Notes de cours – CM1

Introduction

Ce cours vise à présenter des techniques classiques pour résoudre de manière exacte ou approchée des *problèmes d'optimisation*. Formellement, un problème d'optimisation est défini par un ensemble $\mathcal{D}$ d'objets (le *domaine*), et une fonction f définie sur $\mathcal{D}$ qui exprime la qualité de chaque objet. Le but est de déterminer un élément du domaine qui maximise (ou minimise) f.

Par exemple (dessins), on lance une balle depuis le sol avec un angle α et on cherche à l'envoyer le plus loin possible. Ici le domaine $\mathcal{D}$ est $[0, 90]$ et la fonction f associe à chaque angle la distance parcourue (on suppose la vitesse initiale de la balle fixée). Il y a ici un unique maximum atteint quand $\alpha = 45$. Pour le trouver on cherche (comme au lycée !) les points d'annulation de f'.

Dans cet exemple, $\mathcal{D}$ est un intervalle réel. On parle dans ce cas d'*optimisation continue*, à l'inverse, on parlera d'optimisation *discrète* quand $\mathcal{D}$ sera un ensemble discret (par exemple d'entiers), voire *combinatoire* quand $\mathcal{D}$ est fini. Ce type d'optimisation sera abordé en premier lieu dans le cours. Des exemples d'optimisation continue suivront dans un deuxième temps.

En théorie, résoudre un problème d'optimisation combinatoire est facile : il suffit de parcourir le domaine $\mathcal{D}$, et de retenir la valeur maximale de f sur $\mathcal{D}$. Cependant, en pratique $\mathcal{D}$ est certes fini, mais (très) grand, ce qui rend ce genre d'approche naïve impossible.

Exemple (DESSIN) : Le problème du voyageur de commerce consiste, étant donné un ensemble de villes dont on connaît les temps de trajet entre chaque, à déterminer dans quel ordre les visiter (une et une seule fois chacune) avant de revenir au début en y passant le moins de temps possible. Ici, le domaine $\mathcal{D}$ est l'ensemble des tournées possibles, et la fonction f associe à chaque tournée la somme des temps de déplacement entre les villes consécutives. Si on dispose de n villes, il y a $(n-1)!/2$ tournées à considérer. Pour $n = 20$, et en supposant qu'on passe 1ns pour tester chaque tournée, on ne finira qu'au bout d'environ 2 ans !

Pour terminer sur notre exemple balistique, notons qu'on utilise de manière cruciale que f possède de bonnes propriétés (à savoir, être dérivable). Il aurait été beaucoup plus difficile de maximiser f si son comportement était plus chaotique (par exemple si f est le cours d'une action en bourse). Il s'agit ici d'une remarque générale : il est souvent essentiel d'étudier le comportement de f et de déterminer les bonnes propriétés qu'elle vérifie afin de définir des algorithmes efficaces. Cette approche ne marche pas en général : on ne peut pas toujours trouver de bonnes propriétés sur f ni d'algorithmes efficaces pour résoudre un problème d'optimisation (certains sont NP-complets). C'est par exemple le cas de TSP. Dans ces cas, on cherche plutôt des algorithmes efficaces dont on sait qu'ils renverront une solution *pas trop mauvaise* (par exemple dont on peut garantir qu'elle au pire moitié moins bonne qu'une solution optimale) : on parle alors d'algorithmes d'approximation.

Dans une première partie nous allons présenter des problèmes pour lesquelles la fonction f se comporte bien, ce qui permet d'obtenir de montrer qu'un algorithme naïf suffit à les résoudre (et de manière efficace).

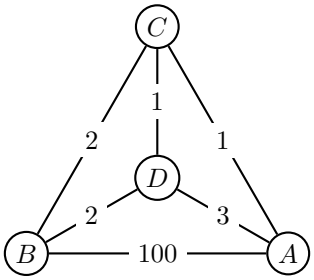
1 Algorithmes gloutons

1.1 Introduction

Un algorithme glouton est un algorithme qui construit progressivement une solution en choisissant à chaque étape le meilleur choix possible à cette étape. Par exemple sur TSP, un algorithme glouton consiste à construire une tournée pas à pas en se rendant à chaque fois à la ville la plus proche pas encore visitée.

Les deux points les plus importants à retenir à propos des algorithmes gloutons sont les suivants :

— Un algorithme glouton est souvent efficace (par exemple sur TSP, on obtient un algorithme quadratique en le nombre de villes).
— En revanche, un algorithme glouton ne fournit pas nécessairement une solution optimale (voir TD).

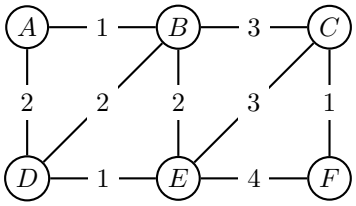


1.2 Arbres couvrants minimaux

On se donne un réseau constitué de machines reliées par des câbles (ayant des longueurs différentes). On souhaite simplifier le réseau en retirant des câbles mais les machines doivent toujours pouvoir communiquer entre elles. Autrement dit, on cherche un ensemble de câbles à conserver de sorte que le réseau reste connexe, et tel que la longueur totale de câbles est minimale.

On peut facilement se rendre compte que si un ensemble de câbles contient un cycle, on peut encore en retirer un, et il n'est donc pas minimal. Un ensemble solution induit donc un graphe connexe et sans cycle, c'est-à-dire un arbre. Cet arbre doit contenir tous les sommets, il est dit *couvrant*. Le problème revient donc à déterminer un arbre couvrant minimal de notre réseau.

