Projet d'Optimisation Stochastique

Implémentation de l'algorithme du Recuit Simulé

Problème du Voyageur de Commerce
Document technique

Auteur :
Jeremie Bosom
Erwan Chaussy

Professeur :
M.Abdel Lisser

19 janvier 2014

1 Présentation du problème

Le but de ce projet est de traiter le problème du voyager de commerce, Travelling Salesman Problem en anglais. Il s'agit d'un problème d'optimisation combinatoire qui a été formulé de manière générale en 1930, à l'aide d'un modèle mathématique clair.

Considérant un certain nombre de villes, le voyageur de commerce doit parcourir toutes les villes une et une seule fois. On cherche le chemin le plus court qui part d'une ville, les parcours toutes et rejoins la ville de départ.

Cela semble simple et ça l'est, si l'on ne considère que des petites instances, c'est à dire une dizaine de villes. En effet, si l'on a n villes, il est possible de rejoindre $n-1$ et ainsi de suite. On trouve donc un premier chemin en un temps proportionel à $O(n)$. Mais pour être sur qu'il s'agit du plus court chemin, il faut examiner les autres chemins possible. Si l'on cherche une solution exacte, on remarque que la résolution du problème se fait en un temps $O(n^n)$ C'est ce que l'on considère être un problème NP-difficile.

Il semble évident que l'on ne peut pas énumérer toutes les solutions possible lorsque l'on traite 5000 villes par exemple. Il faut donc imaginer des méthodes approchées qui permettent de trouver une solution plus proche de la solution optimale. Dans ce projet, nous allons utiliser l'algorithme stochastique du recuit simulé que nous présenterons après le modèle mathématique du problème du voyageur de commerce.

2 Modèle mathématique

Nous allons utiliser le jeu de donné suivant :

- $G = (V, E)$un graphe orienté complet de n sommets
- c_{ij}le coût de l'arc (v_i, v_j)

$$x_{ij} = \begin{cases} 1 & \text{si et seulement si l'arc (i, j) est retenu dans le circuit,} \\ 0 & \text{sinon} \end{cases}$$

Nous cherchons donc à minimiser le coût total du trajet. Cela se traduit par le programme linéaire suivant :

$\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$
$s.t.$
$\sum_{j=1}^{n} x_{ij} = 1, i = 1, \ldots, n$
$\sum_{i=1}^{n} x_{ij} = 1, j = 1, \ldots, n$
$\sum_{i|v_i \in S} \sum_{j|v_j \in S} x_{ij} \leq |S| - 1$, $S \subset \{v_1, \ldots, v_n\}$ et $S \neq \emptyset$
$x_{ij} \in \{0, 1\}, 1 \leq i \leq n$

La première contrainte décrit le fait qu'il n'y a qu'une et une seule arrête partant d'une ville. La seconde contrainte décrit le fait qu'il n'y a qu'une et une seule arrête arrivant dans une ville. La dernière contrainte est la contrainte dite de sous-tour. Elle empêche le fait que l'on passe plusieurs fois par une même ville. C'est cette contrainte qui rend le problème du voyageur de commerce NP-difficile. En effet, cette contrainte vérifie toutes les villes du graphe entres-elles, rendant le calcul extrêmement compliqué, réalisé en un temps en $O(n!)$. Il devient évident qu'une recherche exhaustive ne peut être envisagée.

3 Recuit simulé

L'algorithme que nous allons implémenter est l'algorithme du recuit simulé. Il s'agit d'une méta-heuristique inspiré du processus utilisé en métallurgie. Une phase de refroidissement lent du matériau est alterne avec une phase de réchauffage, aussi appelé recuit. Cet alternance a pour effet de minimiser l'énergie total du matériaux. Il a été constaté que le refroidissement naturel de certains métaux ne permet pas aux atomes de se placer dans la configuration la plus solide. Il faut donc contrôler le refroidissement à l'aide d'une forte chaleur afin de trouver la meilleure configuration possible. Cette méthode a alors été appliquée en optimisation afin de trouver les extrema d'une fonction.

Le principe est d'explorer le voisinage d'une solution, ce qui sera présenté ultérieurement, qui sera accepté ou non. Il est possible d'accepter des solutions qui augmentent le coût total de la solution, afin de ne pas être bloqué dans un minimum local.
Une fois ce voisin généré, nous calculons la différence Δ de coût entre cette solution et la solution précédente. Si le coût de celle-ci est inférieur ou égale, nous gardons cette solution. Sinon, nous acceptons la solution avec une probabilité dépendant de la température T : $\exp^{\Delta/T}$.
La température est un paramètre important de l'algorithme car c'est celle-ci qui détermine la probabilité avec laquelle on accepte une solution qui dégrade le résultat.

Cette exploration de voisinage est exécutée un certain nombre de fois, ce que l'on appelle un palier. Une fois le palier terminé, nous diminuons la température et nous recommençons un palier. Dans notre implémentation, la température suit une décroissance géographique d'un coefficient de $0,9$. La condition d'arrêt de notre recuit simulé tiens compte du taux d'acceptation des solutions. En effet, si ce taux n'est pas atteint un certain nombre de fois, l'algorithme s'arrête. Cela s'explique par le fait que l'on ne trouve plus de solution qui améliore réellement le résultat et que l'on stagne. Il faut alors s'arrêter et renvoyer la meilleurs solutions trouvée.

