INP ENSEEIHT

Graphes et Recherche opérationnelle

Rapport final

Chef de projet :
Martin Carton
Responsable qualité :
Maxime Arthaud

Maxence Ahlouche
Korantin Auguste
Thomas Forgione
Thomas Wagner

20 décembre 2013

Table des matières

1 Introduction

Rappel : L'objectif de ce rapport final (CRF6) est de ***corriger les erreurs*** commises dans les rapports (CRFs) précédents (ndlr : il y en a !) et de faire la preuve que les travaux de l'équipe l'ont conduite à ***acquérir des connaissances théoriques*** dans les divers domaines abordés.

2 UA : Graphes

2.1 Modélisation mathématique

Nous avons choisi de représenter nos graphes comme une liste de sommets, chacun ayant une liste d'arêtes.

En mémoire, cette structure est donc constituée d'une liste de pointeurs vers des sommets. Les sommets contenant une liste de pointeurs vers des arêtes. Chaque arête ayant un pointeur vers chaque sommet extrémité.

Dans la suite nous noterons n le nombre de sommets du graphe.

2.2 Problème du voyageur de commerce

2.2.1 Analyse mathématique

Le problème du voyageur de commerce consiste à chercher un chemin passant par tous les sommets du graphe, de longueur minimale. Ce problème peut s'illustrer par l'exemple d'une fraiseuse qui doit percer des trous dans une plaque le plus rapidement possible, ou encore par un car de touristes qui souhaiterait trouver l'itinéraire le plus rapide pour visiter un certain nombre de lieux.

On peut modéliser ce problème par un graphe complet, dont les arêtes ont un coût qui correspond à la distance entre chaque point, on cherche alors le cycle hamiltonien de coût minimal. On sait qu'un tel cycle existe car le graphe est complet.

Cependant trouver un tel cycle est un problème NP-difficile, et il n'existe donc pas d'algorithme efficace pour trouver ce cycle. En effet, la seule méthode exacte consisterait à tester toutes les chaines hamiltoniennes, et à prendre celle la plus courte, mais le nombre de chaines hamiltonniennes croît exponentiellement en fonction du nombre de sommets dans le graphe.

Nous allons donc nous concentrer sur les méthodes approchées de résolution, qui peuvent donner de très bons résultats tout en étant rapides. Toutefois, le résultat n'est donc pas forcément le plus court.

2.2.2 Heuristiques

Les heuristiques vont nous permettre de construire un chemin court (par rapport au plus court possible), de manière rapide, avec le moins de calcul possible. Étant donné qu'on est confronté à énormément de possibilités pendant la recherche, ils vont permettre d'orienter cette dernière, en faisant des choix les plus judicieux possibles sur les possibilités à explorer.

Exemple Une heuristique simple consiste à partir d'un sommet au hasard du graphe et d'aller au sommet le plus proche sur lequel on n'est pas encore passé (puis à retourner au sommet de départ pour boucler le cycle). Cet algorithme est en $O(n)$ et donc rapide. Mais il n'offre cependant aucune garantie de résultat, il existe même des graphes pour lesquels il donne le pire cycle.

Plus généralement, chercher parmi les p sommets les plus proches s'avère être une solution relativement efficace, avec une complexité en $O(p^n)$ (donc toujours exponentielle si $p \neq 1$).

Une méthode purement basée sur cet heuristique donnerait :

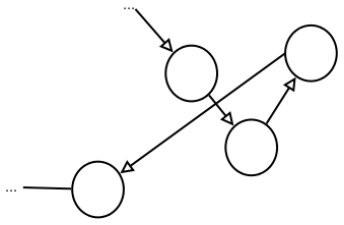
```
Entrée : g (Graphe complet)
Sortie : (coût, cycle) où cycle est un cycle hamiltonien construit selon la
    méthode du plus proche voisin et coût son coût associé sous forme de liste de
    points
coût = 0
cycle = ["un point de g au hasard"]

tant qu'il reste des points:
    # On ajoute au cycle le point suivant
    plus_proche = "point de g sur lequel on est pas encore passé le plus proche
        du dernier point du cycle"

    coût += "coût de plus_proche au dernier point du cycle"
    cycle = cycle :: plus_proche

# On ferme le cycle
coût += "coût du dernier au premier point de cycle"
cycle = cycle :: "premier point de chaine"

retourner (coût, cycle)
```

2.2.3 Recherche locale

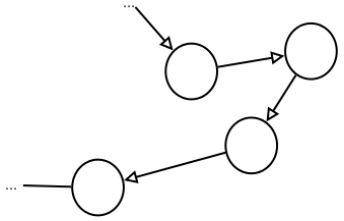
Les heuristiques nous donnant des solutions acceptables, choisies avec un minimum de « bon sens », il est ensuite possible de tenter d'améliorer ces solutions, via de la recherche locale. Partant d'une solution fournie, on va explorer les solutions voisines à cette dernière, afin de voire si on pourrait pas trouver des solutions encore meilleures parmi leur voisinage.

