# Algorithmique et Optimisation de Recherche Opérationnelle

Victor Haren et Benjamin March

20 juin 2012

# Table des matières

# 1 Introduction

## 1.1 Contexte

Souvent, il arrive qu'on ait besoin de l'informatique pour résoudre des problèmes. Ici, c'est un de ces cas là : un fabricant doit déterminer la façon de produire et livrer au client les produits commandés la plus efficace. La dimension du problème fait que la résolution à la main est impossible. Il nous incombe donc de développer une façon de traiter ce problème, et d'en sortir la, sinon une, bonne solution pour le fabricant. La structure du problème nous appelle immédiatement à l'utilisation de divers algorithmes et de recherches opérationnelles, c'est donc dans cette voie que le sujet se développe.

## 1.2 Informations fournies

Plusieurs contraintes nous ont été imposées. Celles-ci peuvent être retrouvés sous leurs formulation mathématique dans le sujet du challenge, donc on ne les ré-écrira pas ici. Ce qui nous a aussi été donné est une structure du problème en Java, dans lequel notre projet devait s'inscrire, et un script de lancement. Nous avons aussi reçu plusieurs instances de problème à résoudre, de tailles variés.

## 1.3 Approche

Avec ceci, nous avons pu prendre une première approche du problème. L'aspect concret des classes Java a rendu la compréhension beaucoup plus simple que la simple formulation abstraite des conditions mathématiques, et nous avons pu assez rapidement commencer à explorer les possibilités de résolution.

## 2 Premières Idées

A l'abordage du problème nous avons tout de suite remarqué la fonction `setFromString(String s)` qui prend en paramètre un String de chiffres et crée la solution correspondante. Il nous est tout de suite venu à l'idée d'utiliser cette fonction en conjonction à une génération de nombres aléatoires pour générer des solutions aléatoires. Ceci a donné la fonction `randomize()`.

### 2.1 Aléatoire pur

```
public void randomize() {
    int i = slpb.getNp();
    int j;

    while (i != 0) {
        Random r = new Random();
        j = r.nextInt(i) + 1;
        i -= j;

        addProductionLast(j);
    }

    i = slpb.getNp();
    while (i != 0) {
        Random r = new Random();
        j = r.nextInt(i) + 1;
        i -= j;

        if(j<=slpb.getTransporter().capacity) {
            i -= j;
            addDeliveryLast(j);
        }
    }
    //System.out.println(productionSequenceMT + "|" + ↩
        deliverySequenceMT + " => " + evaluate());
}
```

Cette fonction a pour but de générer simplement une solution aléatoire. On peut l'évaluer ensuite avec la fonction `evaluate()` qui était donné dans le code original. Le seul test fait ici est de faire en sorte que les batchs de livraison de soient pas plus grands que la capacité du transporteur.

### 2.2 Aléatoire "contrôlé"

Par la suite, au fur et à mesure des tests de cette fonction, nous avons pris conscience de la puissance de la générations aléatoire. En effet, plusieurs itérations du problème avec cette seule fonction de génération aléatoire nous donnaient quelques solutions pas trop mauvaises pour de petites instances.

Nous nous sommes alors dit qu'il serait intéressant d'implémenter un algorithme évolutionniste, dont un des grands composants est l'Aléatoire. Mais avant cela, nous avons élaborer une seconde fonction de génération, qui génère des solutions dont le nombre de batchs est plus élevé, et donc la quantité par batch plus petite.

```
public void randomize2() {
    int nb_products = slpb.getNp();
    int i, current_products, max;
    Random r = new Random();

    current_products = 0;
    max = nb_products / 4;
    while (current_products < nb_products) {
        i = r.nextInt(max + r.nextInt(max)) + 1;

        if (current_products + i > nb_products) {
            i = nb_products - current_products;
        }

        addProductionLast(i);
        current_products += i;
    }

    current_products = 0;
    max = nb_products / 4;
    while (current_products < nb_products) {
        i = r.nextInt(max + r.nextInt(max)) + 1;

        if (current_products + i > nb_products) {
            i = nb_products - current_products;
        }

        if(i<=slpb.getTransporter().capacity) {
            addDeliveryLast(i);
            current_products += i;
        }
    }

    //System.out.println(productionSequenceMT + "|" + ↩
        deliverySequenceMT + " => " + evaluate());
}
```

Par la combinaison de ces deux générations on obtient une assez grande variété de solutions. C'est donc ainsi que nous avons décidé de construire la population de l'algorithme évolutionniste.

