PUBLICATIONS DU DÉPARTEMENT DE MATHÉMATIQUES DE LYON

M. CHEIN

Analyse de la complexité des programmes, des algorithmes et des problèmes

Publications du Département de Mathématiques de Lyon, 1982, fascicule 1B
« Quelques thèmes de la théorie des algorithmes », , p. 1-13

<http://www.numdam.org/item?id=PDML_1982___1B_1_0>

ANALYSE DE LA COMPLEXITE DES PROGRAMMES, DES ALGORITHMES ET DES PROBLEMES

par M. CHEIN

Des programmeurs aux logiciens aucun informaticien ne peut échapper à la "complexité". La multiplicité des points de vue et des objectifs, des méthodes utilisées et des résultats obtenus, ont conduit à une certaine confusion et à de nombreux malentendus. Le but de cet exposé est de présenter rapidement ce qui semble le plus intéressant pour un programmeur. S'il était naturel que l'analyse de la complexité des algorithmes se développe en premier lieu, nous commencerons par exposer la problématique à partir d'un programme (exécutable sur un système informatique donné) pour éviter certaines discussions, sur la taille des données ou les opérations dénombrées par exemple, qui risqueraient de masquer l'essentiel. Les deux derniers paragraphes ne font qu'exposer très brièvement, car il existe une littérature abondante sur ces sujets, l'analyse de la complexité des algorithmes et des problèmes, et les liens entre ces diverses disciplines.

1 - ANALYSE DE LA COMPLEXITE DYNAMIQUE DES PROGRAMMES.

1.1 Le passage du concret à l'abstrait.

Nous allons, pour commencer, poser une question naturelle dans une situation qui peut sembler particulièrement simple : disposant d'un certain programme nous voulons savoir -avant toute exécution de celui-ci- quelle sera la quantité de "travail" réalisée lors de l'exécution de ce programme, et ceci en fonction des données.

On peut considérer que le temps d'exécution et l'espace-mémoire utilisé sont des bonnes mesures de cette quantité de travail. Et, de fait, avant d'exécuter un programme, sur un système donné, il peut être utile de savoir si l'espace-mémoire disponible sera suffisant et si le temps d'exécution sera raisonnable. Si le programmeur fait souvent une analyse a priori de l'espace-mémoire nécessaire il fait plus rarement une telle analyse pour le temps d'exécution. Ce peut être parce que :

- il ne sait pas comment faire
- c'est difficile à faire
- il est commode d'utiliser un jeu de données et un chronomètre pour avoir une évaluation du temps d'exécution.

Cette dernière évaluation, qui peut être "efficace" ou sans aucune signification suivant la manière dont l'expérience est menée, a pour inconvénients majeurs, en plus du fait qu'elle nécessite une exécution, la dépendance du jeu des données et de l'environnement.

La manière la plus simple d'avoir des résultats indépendants de l'environnement c'est de supprimer celui-ci ! On reste alors à l'intérieur du langage dans lequel le programme est écrit. La quantité de travail dont nous parlions au début pourra alors être mesurée par le nombre d'objets d'un certain type utilisés par le programme, et par le nombre d'opérations d'un certain type exécutées. Si, dans la plupart des systèmes il est facile de passer d'un nombre d'objets (d'un langage) à de l'espace-mémoire, il semble plus difficile de passer d'un nombre d'opérations (d'un langage) à du temps d'exécution. C'est ce remplacement d'une quantité de "temps" par un nombre d'opérations qui permet :

- d'obtenir des résultats grâce à des techniques d'analyse combinatoire et d'analyse asymptotique ;

- d'obtenir des résultats généraux puisqu'on ne s'intéresse qu'aux opérations d'un langage et pas à un système réel supportant ce langage ;

- de nombreux malentendus car le passage d'un nombre d'opérations d'un langage à du "temps" sur un système donné n'est pas toujours évident.

Dans le paragraphe suivant nous exposerons brièvement la problématique concernant l'évaluation du nombre d'opérations.

