UNIVERSITÉ DE TECHNOLOGIE DE BELFORT-MONTBÉLIARD


Paul Mollet-Padier

Printemps 2012

# But du projet :

Ce projet, dans le cadre de l'UV AG41 « Optimisation et Recherche Opérationnelle », a pour but la mise en place d'un algorithme permettant d'obtenir rapidement une solution « relativement » proche de la solution optimale à un problème complexe pour lequel il n'est pas forcément facile d'obtenir une solution exacte en un temps limité.

# Données du problème :

Ce problème est un problème de type « supply-chain management », avec un producteur en amont, un client en aval, et un transporteur liant les deux. Chaque problème est modélisé dans un fichier .txt ayant une structure bien définie afin de permettre son traitement par le programme.

Les différents coûts, détaillés dans le sujet, ainsi que les dates de livraison des différents produits, sont traités par la fonction `Solution::evaluate()` déjà implémentée. (On utilisera `Class::method()` pour noter une méthode d'instance et `Class.method()` pour noter une méthode statique).

Ainsi les données utilisées directement lors de l'algorithme sont le nombre total de produits, ainsi que la capacité du transporteur. Ce sont ensuite les vecteurs `productionSequenceMT` et `deliverySequenceMT` (`PSMT` et `DSMT` dans le reste du rapport) qui sont manipulés lors de la création de nouvelles solutions.

# Méthode implémentée :

Pour ce problème, j'ai choisi d'implémenter une recherche Tabou « agressive », avec une définition variable de la distance entre solutions voisines. Couplé avec une liste de solutions voisine de taille conséquente, ce choix me permet de couvrir rapidement un large éventail de solutions. J'ai donc trois paramètres en plus de la durée d'exécution :

`int sizeTL` : la longueur de la liste Tabou

Des trois paramètres, c'est celui ayant eu la moindre incidence sur le résultat final de l'algorithme. J'ai arbitrairement choisi comme paramètre par défaut 36 fois le nombre de produits du problème.

`int sizeNL` : le nombre de voisins calculés par itération Tabou

J'ai choisi de ne pas mettre de tests empêchant la génération de voisins doublons lors d'une même itération afin de ne pas alourdir l'algorithme : ce choix me permet de ne pas stocker tous les voisins à chaque itération, mais seulement le meilleur. De plus, en choisissant une variation maximale élevée (voir ci-dessous), le temps de calcul gaspillé sur des voisins doublons est négligeable par rapport au temps perdu s'il y avait eu des tests de condition à chaque calcul de voisin.

L'augmentation de ce paramètre ralentit l'algorithme, mais au final l'améliore encore plus qu'elle ne le ralentit. Par exemple, pour le problème `001-300,` j'obtenais des résultats de l'ordre de 320k avec `sizeNL = 10000`, alors qu'avec `sizeNL = 500` je tournais seulement entre 370 et 400k sur la même durée d'exécution.

En effet, `sizeNL` est directement lié à la boucle principale de la méthode `Taboo::bestNeighbor()`, appelée en début de chaque itération Tabou. Son augmentation réduit donc le nombre total d'itérations sur une durée fixe, mais permet d'avoir un meilleur voisin par itération. C'est donc là le choix à faire lors du paramétrage. Au final, j'ai opté pour une initialisation à 35 fois le nombre de produits du problème.

`int var` : la variation (ou distance) maximale entre une solution et ses voisines

Ce paramètre n'existait pas dans les premières versions de mon algorithme – initialement deux solutions étaient voisines si il y avait au maximum une permutation dans chacun des deux vecteurs `PSMT` et `DSMT`.

Sa mise en place me permit de traverser bien plus rapidement l'ensemble des solutions du problème, et par exemple de passer directement de 330-540k à 270-290k sur le problème 003-050. Lors de sa première version, chaque solution voisine était donc distante de `var` permutations (ou moins dans les cas ayant des permutations triviales). Ceci engendra un problème : dans certains cas, mon algorithme ne pouvait pas descendre dans un puits. En effet, après ma première version qui avançait de proche en proche, celle-ci sondait le périmètre d'un cercle, et ne voyait donc pas toutes les solutions du disque.

J'ai donc effectué une deuxième optimisation : la création d'un paramètre `bNvar` propre à chaque voisin généré par `bestNeighbor()`, généré aléatoirement entre `1` et `var`. Cette modification permet à mon algorithme de bombarder le disque de solutions plus

ou moins voisines, et donc d'avoir une vision bien moins restreinte. Avec cette seconde optimisation, je suis directement tombé sur la solution optimale pour le problème précédent, et ce en moins d'un cinquième de seconde.

