Théorie de graphes

1 Objectif de cet ouvrage

De nombreux livres sur la théorie des graphes et les algorithmes sur les graphes ont déjà été écrits dans le monde, dont certains sont bons et certains sont merveilleux. Cependant, il y a un problème auquel les auteurs de ce livre ont été confrontés : les auteurs-mathématiciens se plongent souvent dans des formulations mathématiquement précises et vérifiées, conduisant loin des graphes appliqués à la beauté purement scientifique. Au contraire, les auteurs programmeurs négligent souvent les mathématiques et réduisent leurs livres à un «catalogue d'algorithmes». Nous n'aimons vraiment ni l'une ni l'autre approche. Notre objectif était d'écrire une sorte de manuel, montrant à la fois la beauté mathématique de la théorie des graphes, y compris le niveau de concours, et sa nature algorithmique, qui peut aider en informatique.

Selon le niveau de difficulté, ce livre s'adresse aux étudiants intéressés à participer aux concours en mathématiques et en informatique, ainsi qu'à leurs professeurs. La plupart des sujets n'impliquent pas que le lecteur possède des connaissances suffisantes dans un appareil mathématique spécifique. Tous les sujets sont présentés à partir de zéro. Cependant, afin de comprendre les chapitres algorithmiques, il est toujours souhaitable de pouvoir au moins lire les algorithmes écrits pour vous.

Dans de nombreux questions de la recherche et de pratique, le problème pour la plupart des gens est qu'ils compliquent souvent trop la solution. Il est important de comprendre comment résoudre les problèmes avec des méthodes simples et gracieuses. Le livre est divisé en plusieurs chapitres, unies par une idée commune. Vous pouvez voir la répartition détaillée en rubriques dans la table des matières. Chaque chapitre, à son tour, est divisée en plusieurs sections, dont dans la première partie discute la théorie et des solutions détaillées de plusieurs problèmes types sur ce sujet.

La deuxième partie des sections est constituée de quelques tâches pour une solution indépendante ou des questions de réflexion, pensées pour consolider le sujet énoncé. Certains d'entre elles apparaîtront dans la littérature pour la première fois. Leurs solutions sont données à la fin du chapitre.

Ce cours diffère sensiblement des cours traditionnels enseignés dans l'enseignement supérieur à deux égards :

1. Les cours traditionnels sur les algorithmes et les structures de données sont généralement assez académiques. La plupart d'entre eux sont basés sur l'excellent livre de Cormen et al. **Cormen_2013**. Ni ce livre ni les cours académiques ne sont liés à un langage de programmation. Il n'est pas nécessaire d'écrire des programmes qui implémentent un algorithme, et les connaissances qui ne sont pas confirmées par la pratique de leur utilisation ont tendance à s'estomper.
2. Un autre groupe de cours est généralement axé sur l'étude détaillée d'un langage de programmation particulier, et les algorithmes qu'il contient ne sont considérés que comme une illustration des propriétés du langage.

La deuxième partie de ce livre s'adresse aux lecteurs qui ont déjà une connaissance de langages de programmation et qui ont déjà écrit quelques programmes. Son objectif est de créer une base de programmation et algorithmique solide, sur la base de laquelle il est possible de résoudre des problèmes algorithmiques assez complexes. Dans ce livre, chacun des algorithmes analysés est illustré par des exemples écrits en langages C et C++. Au cours de la lecture de cet ouvrage, les étudiants doivent résoudre plusieurs dizaines de problèmes, chacun illustrant soit des algorithmes purs, soit leur composition.

Le principe enseigné dans le cours est la nécessité d'appliquer le division du problème en sous-problèmes plus simples : la triade clas-

sique «analyse, décomposition, synthèse». Pour réussir la division d'une tâche en sous-tâches, il est nécessaire d'imaginer quelles sous-tâches peuvent être résolues en un temps limité et lesquelles ne le peuvent pas, quelles sous-tâches sont pratiques pour une décomposition ultérieure et quelles sous-tâches ne peuvent pas être décomposées davantage. Pour cela, l'étudiant crée un certain ensemble de sous-tâches typiques, leurs principales caractéristiques et propriétés sont fixées, ce qui est appelé *abstractions* dans le livre. L'expérience des auteurs montre qu'avec une approche correcte de la décomposition, même des problèmes complexes peuvent être réduits aux problèmes plus simples, raccourcissant ainsi leur développement et leur mise en œuvre [1].

