moins pire que le batch [1,2]. On peut donc le supprimer.

Ces deux étapes nous permettent donc de ne pas faire un arbre à l'aveuglette, qui créerait trop de branches inutiles.

Exemple : Un client commande 4 produits, et le transporteur a une capacité de 3. Le nombre de batches possible avec répétitions et en gardant les batches inutiles est de 40. Sans répétitions, et en supprimant les batches inutiles, on descend ce nombre à 9.

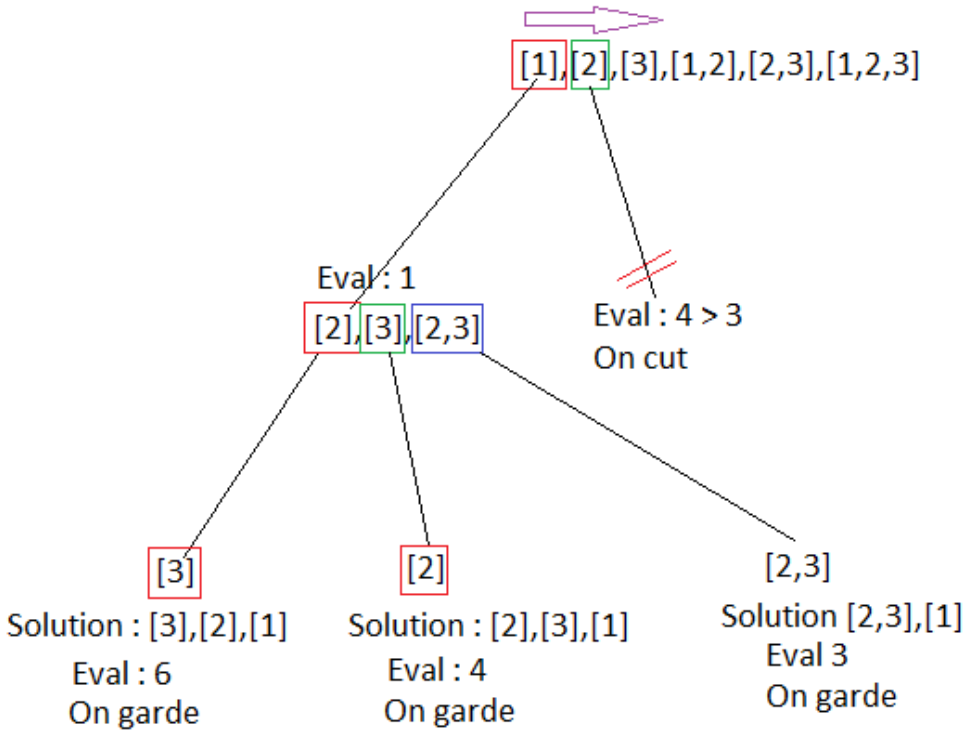
Recherche dans l'arbre avec élagage

La méthode de recherche dans l'arbre est la suivante :

- On prend un batch dans la liste créée auparavant
- On créé une nouvelle liste, qui est la même que la précédente, mais sans le batch nouvellement sélectionné, et sans tous les batches qui contiennent un ou plusieurs produits communs au batch qu'on vient de sélectionner
- On rappelle la fonction récursive avec cette nouvelle liste
- Si à tout moment l'évaluation de la branche courante est supérieure à l'évaluation de la meilleure solution, on coupe la branche

- Si on a utilisé tous les produits et que l'évaluation + le coût minimum est meilleure que l'évaluation de la meilleure solution, on l'enregistre à la place de la meilleure solution antécédente. Le calcul de ce coût minimum est détaillé dans les paragraphes ci-dessous.

Voici un schéma explicatif, pour le cas où un a seulement 1 client qui a commandé les produits 1, 2 et 3 :



Note : la construction de la solution se fait en Backtracking (on commence par la fin). Parcourir l'arbre dans l'ordre [1],[2],[3] donne donc une solution où les batches seront dans l'ordre [3],[2],[1].