Exemple Un algorithme de recherche locale adapté au problème du voyageur de commerce est le 2-opt. Le principe du 2-opt consiste à tenter d'éliminer les « boucles » qui pourraient survenir dans le chemin, afin de le rendre plus court.

Ainsi, partant du chemin suivant (qu'on obtiendrait logiquement en suivant l'heuristique consistant à aller sur le sommet le plus près) :



On obtiendrait ceci, en éliminant le croisement :



L'algorithme pour le 2-opt est le suivant :

```
Entrée : un cycle hamiltonien (liste de sommets) et son coût
Sortie : un cycle hamiltonien et son coût inférieur ou égal au coup d'entrée

pour chaque couple de points (a, b) dans le cycle:
    nouveau_coût = coût
                    - "coût de a à son successeur dans le cycle"
                    - "coût de b à son successeur dans le cycle"
                    + "coût de a à b"
                    + "coût du successeur de a et au successeur de b dans le
                        cycle"

    si nouveau_coût < coût:
        coût = nouveau_coût
        cycle = cycle crée en échangeant a et b dans cycle

retourner (coût, cycle)
```

Il est donc en $O(n^2)$.

L'application du 2-opt sur le chemin obtenu via un heuristique simple peut donner des résultats plus proches de la solution optimale qu'on pourrait le penser, et la combinaison des deux est donc une bonne méthode (voir la section 2.2.5).

2.2.4 Métaheuristiques

Plutôt que d'utiliser un simple heuristique pour trouver une solution à priori plutôt bonne, puis d'y appliquer une recherche locale pour tenter de l'améliorer encore, il est possible d'utiliser des « métaheuristiques ». Ces algorithmes vont avoir besoin d'heuristiques et de recherche locale, mais vont s'en servir en boucle, pour tenter sans cesse de trouver une solution meilleure.

Ils vont partir explorer différentes parties de l'espace, souvent en guidant leur exploration grâce à l'heuristique, et en essayant de retomber sur des parties de l'espace les plus intéressantes possibles grâce aux algorithmes de recherche locale.

Il existe énormément de métaheuristiques. En voici quelques uns :

Recherche locale itérée métaheuristique très simple consistant à utiliser un heuristique puis appliquer de la recherche locale pour améliorer son résultat. Ensuite, on perturbe légèrement ce résultat, on applique à nouveau une recherche locale et on recommence.

Recherche tabou amélioration de la recherche locale itérée, qui va utiliser une « liste taboue » bannissant toute recherche autour des zones de l'espace déjà explorées.

Recuit simulé explore d'abord l'espace sans se restreindre aux parties donnant des solutions efficaces, puis se restreint de plus en plus au voisinage de celles-ci. Converge donc vers les solutions les plus efficaces trouvées, puis relâche les contraintes et explore autour de ces dernières, quitte à trouver des solutions vraiment moins efficaces. Recommence à se contraindre aux plus efficaces, etc...

Algorithmes génétiques imitent de la sélection naturelle, avec une population de solutions qui évoluent en mutant et en s'échangeant leurs caractéristiques entre elles. On peut même faire évoluer des populations séparément avec les modèles en îles, pour avoir plusieurs populations très différentes.

Colonies de fourmis imitent la encore la nature en simulant des phéromones déposées par des fourmis virtuelles, qui orientent la recherche au fil du temps.

2.2.5 Tests

todo

2.2.6 Conclusion

Il est intéressant de constater que les heuristiques, les méthodes de recherche locales et les métaheuristiques sont des choses extrêmement générales, utilisées pour résoudre énormément de problèmes demandant d'explorer un espace extrêmement grand.

Elles ne sont donc pas propres au voyageur du commerce, même si on a vu comment, dans ce cas précis, on pouvait obtenir des résultats corrects en se passant de métaheuristiques. On pourrait donc améliorer ces résultats en en utilisant.

3 UA : Programmation linéaire

3.1 Problème du sac à dos

3.1.1 Présentation du problème

Ce problème paraît simple en apparence : nous avons un ensemble d'objets, chaque objet pouvant avoir une masse différente et ayant une certaine valeur, et nous voulons remplir un sac à dos de manière à maximiser la valeur totale, sans dépasser une certaine masse maximale.

Résoudre ce genre de problème est utile par exemple en gestion de portefeuilles pour trouver le meilleur rapport entre rendement et risque, ou en découpe de matériaux, pour minimiser les chutes.

Ce problème est un problème d'optimisation linéaire, en effet, cela revient à résoudre le problème :

$$\begin{cases} \max v_i \\ i \in S \\ \sum_{j \in S} m_j \leq W \end{cases}$$

où S est l'ensemble des objets à choisir, v_i la valeur de l'objet i, m_i sa masse et W la masse maximale autorisée dans le sac.

Cependant la résolution de ce problème n'est pas simple : déterminer s'il est possible de dépasser une valeur minimale sans dépasser le poids maximal est un problème NP-complet.

