Paradigmes, Heuristiques et Classe de complexité

I.S.I.A.

COURS 5

Franck.Nielsen@sophia.inria.fr

Année 1995-1996

INRIA SOPHIA ANTIPOLIS

Introduction

Un **paradigme** est une méthodologie, schéma algorihmique, utilisé pour construire une solution algorithmique à un problème donné.
→Diviser pour régner (tri par fusion).
→on examine la *programmation dynamique*

Une **heuristique** est une méthode d'approximation qui permet d'avoir une solution approchée d'une solution optimale en un temps meilleur que nécessiterai le calcul d'une solution optimale.
→on regardera le *voyageur de commerce*

Les **classes de complexité** permettent de classer la difficulté des problèmes et de donner des "solutions algorithmiques communes" à certains problèmes via des transformations appelées **réduction** adéquates.

Programmation dynamique

→exemple de paradigme.
→illustré par la multiplication série de n matrices.

$$M_1 \times M_2 \times \ldots \times M_n$$

on peut donner une méthode de calcul grâce aux parenthéses:

$$M_1 M_2 M_3 M_4$$

peut ainsi être calculé par:

$$((M_1(M_2 M_3))M_4),$$

ou encore par

$$((M_1 M_2)(M_3 M_4)).$$

→$C_n = \frac{1}{n+1}\binom{2n}{n}$ façons possibles (arbres strictement binaires à n nœuds) de parenthéser le tout. À chaque schéma de calcul, on peut lui associer un coût: le nombre de multiplications requises pour calculer ce produit.

→coût dépendant du schéma. On désire minimiser le coût des multiplications.

Principe de la Programmation dynamique

→on résoud des problèmes partiels et on combine les résultats pour obtenir la solution finale.
→Une seule façon de calculer $M_1 \times M_2$, $M_2 \times M_3$, ..., $M_{n-1} \times M_n$. On calcule les coûts associés à chaque paire de matrices consécutives.
→puis on calcule les coûts minimums de multiplication de triplets consécutifs $(M_1 M_2 M_3)$ de matrice:

$$c[M_1 M_2 M_3] = \min\{C[((M_1 M_2) M_3)], C[(M_1 (M_2 M_3))]\}$$

→puis pour toute sous-suite de 4 matrices consécutives, etc...

Un algorithme

```
for i := 1 to n do
    for j := i + 1 to n do cout[i, j] := +∞;
for i := 1 to n do cout[i, i] := 0;
for j := 1 to n − 1 do
    for i := 1 to n − j do
        for k := i + 1 to i + j do
            begin
t := cout[i, k − 1] + cout[k, i + j] + r[i] ∗ r[k] ∗ r[i + j + 1];
            if t < cout[i, i + j] then
                                    begin
                                    cout[i, i + j] := t;
                                    meilleur[i, i + j] := k;
                                    end;
            end;
```

→Algorithme en $O(n^3)$ et en espace mémoire quadratique.
Le paquet $(M_i \times \ldots \times M_j)$ a le meilleur coût:

$$\Big(\big(M_i \times M_{meilleur[i,j]-1}\big)\,\big(M_{meilleur[i,j]} \times M_j\big)\Big)$$

Quelques Paradigmes

– Diviser pour régner: on divise un problème en sous-problèmes. On fusionne alors les solutions des sous-problèmes (tri, enveloppe convexe, ...)

– Bisection-Décimation: on cherche à éliminer des données qui ne peuvent pas faire partie de la solution. Exemple calculer le 4^e plus petit élément.

$$c(n) = \begin{cases} O(1) & \text{si } n \leq 4 \\ O(n) + O(\frac{4}{5}n) & \text{sinon} \end{cases}$$

Soit $c(n) = O(n)$.

– Recherche exhaustive: on parcours un graphe de toutes les possibilités et on marque celles qui satisfont nos critéres, etc...

Heuristique: le voyageur de commerce

Trouver un circuit passant une fois par toutes les villes (n villes) de façon à minimiser le coût du circuit.
→solution exponentielle en $O(2^n)$. Algorithme de recherche exhaustive, backtracking, branch-and-bound, ...
→on désire calculer efficacement une solution approchée. Pas plus coûteuse que deux fois l'optimal:
on calcule l'arbre recouvrant de poids minimal (Minimum spanning tree).
Borne inférieure du coût par l'ARM puis on parcourt en profondeur: on obtient un cycle de longueur bornée par deux fois le coût de l'ARM. On supprime alors toutes les occurences multiples des villes dans ce parcourt en créant des raccourcis. On diminue ainsi la complexité à $O(m \log n)$ en ayant une solution approchée.

→Multitudes d'heuristiques connues aujourd'hui. Borne inférieure connue pour l'approximation, etc...

Classer les problèmes

→complexité des problèmes.
→À partir d'une complexité quadratique, les programmes implantant les algorithmes ne sont utilisables pour un grand nombre de données.

→Beaucoup de problèmes dont la meilleure solution algorithmique **actuelle** est exponentielle.

→On ne sait pas s'il existe une solution polynomiale pour ces problèmes.
→Envie de classer les problèmes "faciles" (de complexité polynomiale) aux problèmes "difficiles" (de complexité exponentielle).
→Limite floue. Par exemple:

1. Trouver un chemin de x à y de coût inférieur à M.
2. Trouver un chemin de x à y de coût supérieur à M.

$\boxed{1}$ est polynomial alors que les solutions connues pour $\boxed{2}$ sont exponentielles.