1.2 Nombres d'opérations.

On peut considérer que le nombre d'opérations, d'un certain type, exécutées par un programme est fonction des données de deux manières différentes. D'un point de vue quantitatif les données peuvent être plus ou moins "grandes". Comme on peut supposer que les données sont des séquences finies de caractères d'un alphabet fini, on pourra compter le nombre d'opérations en fonction de la taille des données : ce sera soit la longueur des mots (par exemple $\lfloor \log a \rfloor + \lfloor \log b \rfloor + 2$ dans le cas d'un programme calculant le produit de deux entiers a et b codés en binaire) soit une fonction simple de paramètres permettant d'avoir une évaluation de la longueur du codage (par exemple le nombre de sommets plus le nombre d'arêtes pour un programme manipulant des graphes). D'un point de vue qualitatif il est bien connu que deux données de même taille ne nécessiteront pas nécessairement le même nombre d'opérations. Il suffit de penser à l'exécution d'un programme réalisant l'algorithme d'Euclide pour déterminer le p.g.c.d. de deux entiers, lorsque ces deux entiers sont égaux ou lorsque ce sont deux nombres de Fibonacci consécutifs. Il est donc essentiel de savoir déterminer, pour l'ensemble des données que le programme devra traiter, le nombre maximal d'opérations exécutées.
On parle dans ce cas d'analyse dans le plus mauvais cas.
Lorsque les calculs des nombres d'opérations sont difficiles on envisage souvent des évaluations asymptotiques -en fonction de la taille des données- que l'on exprime en utilisant la notation en O (lire en grand O) qui serait dûe à Bachmann (1854) et remise à l'honneur par Knuth (12) :
Si f est, par exemple, une application de $\mathbb{N}$ dans $\mathbb{R}$, on note par O(f) l'ensemble des applications g, de même type que f, telles qu'il existe une constante positive c et un entier n_0 avec $g(n) \leq c \times f(n)$ ceci pour tout $n > n_0$.

$$O(f) = \{ g : \mathbb{N} \to \mathbb{R} \ ; \ \exists c > 0 \ \exists n_0 \ \forall n > n_0 \quad g(n) \leq c \times f(n) \}$$

Ainsi $1000\, n^2 + 3\, n \log n + 10^7 n$ est dans $O(n^2)$, de même que $n \in O(n^2)$. Ce dernier exemple montre bien qu'il est utile de préciser si l'évaluation asymptotique O(f), est "atteinte", c'est-à-dire s'il existe une constante $c' > 0$ et une infinité de données de taille n, pour lesquelles le nombre d'opérations exécutées est $\geq c' f(n)$.

Pour de nombreux programmes, en particulier pour ceux résolvant des problèmes combinatoires, on ne peut obtenir que des résultats du type suivant :

> Le nombre d'opérations d'un certain type (+, x, -, opér. booléennes, comparaisons, affectations,...) exécutées par un programme traitant des données de taille n est en O(f), où f est une fonction numérique de n. De plus cette évaluation asymptotique est atteinte.

Si on connait (bien) l'ensemble des données on peut chercher à obtenir des résultats autres que le nombre maximal d'opérations d'un certain type. Les difficultés de l'analyse probabiliste d'un programme sont bien connues : il y a d'abord le problème du choix d'une loi de probabilité "raisonnable" sur les données, puis les difficultés des calculs eux-mêmes qui, de plus, doivent être refaits à chaque fois que l'on change de loi ! L'idéal à atteindre étant d'obtenir la distribution de la variable aléatoire "nombre d'opérations d'un certain type", c'est-à-dire, dans le cadre qui nous intéresse, déterminer pour tout entier p le nombre de données de taille n nécessitant p opérations d'un certain type; on est content lorsque l'on a pu obtenir des évaluations asymptotiques de la moyenne et de l'écart-type (4)(10).
Cette analyse en moyenne devient essentielle, et difficile, dans le cas de programmes arborescents (Cf par exemple (14)).

