Sommaire :

1. Problématique apportée par le sujet
   1. Contrainte de temps
   2. Contrainte de coûts
2. L’implémentation
   1. Structures de données
3. L’heuristique
   1. Construction des batches
   2. Test des permutations de batches
4. Résolution exacte
   1. Construction des batches
   2. Test de de toutes les possibilités avec cut
   3. Optimisation : tri de la liste initiale
5. Améliorations possibles

# Problématique amenée par le sujet

Pour trouver la solution optimale, nous avions en premier lieu pensé à ne construire une solution sous forme de liste de produits.

# 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 grande liberté d’implémentation. Après quelques recherches, la performance entre ces deux langages n’est pas forcément évidente. Nous avons un petit regret pour notre choix, puisque C++, de par son exigence, nous a parfois joué des tours.

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 ».

## 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.

### 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 inutile à la résolution.

# Résolution exacte

## 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.

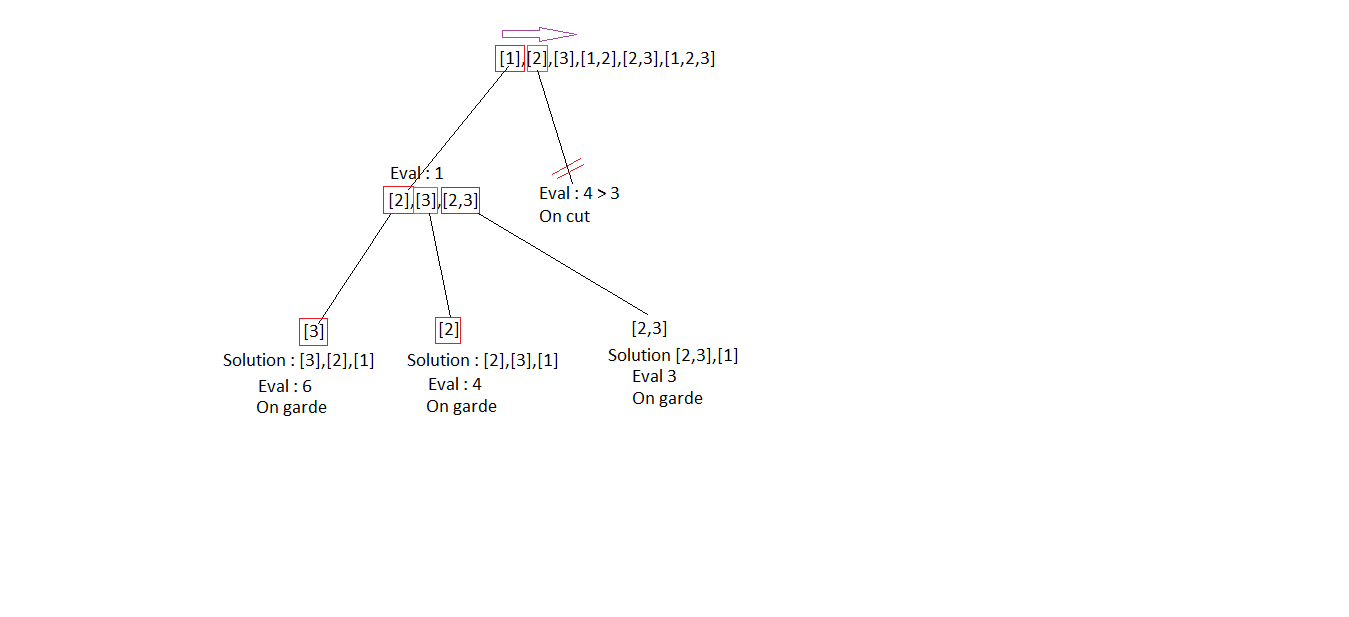
Exemple : Un client commande 4 produits, et le transporteur a une capacité de 3. Le nombre de batches possible avec répétitions et en gardant les batches inutiles est de 40. Sans répétitions, et en supprimant les batches inutiles, on descend ce nombre à 9.

## 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 de récursion 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 les produit et que l’évaluation est meilleure que l’évaluation de la meilleure solution, on l’enregistre.

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 Backtrack (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