→Besoin de savoir si tout peut se faire en temps polynomial. D'où la grande question de l'informatique:

$$\boxed{\mathcal{P} = \mathcal{NP}?}$$

Classes $\mathcal{P}$ et $\mathcal{NP}$

→On appelle classe $\mathcal{P}$ l'ensemble des problèmes qui peuvent être résolus par un algorithme **déterministe** de façon polynomiale.
→On appelle classe $\mathcal{NP}$ l'ensemble des problèmes qui peuvent être résolus par un algorithme **non déterministe** de façon polynomiale.
Le non déterminisme permet de choisir une instruction parmi un ensemble fini d'instructions tel que l'algorithme converge vers la solution si elle existe (en pratique avec une infime probabilité!)
Problème **SAT**: satisfiabilité d'une expression booléenne.
Soit $F(b_1, ..., b_n)$ une expression booléenne à n variables.
Existe-il une instance de $b = (b_1, ..., b_n)$ qui donne $F(b)$ à vrai?
→Algorithme non déterministe linéaire mais algorithme déterministe exponentiel!!!

début
pour k **de** 1 **à** n **faire**
$b_k \leftarrow CHOIX(Vrai, Faux)$;
si $F(b) = Vrai$ **alors** F est satisfiable
sinon F non satisfiable;
fsi;
fin

Réduction Polynomiale

→Permet de relier la difficulté des problèmes entre eux.
→opérateur de **réduction**, **réduction polynomiale**, ...
Un problème P_1 est polynomialement réductible à un problème P_2 ssi il existe une algorithme polynomial résolvant P_1, utilisant un algorithme résolvant P_2, de complexité polynomiale.
Soient:

P_1**: Circuit Hamiltonien.** Trouver un circuit simple qui comprend tous les sommets d'un graphe donné.

P_2**: Problème décisionnel.** Étant donné un graphe de villes valué des distances inter-villes, trouver un chemin passant par toutes les villes de longueur inférieure à une constante fixée M.

Réduction linéaire

Réduction de P_1 à P_2:

- Associer à chaque sommet une ville.
- Si deux villes sont reliées par une arête, valuer la distance inter-ville par 1, ou 2 sinon.
- Executer un algorithme résolvant P_2 sur l'ensemble des villes avec $M = n$. On obtient soit un chemin de longueur n soit la non-existence d'un tel chemin. Si le chemin existe alors on a un cycle hamiltonien.

La transformation (réduction) se fait en temps linéaire.
$\rightarrow$Ici réduction triviale, mais la plupart du temps les réductions à des problèmes types ne sont pas directes!!!

Problèmes $\mathcal{NP}$-difficiles et $\mathcal{NP}$-complets

Existe-t-il des problèmes de $\mathcal{NP}$ canoniques, dont tout autre problème de $\mathcal{NP}$ lui est polynômialement réductible?
→OUI!
Un problème est $\mathcal{NP}$-difficile ssi tout problème de $\mathcal{NP}$ lui est polynomialement réductible.
Théorème de COOK 1971: **SAT** est $\mathcal{NP}$-difficile (thèse).
→Ainsi, si on trouve une solution polynômiale de **SAT** alors on a une solution polynômiale pour tout problème de $\mathcal{NP}$.
Un problème est $\mathcal{NP}$-complet ssi il est $\mathcal{NP}$-difficile et dans $\mathcal{NP}$.
→**SAT** dans $\mathcal{NP}$-complet.
→Recueil de problèmes $\mathcal{NP}$-complets: Garey et Johnson: plus de 300 références.

Quelques problèmes $\mathcal{NP}$-complets

- satisfiabilité d'une fonction booléenne sous forme normale conjonctive à trois litéraux dans chaque disjonction. 3-**SAT**.
- Trouver un stable à m sommets dans un graphe.
- Problème du sac à dos (knapsack). Soit $\mathcal{S}$ un ensemble d'entiers et t un entier donné, existe-il $\mathcal{A} \subseteq \mathcal{S}$ telle que $\Sigma_{x \in \mathcal{A}}\, x = t$?
- Cycle hamiltonien
- Voyageur de commerce
- Colorier les sommets d'un graphe à l'aide de 3 couleurs de sorte que deux sommets adjacents aient des couleurs différentes. 4 couleurs toujours possible pour un graphe planaire (ex. map-monde).
- Problème de partition: séparer un ensemble d'entiers en deux sous-ensembles de même somme.

Un mystère: $\mathcal{P} = \mathcal{NP}$?

→grande question de l'Informatique (Turing Awards)
→au départ tout le monde pensait que la réponse était OUI ($\simeq$ 10 ans). Aujourd'hui les gens pensent que la réponse est NON.
→Quand on se trouve confronté à un problème $\mathcal{NP}$-complet, on cherche à trouver des **heuristiques**, c'est-à-dire des approximations.
→Donner des bornes sur les approximations ainsi trouvées. On cherche bien évidemment des heuristiques polynômiales.

→Limites théoriques des machines. Problèmes rencontrés en pratique.
→Besoin d'algorithmes "pratiques" et performants.