1.3 Des nombres d'opérations au temps.

Pour avoir des chances de pouvoir établir une correspondance précise entre les nombres d'opérationset le temps d'exécution il semble nécessaire de suivre la démarche proposée par Cohen et Zuckerman (8).
Si t_i est une estimation du temps nécessaire à l'exécution, sur un système donné, de l'opération i, du langage dans lequel est décrit le programme à analyser, et si on connait $a_i(n)$, le nombre maximal d'exécution de l'opération i pour les données de taille n, alors $\sum_i t_i \times a_i(n)$ peut être une estimation du temps maximal d'exécution du programme pour les données de taille n. Les précautions de langage ne sont pas superflues. En effet, il faut non seulement pouvoir calculer les $a_i(n)$ pour toutes les opérations du langage (dans (8) Cohen et Zuckerman proposent 40 opérations pour un langage de type Pascal, certaines d'entre elles, comme la déclaration d'un tableau à la dimension, étant paramétrées).

Mais il faut aussi pouvoir donner des valeurs raisonnables aux t_i ce qui semble difficile sur de nombreux "gros" systèmes (optimiseur de code, multiprogrammation, temps partagé, gestion des entrées/sorties,...). On peut sans doute utiliser cette méthodologie avec plus de succès pour comparer les temps d'exécution de deux programmes résolvant le même problème, écrits dans le même langage et exécutés sur un même système. C'est ce que Cohen et Roth (7) ont fait par la comparaison d'un programme réalisant l'algorithme classique et d'un programme réalisant l'algorithme de Strassen calculant le produit de deux matrices.
Il devrait être assez clair maintenant que dire que si $\alpha < \beta$ alors un programme en $O(n^\alpha)$ dans le plus mauvais cas résolvant un problème donné est "meilleur" qu'un programme en $O(n^\beta)$ résolvant ce même problème, ne permet pas nécessairement de dire quelque chose sur le temps d'exécution de ces deux programmes. On ne fait que dire que $\alpha < \beta$! En effet ce type de résultat ne permettra une comparaison a priori des deux programmes que :

- si on a compté les "bonnes" opérations
- si on a un rapport entre opération et temps d'exécution
- si les évaluations asymptotiques sont "proches" des valeurs exactes y compris pour des petites valeurs de la taille
- si les plus mauvais cas sont "fréquents".

Ce qui est sûr c'est que si on a, à système, langage, problèmes et données constants un programme en $O(n^\beta)$ nécessitant un temps d'exécution plus petit qu'un programme en $O(n^\alpha)$ avec $\alpha < \beta$, l'analyse des programmes et des données est incomplète, et qu'il y a des choses intéressantes à faire.
Il est maintenant possible, en ayant présents à l'esprit tous les problèmes soulevés précédemment, de résumer les intérêts de l'analyse des programmes :

- évaluation -a priori- du temps d'exécution d'un programme
- comparaison -a priori- de plusieurs programmes
- meilleure connaissance du problème
- constructions d'algorithmes et de programmes "efficaces" pratiquement (tris, flots, dictionnaires dynamiques, transformée de Fourier rapide, algorithmes géométriques,...)
- remise à leur juste place des expériences numériques.

Nous terminerons ce paragraphe par quelques mots sur ce dernier point.

1.4 Analyse empirique

De toutes les difficultés rencontrées lors de la démarche pour analyser un programme décrite précédemment, il ressort que les expériences numériques et leur interprétation, que l'on pourrait appeler l'analyse empirique des programmes comme le fait Knuth (12), est toujours utile. Le problème crucial est celui du choix du jeu des données : il vaut mieux essayer de construire des données "difficiles" (il existe ainsi des listes de problèmes tests pour des problèmes importants) que de les construire "au hasard". En effet le "hasard" se réduit le plus souvent à des lois plus ou moins uniformes que ne suivront pas nécessairement les données réelles.

Ayant déterminé un jeu de données on peut, en insérant des compteurs pour certaines opérations dans le programme à analyser ou en analysant à intervalle régulier le mot d'état de la machine pour savoir quelle instruction est exécutée, mettre en évidence les endroits cruciaux du programme. (A partir d'une telle analyse appliquée au programme Fordap qui insère des compteurs dans un programme Fortran, Knuth (12), a pu diviser par 2 le temps d'exécution de ce programme Fordap sur son échantillon de programmes).