# 3 Algorithme Évolutionniste

Nous avons décidé d'implémenter l'algorithme évolutionniste assez rapidement. En fait, c'est en prenant quelques idées dans le TP sur ces algorithmes que nous avons pris conscience de la puissance de ces algorithmes, qui, même s'il ne permettent pas toujours de trouver la solution optimale dans des problèmes à beaucoup de solutions, ils permettent d'en trouver un assez bon. Pour ce faire, nous avons donc écrit la classe `AlgorithmeEvolutionniste` et la classe `Population`. Lors de la construction de l'algorithme, celui-ci génère une population de solutions via les fonctions `randomize`. Ceux-ci sont ensuite croisés et mutés pour faire varier la population, mais ne sont gardés que si leur évaluation est meilleure que celle de leurs parent. Ceci donne un côté élitiste à l'algorithme.

## 3.1 Croisement

La fonction croisement est une fonction très simple. Cette fonction prend en paramètre deux solutions et échange leurs séquences de batchs de production, ce qui nous revoie deux solutions fils. Si ces solutions sont meilleurs que leurs parents, ils les remplacent.

```
public void crossbreed(Solution father, Solution mother) {
    Solution newChild1 = new Solution(mother);
    Solution newChild2 = new Solution(father);

    //production of father in newChild1
    newChild1.setProductionSequenceMT((Vector) father.↩
        getProductionSequenceMT().clone());

    //production of mother in newChild2
    newChild2.setProductionSequenceMT((Vector) mother.↩
        getProductionSequenceMT().clone());

    //if the're better solutions than their parents, replace
    if (newChild1.evaluate() < pop.getBest().evaluation) {
        pop.setBest(newChild1);
        father = newChild1;
    }

    if (newChild2.evaluate() < pop.getBest().evaluation) {
        pop.setBest(newChild2);
        mother = newChild2;
    }
}
```

## 3.2 Mutation

La fonction mutation repose sur l'aléatoire pour inverser une sous-séquence de batchs à la fois dans la séquence de production et dans la séquence de

livraison. Nous avons trouvé que la solution générée était assez différente de la solution initiale pour considérer une ré-injection dans la population active si son évaluation était meilleure que l'originale, sinon, on ne le garde pas.

```
public void reverseRandomBatchSequence(Vector batches) {
    Random r = new Random();
    int i = r.nextInt(batches.size());
    int j = r.nextInt(batches.size());

    while (j<i)
        j = r.nextInt(batches.size());

    int k = j-i;
    int iter = 0;
    while (k > 0) {
        swapTwoBatches(i+iter, j-iter, batches);
        iter++;
        k -= 2;
    }
}
```

```
if (r.nextInt(100) < mutationLevel*100) {
    Solution father = pop.getIndividuals().get(r.nextInt(popSize));
    father.reverseRandomBatchSequence(father.productionSequenceMT);
    father.reverseRandomBatchSequence(father.deliverySequenceMT);

    if (father.evaluate() < pop.getBest().evaluate())
        pop.setBest(father);

}
```

Pour les deux fonctions de croisement et de mutation, nous avons un paramètre de l'algorithme qui détermine la probabilité d'effectuer ces changements. Comme vu en TP, trop de mutation ou de croisement est inefficace, et au fur et a mesure des test nous avons trouvé les paramètre idéaux aux alentours de 0.8 pour le croisement et 0.4 pour la mutation.

