Leçon 931 : Schéma algorithmiques. Exemples et application.

Julie Parreaux

2018 - 2019

Références pour la leçon

[1] Beauquier, Berstel et Chretienne, *Éléments d'algorithmique.*
[2] Cormen, *Algorithmique.*
[3] Kleinberg et Tardos, *Algorithms design.*
[4] Moret et Shapiro, *Algorithms from P to NP.*
[5] Sipser, *Introduction to the Theory of Computation.*

Développements de la leçon

Étude de l'alignement optimal de deux mots Algorithme de Dijkstra

Plan de la leçon

Introduction **2**

1 Algorithmes de partitionnement **2**
1.1 Diviser pour régner [2, p.59] . . . 2
1.2 Programmation dynamique [2, p.333] . . . 2
1.3 Algorithmes glouton : une heuristique . . . 2

2 Algorithmes d'exploration **3**
2.1 Backtracking . . . 3
2.2 Branch and Bound . . . 3

Motivation

Défense

La méthode naïve, le brute force n'est pas toujours efficace. Il existe plusieurs types d'algorithmes permettant de résoudre un problème. Selon la nature du problème (optimisation, récursif, ...) ces approches peuvent être plus ou moins efficaces. Nous allons en étudier quelques unes ici, et plus particulièrement des paradigmes de partitionnement et d'exploration.

Ce qu'en dit le jury

Cette leçon permet au candidat de présenter différents schémas algorithmiques, en particulier «diviser pour régner », programmation dynamique et approche gloutonne. Le candidat pourra choisir de se concentrer plus particulièrement sur un ou deux de ces paradigmes. Le jury attend du candidat qu'il illustre sa leçon par des exemples variés, touchant des domaines différents et qu'il puisse discuter les intérêts et limites respectifs des méthodes. Le jury ne manquera pas d'interroger plus particulièrement le candidat sur la question de la correction des algorithmes proposés et sur la question de leur complexité, en temps comme en espace.

Introduction

— Brute force : exemple et contre-exemple

1 Algorithmes de partitionnement

Une première approche de l'algorithmique est de partager le problème en sous problème dont ceux-ci sont à priori plus simple à calculer.

1.1 Diviser pour régner [2, p.59]

Diviser pour régner est plutôt un paradigme récursif. Le principe est de partager la résolution du problème en le divisant puis en recombinant les sous-instance ainsi résolu.

— Paradigme + exemple tri fusion / enveloppe convexe

— Master théorème + applications + contre-exemple : tri rapide [1]

— Application théorique : théorème de Savitch

— Application approximation de problème NP-complet TSPE [4, p.480]

— Limites : deux infinité : infinité de l'espace de la solution : pile d'appel (problème d'optimisation NP-complet) et infinité structurelle : recoupement (Fractale, Fibonacci)

1.2 Programmation dynamique [2, p.333]

Cette approche vient rectifier les lacunes du à l'infinité structurelle du paradigme diviser pour régner en stockant dans des tables les résultats intermédiaires. Il s'applique généralement aux problèmes d'optimisation.

— Quand doit-on l'utiliser ?

— Paradigme (la recette de cuisine) + exemples (distance d'édition DEV, PLSC, CYK)

— Memoisation + exemple (découpe de barre)

— Application à l'approximation de problème NP-complet Sac à dos

1.3 Algorithmes glouton : une heuristique

Cette approche permet d'optimiser des problèmes en considérant une approche locale : elle vient rectifier les limite dû à l'infinité de l'espace des solutions. C'est une première heuristique simple qui peut parfois donner de bons résultats. Elle vient rectifier les lacunes de l'approche diviser pour régner de du à l'infinité de l'espace des solutions.

— Paradigme
— Exemple optimal : Dijkstra DEV + Prim Kruskal (ces algorithmes sont optimaux)
— Application à la compilation : analyse syntaxique (LL(k), LR(k))
— Application à l'approximation de problème NP-complet (Set cover, TSP)

2 Algorithmes d'exploration

Lorsque l'ensemble des solutions devient vraiment très important : le partage choisi par les algorithmes de partitionnement ne sont pas nécessairement les meilleurs. On utilise alors des algorithmes d'explorations qui d'une certaine manière partitionne l'espace des solutions mais sont l'exécution se base sur l'exploration d'un arbre. Remarque : on a aussi des algorithmes d'exploration sur les graphes.

2.1 Backtracking

Cette approche de l'exploration est plus efficace dans le cadre d'un algorithme d'exploration sur un problème de décision (où on cherche à construire une instance). On a des améliorations de cette approche nous permettant de ne pas explorer des parties de l'arbre (backjumping avec CDCL).

— Principe On fait un parcours en profondeur de notre arbre des possibles.
— Application à la logique : DPLL
— Application à la compilation : analyse syntaxique pour toute grammaire Attention, si on a CYK, ce n'est plus nécessaire

2.2 Branch and Bound

Cette approche de l'exploration est plus efficace dans le cadre d'un algorithme d'exploration sur un problème d'optimisation. On se donne une fonction de coût, une heuristique facile à calculer, (qui majore ou minore celle du problème) qui nous permet de classer les solutions en fonction de leur affinité avec la solution.