Au lieu de compter des opérations,ettoujours sur un échantillon bien choisi, on peut naturellement indiquer des temps d'exécution. Ceux-ci seront intéressants dans le cadre d'un système donné "déterministe", c'est-à-dire si la quantité de moyens utilisés lors de l'exécution d'un programme n'est fonction que du programme et des données et pas de l'état du système au moment où on exécute le programme.

Depuis Knuth (12) en 1971 l'apport le plus important dans ce domaine semble être celui de Cohen (6) en 1977. Il propose un système qui permet de construire au cours de l'exécution une base de données contenant non seulement les nombres d'exécutions de telle ou telle opération, mais aussi le nombre de fois où une comparaison est vraie, les valeurs extrêmes de certaines variables,...et un langage pour interroger cette base de données.

1.5 Analyse de la complexité statique.

L'analyse de la complexité dynamique d'un programme a pour but de mesurer et d'améliorer son "efficacité". Ceci est particulièrement important dans deux cas extrêmes :

- La résolution de "gros" problèmes (intelligence artificielle, emploi du temps, recherche opérationnelle,...), car on peut alors franchir dans un sens ou un autre la frontière des problèmes résolubles (pratiquement) ;

- la construction de programmes utilisés très souvent (opérations arithmétiques, tris, gestion de dictionnaires, opérations matricielles, parcours d'arborescences,...)

Mais "l'efficacité" est loin d'être la seule vertu d'un programme.
L'analyse de la complexité statique d'un programme a pour but de mesurer, et d'améliorer, la "valeur" de celui-ci : sa simplicité, sa lisibilité, sa modularité, etc... Ce type d'analyse se ramène généralement à la mesure de quantités qui ne dépendent que du texte du programme et pas de son exécution.
De nombreuses fonctions de "valeur" ont été proposées combinant des mesures associées à des unités syntaxiques (nombre d'opérateurs et d'opérandes par exemple) et d'autres associées au graphe de contrôle de programme (nombre cyclomatique par exemple).(Cf un article récent (16)).
Nous ne faisons que mentionner ici ce type d'analyse non pas à cause de son manque d'intérêt mais parce qu'elle n'a que peu de support avec l'analyse dynamique. (Signalons la détermination d'une base de cycles du graphe de contrôle qui permet d'insérer un nombre minimal de compteurs si on veut faire une analyse dynamique empirique).

II - ANALYSE DE LA COMPLEXITE DES ALGORITHMES.

On appelle : analyse des algorithmes, analyse de la complexité des algorithmes, complexité concrète... la démarche décrite en 1.2 et appliquée non plus à un programme mais à un algorithme (au sens intuitif). Knuth (11) rappelle que la première analyse de l'algorithme d'Euclide a été publiée par Lamé en 1844. Dans un autre domaine Lemoine s'intéressait, dans son livre "Géométrographie"publié en 1907, à ce qu'il appelait la "simplicité" d'une construction géométrique plane, c'est-à-dire au nombre d'opérations euclidiennes : placer la pointe du compas en un point, placer la pointe du compas sur une droite, placer la règle en un point, tracer un cercle, tracer une droite, nécessitées par cette construction.

Si nous avons déja signalé que le passage d'un nombre d'opérations à des quantités de moyens informatiques utilisés pouvait poser des difficultés dans le cas de l'analyse d'un programme, ceci sera a fortiori vrai dans le cas de l'analyse d'un algorithme : tant il est vrai que la différence entre un programme et un algorithme, à notre niveau de discours, est que ce dernier utilise des opérations et des objets plus flous que le premier. Prenons par exemple le problème de la taille des données en fonction de laquelle on exprime les nombres d'opérations. Dans le cas d'un programme on a un codage fixé des données et la taille est la longueur de ce codage, alors que, bien souvent, dans le cas d'un algorithme on ne précise pas de codage des objets manipulés. Ne pas préciser le codage des objets a d'autres inconvénients que des discussions sur la taille des données. Ceci peut conduire à l'oubli du dénombrement d'opérations qui deviendront importantes pour un programme réalisant l'algorithme analysé. Dans l'analyse des algorithmes numériques on ne compte souvent que les opérations arithmétiques alors que dans les programmes réalisant ces algorithmes les nombres d'accès à des éléments de tableaux pourront être très importants pour le temps d'exécution.