### 3.3 Ouverture

Au fur et à mesure de tests sur l'algorithme développé, nous nous sommes rendus compte d'une chose, c'est que, si l'algorithme évolutionniste permet de trouver une bonne solution, il ne permet pas toujours de trouver la meilleure, étant donné le caractère purement aléatoire de la chose. Nous avons donc pensé qu'il serait avantageux de perfectionner en quelque sorte la solution trouvée.

# 4 Tâtonnement

Pour perfectionner la solution trouvée par l'algorithme évolutionniste, nous avons naturellement pensé à l'utilisation d'un algorithme par tâtonnement, ou "Hill-climbing" en anglais. Celle-ci est basée sur une fonction de voisinage très basique, et l'évaluation de ces voisins pour en garder le meilleur. Ensuite, l'étape est répété jusqu'à ce qu'il n'y ait aucune progression. C'est donc que nous avons atteint l'optimum local. Cependant, il n'y a aucune garantie que cela soit l'optimum global, du fait de la nature aléatoire de la solution initiale.
Nous avons donc créé la classe `AlgorithmeTatonnement`.

## 4.1 La fonction voisinage

```
public void getBestxxxNeighbour(Solution sol, Solution temp, Random ↩
    r) {
    int size = sol.xxxSequenceMT.size();

    for(int i = 0; i < size; ++i) {
        //Copy solution into temp
        temp.productionSequenceMT = (Vector<Batch>)sol.↩
            productionSequenceMT.clone();
        temp.deliverySequenceMT = (Vector<Batch>)sol.↩
            deliverySequenceMT.clone();

        //Subtract 1 from the i-th batch
        int k = sol.getxxxBatchSize(i);
        temp.setxxxBatchSize(i, k-1);

        //Add it to another batch
        k = r.nextInt(size);
        while(k == i)
            k = r.nextInt(size);
        int qte = sol.getxxxBatchSize(k);
        temp.setxxxBatchSize(k, qte + 1);

        temp.evaluate();

        //Afficher
        //System.out.println("Resulting solution : " + temp.↩
            productionSequenceMT + " | " +          temp.↩
            deliverySequenceMT + " => " + temp.evaluate());
        //System.out.println();
        //Test the resulting solution
        if (temp.evaluation < bestNeighbour.evaluation) {
            bestNeighbour.productionSequenceMT = (Vector<Batch>)temp↩
                .productionSequenceMT.clone();
            bestNeighbour.deliverySequenceMT = (Vector<Batch>)temp.↩
                deliverySequenceMT.clone();
            bestNeighbour.evaluate();
        }
    }
}
```

Évidemment, la partie sur la livraison prend en compte la capacité du transporteur via les test suivant.

```
while(temp.getDeliveryBatchSize(k) == temp.slpb.getTransporter().↩
    capacity) {
    k=r.nextInt(size);
}
int qte = temp.getDeliveryBatchSize(k);
temp.setDeliveryBatchSize(k, qte + 1);
```

## 4.2 Son implémentation

Cette fonction de recherche de meilleur voisin peut, et est, appliquée sur les deux séquences de batchs, à la fois celle de production et celle de livraison. Cela nous donne deux fois plus de chances de trouver une meilleure solution.

```
public Solution getBestNeighbour(Solution sol) {
    Solution temp = new Solution(pb);
    Random r = new Random();
    bestNeighbour.evaluate();

    //Verifier tous les voisins :

    //Cote production
    getBestProductionNeighbour(sol, temp, r);
    //Cote delivery
    getBestDeliveryNeighbour(sol, temp, r);

    return bestNeighbour;
}
```

Après l'implémentation de cette partie de programme nous avons pu constater une vraie différence d'efficacité dans nos recherches de solutions. En effet, là où, avant, nous n'avions que peut-être une solution sur quatre qui était particulièrement bonne, nous avons remarqué que désormais l'écart-type entre les meilleures solutions trouvées était beaucoup plus petite, et la valeur moyenne était plus optimale. C'est donc une étape cruciale de notre programme que l'amélioration par tâtonnement.