Une preuve rigoureuse de l'exactitude de tous les algorithmes et de leur complexité est, bien sûr, nécessaire. Malheureusement, cette rigueur ne peut souvent être atteinte qu'en utilisant un appareil mathématique assez avancé. L'accent mis sur le côté pratique de ce cours ne nous permet pas de fournir toutes les preuves nécessaires, à ces fins, il existe des livres aussi merveilleux que **Knuth1** ; **Knuth2** ; **Knuth3** ; **Cormen_2013**. Cependant, il n'est en aucun cas possible d'abandonner complètement la compréhension de la manière dont l'exactitude et la complexité des algorithmes sont prouvées. Un développeur d'algorithmes qui n'est pas en mesure de décrire au moins en général les caractéristiques de l'algorithme développé est regrettable, et, en fait, est professionnellement inadapté à cette activité. Par conséquent, pour chaque algorithme considéré, nous évaluerons certainement, peut-être pas rigoureusement, sa complexité et sa justesse. S'appuyer sur des sous-problèmes déjà résolus peut être très utile là.

1. Par exemple, l'un des projets, dont le développement et la mise en œuvre étaient prévus sur trois ans, a été mis en exploitation commerciale en moins d'un an.

2 Que sont les graphes et pourquoi sont-ils nécessaires ?

Toute notre vie, nous sommes confrontés aux graphes.

— Cartes géographiques. Quel est le trajet le moins cher entre Paris et Londres ? Lequel prend le moins de temps ? Pour répondre à ces questions, des informations sont nécessaires sur les trajets entre les villes et les coûts de ces trajets.
— Microcircuits. Les transistors, les résistances et les condensateurs sont reliés entre eux par des conducteurs. Y a-t-il des courts-circuits dans le système ? Est-il possible de réarranger les composants pour qu'il n'y ait pas d'intersection de conducteurs ?
— Planifications des tâches. Une tâche ne peut pas être démarrée sans en résoudre d'autres, par conséquent, il existe des connexions entre les tâches. Comment établir un calendrier de résolution des problèmes afin que le processus total soit terminé dans les plus brefs délais ?
— Réseaux informatiques. Il y a des nœuds tels que terminaux, ordinateurs, tablettes, téléphones, commutateurs, routeurs… Chaque lien a des propriétés de latence et de bande passante. Quel est le chemin pour envoyer un message afin qu'il puisse être livré au destinataire dans les plus brefs délais ? Y a-t-il des « nœuds critiques » dans le réseau dont le dysfonctionnement mène à la division du réseau en composantes non connexes ?
— La structure du programme. Les nœuds sont des fonctions dans un programme. Connexions montrent si une fonction peut en appeler une autre (analyse statique) ou ce qu'elle appellera pendant l'exécution du programme (analyse dynamique). Pour savoir quelles ressources devront être allouées au système, un graphe d'exécution du programme est nécessaire.

Voici quelques tâches typiques liées aux graphes :

- Il y a un graphe. Est-il connexe ?
- Un graphe est-il un arbre ?
- Trouver le chemin le plus court du nœud u au nœud v.
- Trouver un cycle qui traverse toutes les arêtes exactement une fois (cycle eulérien).
- Trouver un cycle qui passe par tous les sommets exactement une fois (cycle hamiltonien).
- Contrôle de planarité : déterminer si un graphe peut être tracé sur un plan sans auto-intersections.

Évidemment, la liste des problèmes de graphes est vraiment longue, et nous ne pourrons pas étudier tous les algorithmes existants. Mais nous allons tout de même étudier un certain nombre d'algorithmes.

3 Algorithmes

Depuis l'école, tout le monde connaît la définition intuitive d'un algorithme : « un ensemble d'instructions décrivant l'ordre des actions de l'exécuteur pour parvenir à un certain résultat ». On considère que l'algorithme accepte un certain *objet constructif* en entrée, effectue une séquence d'actions et renvoie (s'il s'arrête) une nouvelle objet constructif. Un objet constructif est un objet pour lequel il existe un moyen constructif de le définir, par exemple, un nombre naturel, une matrice ou un graphe.