Pour résoudre ce problème, on utilise un algorithme glouton : l'*algorithme de Kruskal*. Il s'agit de construire pas à pas un arbre couvrant, en choisissant à chaque étape le câble le plus court qu'on peut ajouter à notre arbre partiel.



1.2.1 Algorithme et preuve

On peut écrire plus formellement l'algorithme :

Algorithme 1 : Algorithme de Kruskal

Entrées : Un graphe $G = (S, A)$ pondéré non orienté
Sorties : Un ACM (arbre couvrant minimal)

```
1 E := ∅
2 L := tri(A) par ordre croissant des poids.
3 pour xy ∈ L (dans l'ordre) faire
4     si x et y ne sont pas dans la même composante de E alors
5         E := E ∪ {xy}
6         Fusionner les composantes de x et y
7 retourner E
```

Théorème 1. *À la fin, E contient les arêtes d'un ACM.*

Démonstration. On va démontrer l'invariant : « à chaque étape, il existe un ACM contenant les arêtes de E. »

Initialisation : $E = \varnothing$ et il existe un ACM.

Conservation : Supposons qu'à une certaine étape, on ait l'invariant.

Soit T l'ACM donné par l'invariant. Supposons qu'on ajoute l'arête xy à E. On veut montrer qu'il existe un ACM contenant E et xy.

Si $xy \in T$, alors c'est bon. Sinon, il existe un chemin de x à y dans T, et ce chemin ne contient pas que des arêtes de E (sinon x et y seraient dans la même composante de E). Parmi ces arêtes, soit uv celle de poids minimal. Notons qu'on considère les arêtes par ordre croissant de poids donc $w(uv) \geqslant w(xy)$.

On définit T' par $T' = T \cup \{xy\} \setminus \{uv\}$. Notons que T' est toujours un arbre couvrant, et son poids est

$$w(T') = w(T) + w(xy) - w(uv) \leqslant w(T)$$

Ainsi, T' est un ACM contenant E et xy, ce qui conclut. □

1.2.2 Union-find

On peut remarquer que cet algorithme nécessite de garder en mémoire quelles zones du graphes sont connectées par l'arbre partiel à chaque étape. On doit donc trouver une bonne structure de données pour représenter une partition des sommets. Cette structure doit permettre deux opérations majeures :

— déterminer dans quelle partie est un sommet donné (`find`).

— fusionner deux parties (`union`).

En termes de complexité, le tri des arêtes coûte $O(|A| \ln |A|)$. On a ensuite $O(|A|)$ appels à `find` et à `union`. La complexité dépend donc de la structure de données choisie pour représenter les partitions.

Une implémentation classique de cette structure consiste à représenter chaque partie par un arbre. La racine de chaque arbre est alors un représentant de chaque partie. Plus précisément, on va retenir un tableau `t`, où `t[i]` représente le père du sommet i (si i est la racine, alors `t[i]=i`).

TODO EXEMPLE

La partition initiale est donc le tableau `[0,1,2,... ]`. Pour déterminer si deux éléments sont dans la même partie, on calcule la racine de leur arbre, puis on compare les deux racines :

Algorithme 2 : `find`

Entrées : Un sommet x et une partition t

Sorties : La racine de l'arbre qui contient x

```
1 si x = t[x] alors
2 |  retourner x
3 sinon
4 |  retourner find(t[x])
```

Enfin, pour fusionner les parties de deux sommets x et y, il suffit de calculer les racines r_x et r_y de leurs arbres, et de faire pointer l'une vers l'autre. En général, on choisit de faire pointer la racine de plus petite hauteur vers celle de plus grande hauteur.

On remarque que la complexité de `union` et `find` est linéaire en la hauteur des arbres contenant x et y. Avec la gestion des hauteurs, on peut montrer le lemme suivant :

Proposition 1. *Si un des arbres obtenus a hauteur k, alors il contient au moins 2^k sommets.*

En particulier, s'il y a n sommets dans le graphe, la hauteur des arbres ne dépasse pas $\log n$. Ainsi, la complexité finale est $O(|A| \log |A|)$.

1.2.3 Amélioration

Quand on cherche un représentant avec `find`, on est idiots. On peut en profiter pour réduire le chemin d'un sommet à sa racine. On parle de compression de chemins.

On remplace l'implémentation de `find` par :

TODO DESSIN

Algorithme 3 : `union`

Entrées : Deux sommets x, y et une partition t
Sorties : La partition obtenue en fusionnant les parties de x et y

```
1 r_x := find(x, t)
2 r_y := find(y, t)
3 si r_x = r_y alors
4     Ne rien faire
5 si h(r_x) > h(r_y) alors
6     t[y] ← x
7 sinon
8     t[x] ← y
9 retourner t
```

Algorithme 4 : `find`

Entrées : Un sommet x et une partition t
Sorties : La racine de l'arbre qui contient x

```
1 si x ≠ t[x] alors
2     t[x] ← find(t[x])
3 retourner t[x]
```

Théorème 2. *Avec cette implémentation, les $O(|A|)$ appels à* **`union`** *et* **`find`** *dans l'algorithme de Kruskal prennent un temps $O(|A| \log^* |S|)$, où $\log^*(n)$ est le plus petit nombre k tel que*

$$\underbrace{\log(\log(\cdots(\log}_{k \text{ fois}}(n)\underbrace{)\cdots)}_{k \text{ fois}} \leqslant 1.$$

Bilan : Dans le cas général, on a une complexité en $O(A \ln(A))$. Si le tri de A a déjà été fait alors la complexité est en $O(A \ln^*(S))$.