— Principe On effectue un sorte de parcours en largeur.
— Exemple ?

Remarque : Dans les deux cas, on peut être amené à explorer l'ensemble de l'arbre qui nous sert de base.

Quelques notions importantes

Les principes de la programmation dynamique

Nous allons présenter la programmation dynamique et ses principes [2, p.333]. La programmation dynamique s'inscrit dans le but de résoudre les limites du paradigme diviser pour régner dans le cas d'une structure "infinie". En effet, elle s'applique à des problèmes possédant des sous-structures optimales dont les sous-problèmes qui lui sont liés se chevauche. On a une belle application du principe compromis temps-espace.

Définition (Paradigme de la programmation dynamique)**.** *Écrire un algorithme sous le paradigme de la programmation dynamique se réalise en quatre étapes.*

1. *Caractériser la structure optimale dans notre problème. Un problème exhibe une sous-structure optimale si une solution optimale au problème contient en elle des solutions optimales de sous-problèmes. Un alignement optimale pour un préfixe de taille n contient un alignement optimal du préfixe de taille $n-1$. Pour exhiber cette sous-structure optimale on suit la méthode suivante :*
   (a) *Exhiber le choix nécessaire pour trouver la solution du problème (Ins, Del et Sub).*
   (b) *On suppose connaître une solution optimale que nous possédons.*
   (c) *On donne alors les sous-problèmes qui en découlent et on décrit comment caractériser au mieux ce sous-espace (les alignements optimaux pour les préfixes de taille inférieure).*
   (d) *Montrer que les sous-solutions sont optimales (raisonnement par l'absurde).*
2. *Définir récursivement la valeur de cette structure optimale. Pour cela, on calcul le graphe des sous-problème qui nous permet d'établir l'ordre du calcul des différents sous-problèmes.*

   Définition. *Le graphe des sous-problèmes est un graphe orienté contenant chacun des sous-problèmes et leur dépendance.*

3. *Calcul de la valeur d'une solution optimale, généralement de manière ascendante. On a deux approches possibles.*

   Mémoïsation *procédure écrite de manière récursive classique dont le résultat de chaque sous-problème est stocké dans un tableau. On commence par vérifier si on a déjà résolut un problème.*

   Ascendante *on résout les problèmes du plus petit au plus gros en utilisant les résultats précédemment calculés.*
4. *Construction d'une solution optimale (si nécessaire).*

Une solution exponentielle d'un problème peut alors devenir polynomiale si et seulement si les sous-problèmes distincts impliqués sont en nombre polynomial et peuvent être résolus en temps polynomial.

Les principes du diviser pour régner

L'approche diviser pour régner est un paradigme qui se décompose en trois parties.

Diviser On sépare le problème en sous-problèmes identiques au problème de départ mais sur des instances plus petites.

Résoudre On résout (généralement récursivement) les sous-problèmes.

Combiner On reconstruit la solution à partir des sous-problèmes.

Théorème (Master theorem). *Soient $t : \mathbb{N} \mapsto \mathbb{R}_+$ une fonction croissante à partir d'un certain rang n_0, $b \geq 2$ entiers, $k \geq 0$ entier et $a, b, c, d > 0$ réels positifs tels que pour tout n tel que $\frac{n}{n_0}$ puissance de b :*

$$\begin{cases} t(n_0) = b \\ t(n) = at(\frac{n}{b}) + cn^k \end{cases}$$

Alors, on a :

$$t(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \\ \Theta(n^k \log_b n) & si\ a = b^k \\ \Theta(n^{\log_b a}) & si\ a > b^k \end{cases}$$

Une autre utilisation du paradigme diviser pour régner est l'approximation de problème NP-complet [4, p.480]. On donne ici un exemple. Considérons le problème du voyageur de commerce sur un plan euclidien. L'approche diviser pour régner approchant cette algorithme se décompose comme suit. Comme le nombre de point est fini, on peut considérer une enveloppe convexe rectangulaire encadrant tous les points dont les extremums soient dans les côtés.

Diviser Partition du rectangle en deux de tel sorte que la division se fait par une droite parallèle à la largeur du rectangle passant par un des sommets du problème et séparant l'espace tel que le nombre de points soient identiques en haut et en bas.

Résoudre On résout récursivement le problème dans les deux rectangles.

Combiner On reconstruit le tour en s'assurant que chaque point apparaît une seule fois dans le tour.

La complexité de cette approximation est en $\Theta(n \log n)$ car chacune des solutions est en temps constante pour chacun des sommets et la combinaison de fait en $\Theta(n \log n)$ (on peut appliquer le master theorem : on se retrouve dans le deuxième cas).

L'approximation via le paradigme diviser pour régner est efficace en terme de complexité, cependant, l'approximation n'est pas optimale.

Références

[1] D. Beauquier, J. Berstel, and P. Chrétienne. *Eléments d'algorithmique*. Manuels informatiques Masson. Masson, 1992.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Algorithmique, 3ème édition*. Dunod, 2010.

[3] J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson international Edition, 2006.

[4] B.M.E. Moret and H.D. Shapiro. *Algorithms from P to NP*. The benjamin / Cumming Publishing Company, 1990.

[5] M. Sipser. *Introduction to the Theory of Computation*. Cengage learning, 1133187811.