Ci-dessous se trouve l'algorithme tel que nous l'avons implémenté. La solution de base est obtenue à l'aide de l'algorithme glouton du Plus Proche Voisin que nous ne détaillerons pas. La température T_0 est obtenue par réglage automatique, ce qui est détaillé plus tard dans le rapport. Le taux d'acceptation minimal a été fixé à $0,2$. Le nombre d'itération par palier est de l'ordre de n^2, n étant le nombre de ville par instance, si l'instance possède moins de 300 villes, $n * 2$ sinon.

Algorithme 1: Recuit Simulé

Données: E le premier cycle généré, T_0 la température du recuit, P le nombre d'itération par palier, Tx le taux d'acceptation minimale, *Coeff* le coefficient de décroissance de la température

Résultat: meilleureSol la meilleure solution retenue

début

```
// Initialisation
T ← T_0;
S ← E;
meilleureSol ← S;
compteur ← 0;
// Recuit en lui même
répéter
    nb_mouvements ← 0;
    // Pallier à température fixe
    pour i ← 1 à P faire
        S' ← engendrerVoisinDe(S);
        Δ ← distanceTotale(S') − distanceTotale(S);
        si Δ ≤ 0 ‖ alea() ≤ exp^(−Δ/T) alors
            S ← S';
            nb_mouvements ← nb_mouvements + 1;
            si distanceTotale(S) < distanceTotale(meilleureSol) alors
                meilleureSol ← S;
                compteur ← 0;
        fin
    fin
    // Actualisation des paramètres
    tauxAcceptation ← i/nb_mouvements;
    si tauxAcceptation < Tx alors compteur ← compteur + 1;
    T ← T * Coeff;
jusqu'à compteur < 100;
retourner meilleureSol;
```

fin

Il est possible de passer en paramètre de l'application le taux Tx d'acceptation minimal, le coefficient *Coeff* de décroissance de la température et le nombre P d'itération par pallier. Si aucun de ces arguments n'est transmis, la valeur de base, décrite plus haut, leur est attribuée. Afin d'éviter que notre algorithme ne tourne trop longtemps, nous avons ajouté une limite de temps qui est de 15 minutes, à moins de préciser une autre valeur par passage d'arguments à l'application. Le dernier paramètre réglable est l'algorithme utilisé afin de déterminer la solution de base. Toutes ces informations sont décrites dans le document utilisateur.

4 Voisinages considérés

Nous avons condisérés plusieurs manières de trouver une solution voisine à celle actuelle. Néanmoins, peu d'entre elles se sont révelées concluante. Nous avons donc choisi d'utiliser, après de nombreux tests, un algorithme proche du 2-Opt.

4.1 Aleatoire

Le premier algorithme que nous avons implémenté est un algorithme inversant deux villes toutes deux choisient aléatoirement. Mais les résultats n'étaient pas très concluants.

4.2 Inversion de deux villes voisines

Dans un deuxième temps, nous avons essayé d'inverser deux villes voisines en choisissant aléatoirement la ville à inverser. Mais ici aussi notre algorthme n'étaient pas très concluant.

4.3 Déplacement d'un groupe de ville

Nous avons suppposé que notre algorithme n'était pas efficace car il inter-changeait un trop petit nombre de ville. ALors nous avons implémenté un algorithme qui déplace un nombre de ville donné en paramètre en fin de boucle. Mais cet algorithme ne nous as pas apporté les résultats espérés.

4.4 2-Opt

Nous en sommes donc venu à implémenter un algorithme proche de celui du 2-opt pour trouver une solution voisine. Cela permet de voir émerger un grand nombre de solutions corrrectes et donc d'améliorer significativement notre solution. Le principe est d'essayer d'échanger deux extrimités d'arrêtes puis d'inverser le sous-tour formé entre ces deux extrémités. C'est cet algorithme que nous utilisons pour trouver une solution voisine.

4.5 2-Opt simplifié

Nous avons essayé d'implémenter un algorithme proche du 2-opt mais un peu plus simplifié qui se contenterait d'inverser extrémités d'arrêtes et le sous-tour associé en prenant ces extrémités au hasard. Mais cela n'a pas donné les résultats attendus.

5 Réglage de la température du recuit simulé

Le réglage de la température utilisée pour l'algorithme est quelque de compliqué puisque cela influe sur le résultat final. En effet, une température trop haute acceptera trop de changement, rendant le début de l'algorithme inutile. Au contraire, une température trop basse n'acceptera que les changements qui réduisent vraiment le coût total, au risque de bloquer dans un minimum local. Il faut donc trouver la bonne température qui produira un taux d'acceptation suffisamment grand pour ne pas rendre le début de l'algorithme inutile.

En général, ce taux est fixé à au moins 80% d'acceptation. Nous exécutons donc l'algorithme du recuit simulé avec une température assez basse, calculé en fonction du nombre de villes dans l'instance, à savoir $arrondie\big((nbVilles/100)+1\big)$. Au lieu de diminuer la température à la fin des itérations, nous vérifions si le taux est atteint ou non. S'il ne l'est pas, nous doublons la température et recommençons. Une fois le taux atteint, nous retournons la dernière température et lançons l'algorithme du recuit simulé.