Ce paramètre doit être strictement positif et strictement inférieur au nombre total de produits. L'augmentation de ce paramètre permet d'élargir le champ de recherche, tandis que la réduction permet de mieux chercher en profondeur sur un champ restreint. Il faut donc trouver un équilibre entre les deux extrêmes, d'où le choix final de prendre la moitié du nombre de jobs du problème.

```
int sizeTL = 36 * n;   // taboo length
int sizeNL = 35 * n;   // number of neighbors considered per taboo iteration
int var = n / 2;       // max variation between neighbor solutions
```

Finalement, ces trois paramètres peuvent être choisis en ligne de commande avec le fichier problème et le temps d'exécution, afin de peaufiner l'ensemble des paramètres pour un problème particulier.

# Structure du code :

L'ensemble du code source est contenu dans sept classes dans le package `challenge`, ce qui permet un accès direct aux attributs `protected` entre les différentes classes. Six d'entre elles existaient déjà avant de commencer le projet, et j'ai ajouté la classe `Taboo` contenant l'essentiel de l'algorithme.

Concernant les classes `FichierLecture`, `Problem`, et `Transporteur`, je n'ai apporté que quelques modifications triviales, essentiellement des suppressions de warnings. J'ai ajouté une poignée de méthodes sur les classes `Batch` et `Solution`, surtout pour résoudre le problème du clonage des `Vector<Batch>`, ainsi que pour la modification des vecteurs `PSMT` et `DSMT`. Finalement la classe `Main` a été modifiée pour permettre la bonne initialisation du problème, ainsi que l'ajout de la méthode statique nécessaire pour le bon clonage des vecteurs.

# Algorithme Tabou :

```
public Taboo(Problem pb, long length, int sizeTL, int sizeNL, int var) {
        start = System.currentTimeMillis();
        end = start;

        tabooList = new ArrayList<Solution>();
```

```
        tabooLength = sizeTL;
        solution = randomSolution(pb);
        bestSolution = solution.clone(pb);
        tabooList.add(solution.clone(pb));

        while (end - start < length) {// standard taboo algorithm, time-based
                end = System.currentTimeMillis();

        System.out.print("\r" + (int) (100 * (end-start) / length) + "%...");
                newSolution = bestNeighbor(pb, solution, sizeNL, var);

                if (newSolution.evaluation < bestSolution.evaluation) {
                        bestSolution = newSolution.clone(pb);
                        timeBestFound = end - start;
                }

                if (tabooList.size() == tabooLength)
                        tabooList.remove(0);

                tabooList.add(newSolution.clone(pb));
                solution = newSolution.clone(pb);
        }
}
```

Ce constructeur de la classe `Taboo` permet de voir un point de vue global de l'algorithme Tabou. On y retrouve les quatre paramètres passés lors de la création du `Taboo` dans `Main`, la génération aléatoire d'une solution initiale, et l'initialisation de la liste Tabou et de la `bestSolution`, avant la boucle principale du Tabou.

Dans celle-ci, ayant comme condition d'arrêt le temps d'exécution passé en paramètre, on cherche un `bestNeighbor`, que l'on compare à la meilleure solution jusqu'alors trouvée, ajoute à la liste Tabou, et qu'on met comme nouvelle solution principale pour la prochaine itération de la boucle.

## *Solution Taboo::RandomSolution(Problem) :*

Cette méthode complètement aléatoire ne fut pas toujours la méthode choisie pour générer une première solution dans mon algorithme. J'ai également testé une variante créant autant de lots qu'il y a de dates de livraison distinctes, et une autre créant un nombre minimal de lots vis-à-vis de la contrainte de capacité du transporteur. Ces méthodes n'apportèrent pas un avantage considérable à l'algorithme dans la pratique, et dans certains cas l'empiraient, donc je suis revenu à mon choix initial.

```
public Solution randomSolution(Problem pb) {

        // n number of total jobs, rj number of jobs randomly chosen for each
batch, rb randomly chosen index of each batch

        int n = pb.getNp(), rj, rb;
        Solution sol = new Solution(pb);
        Random rand = new Random();;

        // random selection of production batches

        while (n > 0) {
                rj = rand.nextInt(n) + 1;// between 1 and n jobs per batch
                rb = rand.nextInt(sol.getProductionSequenceMT().size()+1);
        // batch randomly inserted in vector to prevent front-stacking (due
      to probabilistic properties)
                sol.getProductionSequenceMT().add(rb,new Batch(rj));
                n -= rj;
        }

        n = pb.getNp();

        // random selection of transport batches, taking into account
      transporter capacity

        while (n > 0) {
                rj = rand.nextInt(Math.min(pb.transporter.getCapacity(),n))+1;
                rb = rand.nextInt(sol.getDeliverySequenceMT().size()+1);
                sol.getDeliverySequenceMT().add(rb,new Batch(rj));
                n -= rj;
        }

        sol.evaluate();

        return sol;
}
```