3.1.2 Résolution exacte

Nous avons implémenté un algorithme de programmation dynamique, qui permet de résoudre le problème du sac à dos. Toutefois, il fonctionne uniquement si les poids des objets sont des entiers.

Sa complexité en temps est en $O(nW)$ et celle en mémoire en $O(W)$, avec n le nombre d'objets et W le poids maximum du sac.

Nous l'avons testé [1] sur plusieurs instances du problème, et l'algorithme est rapide.

3.1.3 Résolution approchée

Nous avons aussi implémenté l'algorithme glouton : celui-ci consiste simplement à choisir les « meilleurs » objets jusqu'à que la masse maximale soit dépassée. Le critère déterminant quels sont les meilleurs objets peut être la masse faible, le prix élevé, ou le rapport prix/masse élevé.

1. En utilisant le générateur de problèmes trouvé à l'adresse suivante : `http://www.diku.dk/~pisinger/codes.html`.

Cet algorithme est beaucoup plus rapide que le précédent, mais n'est qu'un algorithme approché ; les résultats obtenus sont cependant très satisfaisant (voir section 3.1.4).

3.1.4 Tests comparatifs

On remarque qu'en général, la résolution approchée en considérant le ratio prix/masse donne de très bon résultats, voire le résultat optimum. Le résultat que fournit cet algorithme est le moins bon quand la masse maximale autorisée est faible comparée à l'amplitude des valeurs que peuvent prendre le prix et la masse des objets.

Toutefois cet algorithme est beaucoup plus rapide, et a une complexité en temps de $O(n \log n)$ (pour le tri des objets), et ne nécessite en mémoire que la liste des objets, de plus il peut être utilisé quand les masses ne sont pas entières.

Nombre d'objets/ Amplitudes des prix et masses/ Masse maximale autorisée	Résultat optimum	Prix le plus élevé	Masse la plus faible	Meilleur ratio prix/masse
50/25/20	85	49/42.4%	67/21.2%	81/4.7%
500/25/500	2016	1125/44.2%	1725/14.4%	1983/1.6%
5000/25/500	5540	1175/79%	4577/17.4%	5540/0%
50000/25/500	11195	1175/90%	6684/40.3%	11195/0%
50000/1000/500	118260	5959/95%	101857/13.9%	118147/0.1%
50000/5000/100	100847	14931/85.2%	93532/7.3%	100282/0.6%

Table 1 – Résultats de l'algorithme glouton

4 UA : Jeux

4.1 Shifumi

Une stratégie simple et efficace à laquelle on pourrait penser pour gagner au Shifumi serait de jouer de manière aléatoire.

Et en effet, il s'avère que si les deux joueurs jouent de manière équiprobable, on a affaire à un équilibre de Nash : aucun changement de stratégie de la part d'un joueur ne pourra lui permettre d'augmenter ses chances de gagner.

De plus, si un adversaire ne joue pas de manière aléatoire (ou augmente la probabilité de jouer un certain élément), alors on pourra prévoir ce qu'il va jouer et donc trouver une stratégie qui pourra le battre. Les humains étant très mauvais pour jouer de manière aléatoire, il est assez facile d'écrire une stratégie permettant de les battre.

4.1.1 Stratégie développée

Afin de démontrer qu'un adversaire ne jouant pas aléatoirement est facile à battre, nous avons développé une stratégie qui se base sur des chaînes de Markov : en se basant sur les derniers éléments joués, elle regarde dans l'historique pour voir l'élément qui était joué le plus souvent par l'adversaire après les derniers coups joués.

Cette stratégie s'avère vraiment efficace contre un joueur humain. Toutefois, elle est prévisible : si on sait qu'on a affaire à une telle stratégie, on peut jouer de manière à la battre.

C'est pour cela qu'une stratégie aléatoire est la seule pouvant maximiser nos gains dans le pire des cas.

4.1.2 Variantes

Toutes les variantes du Shifumi qui consistent à rajouter des éléments pour obtenir un nombre d'éléments pair (par exemple pierre/papier/ciseaux/puits) vont créer un déséquilibre, car un élément sera moins efficace contre les autres. L'équilibre de Nash du jeu va alors consister à ne jamais jouer cet élément.

Si le nombre d'éléments est impair, alors le jeu pourra être équilibré, comme un Shifumi classique.

5 UA : Processus stochastiques

6 UA : Ingénierie robotique

7 Bilan

TODO : Un bilan devra être présenté dans ce dernier rapport final sur la façon de travailler de l'équipe au cours de toutes les UAs. Une autoévaluation devra y être conduite de sorte à faire apparaitre les points positifs et/ou les disfonctionnements apparus dans l'équipe.

8 Références

Références

[1] Wikipédia, *Problème du sac à dos*,
`http://fr.wikipedia.org/wiki/Probl%C3%A8me_du_sac_%C3%A0_dos`

[2] Wikipedia, *Knapsack problem*,
`http://en.wikipedia.org/wiki/Knapsack_problem`

[3] David Pisinger's optimization codes,
`http://www.diku.dk/~pisinger/codes.html`