# 5 Le Programme Principal

Bien évidemment, une fois les algorithmes développés, il fallait les inclure dans le programme principal. Nous avons pensé augmenter les chances de trouver une bonne solution en utilisant un paramètre de stérilité, c'est à dire un nombre de fois d'application de l'algorithme au problème au bout duquel, si la meilleure solution n'a pas varié, on passe à l'étape de perfectionnement.
Dans le programme principal, nous faisons donc appel aux algorithmes en les instanciant avec des paramètres améliorés au fil des exécutions et de la phase de test du programme. En effet, nous avons rapidement constaté qu'il y a des nombres limite de population et d'itérations au bout duquel l'algorithme évolutionniste est moins efficace. Pour cela, nous avons trouvé plus utile de garder le nombre d'itérations et de population relativement bas, et nous intéresser plutôt au nombre de fois que l'algorithme est appelé. Ceci rend bien sûr la durée d'exécution légèrement différente à chaque exécution du programme, cependant l'ordre de grandeur étant seulement de 2 ou 3 secondes nous n'en prenons que peu compte.
Dans un second temps, pour répondre à la contrainte sur le temps d'exécution, nous avons aussi mis en place un système pour interrompre la recherche si le temps est dépassé. Cela se fait au moyen d'un paramètre a renseigner dans le programme.

## 5.1 La partie évolutionniste

### 5.1.1 Sans contrainte de temps

```
while(unchanged < 40) {
    algo = new AlgorithmeEvolutionnaire(25000, 150, (float)0.8, (↩
        float)0.5, pb);
    sol = algo.run();

    if (sol.evaluation < best.evaluation)
        best = sol;
    else
        unchanged++;
}

//System.out.println("Meilleure solution apres genetique : " + best.↩
    evaluation);
//System.out.println(best.productionSequenceMT + "|" + best.↩
    deliverySequenceMT);
```

### 5.1.2 Avec contrainte de temps

```
if (exectime > 0) {
    double starttime = System.currentTimeMillis();
    while ((System.currentTimeMillis() - starttime) < exectime * ↩
        1000) {
        algo = new AlgorithmeEvolutionnaire(generations_nbr, ↩
            population_size, crossbreed_rate, mutation_rate, pb);
        sol = algo.run();

        if (sol.evaluation < best.evaluation)
            best = sol;
    }
}
```

## 5.2 La partie tâtonnement

```
AlgorithmeTatonnement hc = new AlgorithmeTatonnement(pb, best);
Solution optimum = new Solution(pb);
optimum.copy(best);

boolean foundbetter = true;
int i = 0;
int limit = 50;

while(foundbetter && i < limit) {
    best.copy(optimum);
    System.out.println("Optimizing");
    optimum.copy(hc.getBestNeighbour(best));
    optimum.evaluate();

    if(optimum.evaluation < best.evaluation) {
        System.out.println("Found better ! " + optimum.evaluation);
    }
    else {
        foundbetter = false;
    }

    i++;
}
```

Une autre solution que nous avons trouvé pour améliorer la vitesse d'exécution de notre algorithme était de garder un minimum de `System.out.print`. Nous avons constaté que cette fonction prend beaucoup de temps pour peu d'intérêt final, alors nous nous sommes contenté du minimum nécessaire à la bonne compréhension de l'utilisateur.

# 6 Conclusion

Voilà en somme notre approche du challenge qui nous a été donné. Ce qui est intéressant à remarquer avec notre solution, c'est que nous obtenons de très bons résultats en très peu de temps pour les problèmes de tailles petites et intermédiaires, mais que, pour les problèmes à plus grande dimension, nous trouvons rarement de bons résultats, et l'augmentation du temps de recherche n'y change que peu de chose.

Une chose intéressante à faire serait peut-être de revoir la fonction de voisinage pour en définir un plus adapté à l'échelle des grandes instances. Là, le tâtonnement pourrait peut-être plus efficace et chercher plus loin autour de la meilleure solution trouvée par l'algorithme évolutionniste.

Quoiqu'il en soit, vous pouvez trouver quelques résultats ci-joints.