Et si nous avons déjà dit qu'il n'y avait pas nécessairement de mystère lorsqu'un programme en $O(n^{\beta})$ était plus "efficace" qu'un programme en $O(n^{\alpha})$ avec $\alpha < \beta$, ceci est a fortiori vrai dans le cas de l'analyse des algorithmes. (sur cette question nous renvoyons encore une fois le lecteur au remarquable article de Cohen et Roth (7) sur l'analyse de l'algorithme de Strassen).

L'analyse de la complexité d'un algorithme qui est, à notre avis, aussi cruciale que sa démonstration, est maintenant bien entrée dans les moeurs des informaticiens. Mais ceci ne devrait pas satisfaire les programmeurs.
Dans les diverses phases de décomposition-démonstration qui conduisent d'un algorithme à un programme, on doit également faire évoluer l'analyse de la complexité en fonction des précisions apportées aux objets et aux opérations. Car il n'est pas difficile de passer d'un algorithme en $O(n^{\alpha})$ à un programme le réalisant en $O(n^{\beta})$ avec $\alpha<\beta$ ...

III - ANALYSE DE LA COMPLEXITE DES PROBLEMES.

Nous avons maintenant un certain problème et divers algorithmes pour le résoudre est-il possible de faire mieux (au sens des mesures décrites précédemment) ? En d'autres termes est-il possible d'avoir des informations sur la complexité de tous les algorithmes (connus ou non) pour résoudre un problème donné ? Ou encore:quels sont les meilleurs algorithmes pour résoudre ce problème ? On peut aborder ces questions avec la formalisation suivante :

Soit P un certain problème et I_n l'ensemble des instances (des données) de P de taille n. Si $\mathcal{C}_n$ est une classe d'algorithmes, qu'il peut être commode de définir par une machine à capacités limitées, de résolution des instances de P de taille n, on s'intéresse à la quantité :

$$\min_{A \in \mathcal{C}_n} \max_{I \in I_n} op_A(I)$$, où $op_A(I)$ est

le nombre d'opérations (d'un certain type) exécutées par A pour résoudre P sur I. (On peut aussi s'intéresser à la moyenne de $op_A(I)$ si on a une loi de probabilité sur I_n).

Si on peut prouver que cette quantité est $\geq \mu(n)$ on sait que sur la classe $\mathcal{C}_n$ il est impossible de faire mieux que $\mu(n)$. L'idéal étant ensuite d'exhiber un algorithme A de $\mathcal{C}_n$ avec $\max_{I \in I_n} op_A(I) = \mu(n)$.

A est alors un algorithme "optimal" (au sens du critère utilisé et sur la classe $\mathcal{C}_n$). Nous ne reprendrons pas ici la discussion sur les mesures utilisées mais nous insisterons sur un point nouveau : des minorations de ce type sont intéressantes si la classe $\mathcal{C}_n$ est "grande", par exemple si elle contient tous les algorithmes connus à ce jour.

Lorsqu'on dit, par exemple, que le shcéma de Hörner pour calculer la valeur d'un polynôme en un point est optimal on considère comme mesure le nombre d'opérations arithmétiques +, -, * (pas de division) ceci dans le plus mauvais cas (on peut calculer x^n plus rapidement qu'avec le schéma de Hörner), et en utilisant des algorithmes linéaires, c'est-à-dire des algorithmes constitués d'une séquence d'instructions de type c_i := u op v , où c_i est une variable n'apparaissant pas dans des instructions précédentes, u, $v \in \mathbb{R}$ ou des c_j avec $j < i$ et op $\in \{+, -, *\}$.