J'utilise deux boucles `while` pour peupler les vecteurs `PSMT` et `DSMP` en utilisant une distribution aléatoire des lots de taille aléatoire également. La distribution aléatoire n'était pas présente dans la première version de l'algorithme, et sert à éviter des vecteurs trop chargés dans les premiers coefficients. En effet, sans la variable `rb`, on aurait bien plus souvent des vecteurs du type `[45,1,1,1,1]` que du type `[1,1,1,1,45]`.

Cette nouvelle solution est ensuite évaluée et renvoyée à l'algorithme Tabou.

## *Solution Taboo::bestNeighbor(Problem,Solution,int,int) :*

Cette méthode de la classe `Taboo` est sans doute la plus importante de l'ensemble de l'algorithme. Elle calcule `sizeNL` voisins de la solution passée en paramètre, et renvoie la meilleure. La définition de voisinage est floue par choix : chaque voisin est voisin de la solution passée en paramètre à une distance aléatoirement choisie à partie de `var`, et donc à une distance souvent différente de celle choisie pour les autres voisins. Comme dit précédemment, ceci permet à l'algorithme d'avoir une large portée de recherche.

```
int bNvar;      // bNvar is the variation between the calling solution and
                each of the sizeNL neighbors generated
int dWvar;      // dWvar is a tmp variable for bNvar in the two do...while
                loops
```

Concernant ces deux entiers:
`bNvar` représente la variation maximale générée aléatoirement pour une solution voisine en particulier, tandis que `dWvar` est une variable temporaire basée sur `bNvar` pour les deux boucles `do...while` de l'algorithme.

```
bNvar = rand.nextInt(var)+1;  // bNvar between 1 and var

dWvar = bNvar; // dWvar initialized to bNvar before each do...while loop
do {
        randBatch = rand.nextInt(pS);
        if (tmp.getProductionSequenceMT().elementAt(randBatch).getQuantity()
                                                                <= dWvar ) {
                dWvar -= tmp.getProductionSequenceMT().elementAt(randBatch).
                                                                getQuantity();
                tmp.getProductionSequenceMT().remove(randBatch);
                --pS;
        } else {
                tmp.getProductionSequenceMT().elementAt(randBatch).
                                                        decQuantity(dWvar);
                dWvar = 0;
        }
} while (dWvar > 0);
```

Cette première boucle supprime `bNvar` lots du vecteur `PSMT`. On retrouve la même pour le vecteur `DSMT`. On choisit aléatoirement un lot, si ce lot n'est pas suffisamment rempli pour enlever tous les lots d'un coup, on le supprime et on recommence.

```
dWvar = bNvar;
do {
        randBatch = rand.nextInt(2*dS);
        if (randBatch >= dS) {
                randBatch = rand.nextInt(dS+1);
                tmp.getDeliverySequenceMT().add(randBatch, new Batch(Math.
                                                                min(dWvar, cap)));
                dWvar -= dWvar > cap ? cap : dWvar;
        } else if (tmp.getDeliverySequenceMT().elementAt(randBatch).
                                                getQuantity() >= cap - dWvar) {
                dWvar -= cap -tmp.getDeliverySequenceMT().
                                        elementAt(randBatch).getQuantity();
                tmp.getDeliverySequenceMT().elementAt(randBatch).
                                                        setQuantity(cap);
        } else {
                tmp.getDeliverySequenceMT().elementAt(randBatch).
                                                        incQuantity(dWvar);
                dWvar = 0;
        }
} while (dWvar > 0);
```