Optimisation : tri de la liste des batches et recherche d'un coût minimum

On peut optimiser la recherche en triant les batches de manière à trouver plus rapidement la bonne solution et pourvoir élaguer les mauvaises solutions grâce à l'évaluation de celle-ci. Nous avons plusieurs critères de tri :

- tri 1 : on trie les batches sur leur taille, de manière à commencer par les batchs les plus gros ; ce tri nous donne de bons résultats.
- tri 2 : on calcule un coefficient en fonction de la date due ; qui nous donne également de bons résultats. Nous le conservons.
- tri 3 : par dates dues ; mais ce n'est pas suffisant : il faut également prendre en compte les coefficients de stockage, qui, s'ils sont très bas, nous poussent à choisir de livrer très tôt certains lots de produits…
- tri 4 : par coefficients de stockage, mais encore une fois ce n'est pas suffisant…

Certains critères de tris nous donnent des résultats plus intéressants en fonction du type de l'instance. La problématique (dates qui changent l'évaluation des coûts de stockage) nous empêche de trier une bonne fois pour toute la liste des batches restants à distribuer. Nous sommes obligés de retrier à nouveau à chaque itération.

Une autre optimisation, qui permet d'accélérer grandement la recherche est de considérer le coût minimum : c'est le coût que l'on payera au moins, nous en sommes toujours sûrs. Il s'agit donc des coûts de livraisons si on suppose que les batches contiennent à chaque fois un maximum de produits à livrer ; c'est le nombre minimum d'allers/retours pour chaque client.

Finalement, la dernière optimisation que nous avons implémentée consisterait à calculer les coûts minimums à chaque choix de batch : nous choisissons de livrer un batch donné, nous avons alors un nombre minimum (en supposant que l'on maximise les produits par batches par la suite) d'allers/retours à effectuer. Nous essayons à chaque fois d'évaluer ce coût pour essayer de couper plus tôt. C'est en fait une généralisation de l'optimisation précédente : au lieu de calculer ce coût minimum seulement au niveau 0 de recherche, nous le calculons à chaque étape. Cela peut être utile dans le cas où nous avons plusieurs batchs pour un même client, qui entraînent beaucoup de permutations, mais de par leurs allers/retours très nombreux ne donnent pas de bons résultats.

Améliorations possibles

Bien que nous ayons obtenu un programme qui fournit une solution relativement rapide, nous avons pensé à plusieurs améliorations qui auraient pu être intégrées, et qui pourraient grandement optimiser notre méthode de résolution :

- Ajouter des Threads : En effet, un processeur multi-cœur ne peut être utilisé pleinement que si plusieurs threads sont implémentés. Un tel ajout pourrait diviser le temps de résolution par au moins 2 (par exemple, pour le niveau 0 de l'arbre de recherche, affecter à un thread une branche et à l'autre une seconde, puis laisser tourner chaque thread séparément).
- Améliorer l'heuristique : Puisque le branch'n'cut est plus rapide si une bonne heuristique est appliquée, on pourrait imaginer trouver une heuristique encore meilleure que celle que nous avons actuellement, pour couper encore plus rapidement les branches inutiles. On peut également mettre plusieurs heuristiques, basées sur des hypothèses différentes chacune, et ne garder que le meilleur résultat. On va directement vers des solutions avec du bon sens, le meilleur résultat peut alors être le plus important.
- Eliminer les branches qu'on sait inutiles : Pour le moment, l'élagage ne se fait qu'au moment de l'évaluation, ou avec le coût minimum, mais il y a certaines branches qu'on sait qu'il est inutile de visiter. Par exemple : on sait qu'envoyer le produit qui a la plus petite date due dans un lot tout seul, en dernier, est un choix qui est forcément pire qu'envoyer ce même produit dans un lot avec un autre produit, aussi en dernier. Malheureusement nous n'avons pas trouvé de généralisation à cet élagage, donc nous ne l'avons pas implémenté.

Conclusion

Ce projet fût très intéressant sur certains aspects. Nous avons dû, sans indications, trouver une méthode de résolution adéquate à un problème donné qui pourrait être appliqué dans la réalité (dans une compagnie de transport, par exemple). Nous avons passé de nombreuses heures à retourner le sujet sous tous les angles afin de trouver l'approche qui nous assurait une réponse exacte.

En plus de la phase de recherche d'algorithme, la phase d'optimisation a été très intéressante : nous nous sommes efforcés de diminuer, de part des petites optimisations, le temps de recherche des résultats exacts, ce qui nous a permis au final d'accéder au résultat optimal d'instances plus complexes, avec plus de produits. Nous regrettons cependant le fait que nous avons perdu énormément de temps à remettre en question nos méthodes, du fait de l'ambiguïté de l'énoncé.

Mais finalement, nous considérons le challenge comme réussi, puisque nous avons abouti à un programme totalement fonctionnel, capable d'analyser une instance du problème, puis d'en tirer une meilleure solution, en plus ou moins de temps, et ce avec une précision exacte.