Si dans cet exemple on peut associer un graphe sans circuit à un tel algorithme, dans de nombreux cas les classes d'algorithmes considérés seront associées à des arborescences. (Arborescences binaires pour les problèmes de tri, ou des problèmes de graphes (15), arborescences ternaires de décision linéaire pour des problèmes de plus courtes chaînes ou de décision quadratique pour des problèmes de géométrie plane (17),...)

Dans tous les cas précédemment cités les minorations obtenues sont bornées par des polynômes en la taille n des instances. Il existe quelques cas où sur des classes d'algorithmes assez "grandes" les minorations sont exponentielles, citons les résultats de Chvátal sur le nombre de stabilité d'un graphe (5) et ceux de Mc Diarmid sur le nombre chromatique d'un graphe (9). La question de savoir si, par exemple, pour ces deux derniers problèmes il existe des algorithmes pour lesquels le nombre maximal d'opérations dans le plus mauvais cas soit borné supérieurement par un polynôme en la taille des instances nous amène à "one of the foremost open questions of contemporary mathematics and computer science", comme disent Garey et Johnson, P = NP ? Mais ceci est une autre affaire et nous renvoyons le lecteur intéressé à leur livre (3).

B I B L I O G R A P H I E

Quelques ouvrages essentiels sur l'analyse des algorithmes et des problèmes :

(1) AHO (A.V.), HOPCROFT (J.E.), ULLMAN (J.D.) - The design and analysis of computer algorithms. - Addison-Wesly, 1974.

(2) BORODIN (A.), MUNRO (I.) - The computational complexity of algebraic and numeric problems. - Elsevier Computing Science Library, 1975.

(3) GAREY, JOHNSON (D.S.) - Computers and intractability. - W.H. Freeman and C°, 1979.

(4) KNUTH (D.E.) - The Art of computer programming. - Addison-Wesley, 1968, 1969, 1973. Vol. 1, 2, 3.

Références des quelques articles cités dans l'exposé :

(5) CHVATAL - Determining the stability number of a graph. - S.I.A.M.,J. Comput. 6, 1977, p. 643-662.

(6) COHEN (J.) - A langage for inquiring about the run-time behaviour of programs. - Software, 7, 1977, p. 445-460.

(7) COHEN (J.), ROTH (M.) - On the implementation of Strassen's fast multiplication algorithm. - Acta Informatica, 6, 1978,p. 341-355.

(8) COHEN (J.), ZUCKERMAN (C.) - Two languages for estimating program efficiency. - C.A.C.M., 17, 6, 1974, p. 301-308.

(9) DIARMID (Mc) - Determining the chromatic number of a graph. - S.I.A.M., J. Comput, 1, 1979, p. 1-14.

(10) KARP (R.M.) - The probabilistic analysis of some combinatorial search algorithms, in J.F. Traub (Ed), Algorithms and Complexity, Academic Press, 1976

(11) KNUTH (D.E.) - The analysis of algorithms. - Actes, Congrès Inter. Math., 1970, t.3, p. 269-274.

(12) KNUTH (D.E.) - An empirical study of fortran programs. - Software,1, 1971, p. 105-133.

(13) KNUTH (D.E.) - Big micron and big omega and big theta. - Sigact news, Apr.-June 1976, p. 18-24.

(14) KNUTH (D.E.), MOORE (R.W.) - An analysis of Alpha-Beta pruning. - Artificial intelligence, 6, 1975, p. 293-326.

(15) RIVEST (R.L.), VUILLEMIN (J.) - On recognizing graph properties from adjacency matrices. - Theor. Comput. Sc., 3, 1976, p. 371-384.

(16) PIWOWARSKI (P.) - A nesting level complexity measure. - SIGPLAN, Notices, 17, 9, 1982,p. 44-50

(17) YAO (Chi-Chih) - A lower bound to finding convex hulls. - J.A.C.M., 28, 4, 1981, p. 780-787.