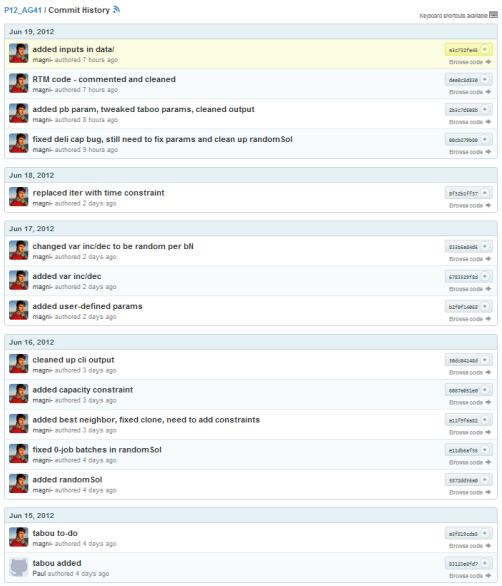
Cette boucle sert à rajouter les `bNvar` jobs précédemment supprimés dans les vecteurs ; le vecteur `DSMT` pour le code ci-dessus. Il existe donc une version similaire (sans la contrainte de la capacité du transporteur propre au cas de `DSMT`) pour `PSMT`.
On choisit d'abord un index de lot pour insérer ces jobs, en laissant une probabilité de création de nouveau lot. Dans ce cas là, on prend un nouvel indice aléatoirement afin de ne pas toujours ajouter les nouveaux lots en fin de vecteur (même principe que pour `randomSolution()`). Ensuite dans tous les cas, on ajoute le plus de jobs possibles au lot choisi en respectant la contrainte de capacité, et on répète jusqu'à ce qu'on ait bien rajouté l'ensemble des `bNvar` jobs.

Après chaque génération de nouveau voisin, on vérifie qu'il ne soit pas dans la liste Tabou. Si ce n'est pas le cas, on décroit le nombre de voisins restant à générer et on compare avec le meilleur voisin trouvé jusqu'à présent. On termine en renvoyant le meilleur voisin.

## Outils utilisés :

Bien que seul pour ce projet, j'ai utilisé `git` pour le version-control, ce qui a prouvé très utile lorsque je me suis trop enfoncé dans certaines modifications en évitant des `Ctrl+Z` peu efficaces. Cela m'a aussi permis de pouvoir re-visualiser l'évolution du projet avec les différents messages de commit jusqu'à la ligne de code près.
Le repository sur github est accessible publiquement depuis que le projet ait été fini : https://github.com/magni-/P12_AG41

L'historique des commit :




Ci-dessous, un exemple de modification. Ici, il s'agit de la correction du bug responsable pour la création de nouveaux lots dans DSMT excédant la capacité du transporteur, suite à une absence de test lors du cas où un nouveau lot est créé.

```
116     -       boolean repeat;
    124 +       dWvar = bNvar;
117 125         do {
118     -         repeat = false;
119 126           randBatch = rand.nextInt(2*dS);
120 127           if (randBatch >= dS) {
121 128             randBatch = rand.nextInt(dS+1);
122     -           tmp.getDeliverySequenceMT().add(randBatch, new Batch(0));
123     -         } else if (tmp.getDeliverySequenceMT().elementAt(randBatch).getQuantity() == cap)
124     -           repeat = true;
125     -       } while (repeat);
126     -       tmp.getDeliverySequenceMT().elementAt(randBatch).incQuantity(bNvar);
    129 +           tmp.getDeliverySequenceMT().add(randBatch, new Batch(Math.min(dWvar, cap)));
    130 +           dWvar -= dWvar > cap ? cap : dWvar;
    131 +         } else if (tmp.getDeliverySequenceMT().elementAt(randBatch).getQuantity() >= cap - dWvar) {
    132 +           dWvar -= cap - tmp.getDeliverySequenceMT().elementAt(randBatch).getQuantity();
    133 +           tmp.getDeliverySequenceMT().elementAt(randBatch).setQuantity(cap);
    134 +         } else {
    135 +           tmp.getDeliverySequenceMT().elementAt(randBatch).incQuantity(dWvar);
    136 +           dWvar = 0;
    137 +         }
    138 +       } while (dWvar > 0);
```

# Conclusion :

Ce projet m'aura permis de mettre en œuvre les principes appris en cours. Sachant que la topologie des coûts pouvait varier drastiquement d'un problème à un autre, j'ai essayé de garder un algorithme très agressif sans pour autant faire d'algorithme glouton, impraticable sur les grandes instances avec des contraintes de temps.
En comparant les résultats obtenus avec les résultats divulgués publiquement, j'estime que c'était un bon choix, vu que pour chacune des instances, je battais ou égalisais le meilleur score de manière très régulière.
Il y avait bien entendu d'autres méthodes envisageables ; j'ai par exemple envisageais un temps la mise en place de Threads, ou alors de combiner mon algorithme avec un algorithme génétique. Cependant, vu les résultats que j'obtenais, j'ai préféré peaufiner les paramètres de mon algorithme pour un problème général plutôt qu'effectuer une vaste modification du code.