Ce concept d'algorithme est assez vague et ne permet pas de réponses précises à de telles questions :

- *Est-ce que tout problème peut être résolu algorithmiquement ?*
- *Si un problème ne peut pas être résolu par aucun algorithme,*

comment le prouver ? En d'autres termes, comment montrer que tout algorithme résout un problème autre que celui donné ?

— *Comment calculer le temps engagé par l'algorithme, ou la mémoire requise ?* Vous ne pouvez pas calculer le temps d'exécution d'un algorithme simplement en examinant une implémentation de celui-ci (par exemple, en C) et en additionnant le nombre de fois que chaque opération est exécutée. Les commandes ne sont pas unitaires ni en temps ni en mémoire requise, le temps d'exécution d'une même commande peut être différent. La meilleure chose à faire dans ce cas est d'estimer (si possible) le temps d'execution de l'operation (dans le pire des cas) d'en haut par une constante. Mais là, il est important de savoir quelles opérations nous considérons comme élémentaires et pour quelles opérations peut le temps être estimé d'en haut par une constante (en particulier, l'addition de nombres ne peut être considérée a priori comme une opération effectuée en temps constant, si nous ne considérons pas, par exemple, que ces nombres sont limités).

— *Comment montrer qu'un problème donné ne peut pas être résolu exactement en temps relativement court ?* L'existence d'un algorithme en soi ne garantit pas qu'un algorithme qui résout le même problème plus rapide ne peut pas être construit.

Des réponses claires à ces questions nécessitent une notion logique d'algorithme. Pour des raisons pratiques, une définition intuitive d'un algorithme est généralement suffisante ; à chaque fois nous préciserons ce qu'est une unité de mémoire et quelles opérations sont *élémentaires*, c'est-à-dire qu'elles sont exécutées en temps à-peu-près constant.

La plupart des algorithmes que nous allons traiter sont *déterministes*, c'est-à-dire que les étapes qu'ils effectuent, ainsi que le résultat qu'ils produisent, dépendent uniquement de l'entrée de l'algorithme. Cependant, dans certains cas, il est possible d'utiliser des procédures

probabilistes au lieu de procédures déterministes. Ceci est fait principalement afin de réduire le temps d'exécution de l'algorithme. Bien sûr, vous devez payer pour cela par la possibilité que la solution au problème s'avère inexacte ou que la réponse correcte est donnée avec une certaine probabilité. Il arrive souvent que vous deviez faire plus d'itérations, mais le coût d'une itération devient beaucoup «moins coutable». Il convient de noter que les algorithmes probabilistes sont importants non seulement d'un point de vue théorique, mais aussi d'un point de vue pratique, car certaines procédures probabilistes ont fait leurs preuves en pratique. Par exemple, certaines personnes engagées dans l'apprentissage en profondeur (Deep Learning) par les mots «descente de gradient» (quoi que cela signifie) comprennent souvent sa version stochastique, en raison de sa grande valeur pratique. Il est donc utile de s'habituer littéralement au langage de la théorie des probabilités dès les premières étapes, et lors du développement et de l'analyse d'algorithmes, le hasard doit être traité comme une ressource potentiellement importante. Mais pour se rendre compte de ce qu'est, en fait, cette ressource, il n'est possible qu'en résolvant des problèmes.

Comment estimer le temps des procédures probabilistes ? Il existe 2 approches principales.

1. On dit que l'algorithme fonctionne le temps $T(n)$ sur une entrée de taille n *dans le pire des cas*, si à n'importe quelle entrée de taille n cela fonctionne le temps $\leqslant T(n)$, et l'égalité est atteinte.
2. On dit que l'algorithme fonctionne le temps $T(n)$ sur une entrée de taille n *en moyenne*, si l'espérance mathématique de son temps de fonctionnement à une entrée de taille n est égale à $T(n)$.

Considérons le problème suivant. L'entrée est un tableau de $2N$ lettres, et la moitié d'entre elles sont des lettres a et l'autre moitié sont des lettres b. Il est nécessaire de trouver le numéro de quelque cellule contenant la lettre a. Considérons 2 idéologies de solution

probabiliste d'un tel problème.

Un algorithme de Las Vegas donne toujours un résultat exact, mais le temps de calcul est petit avec une très forte probabilité. Considérons l'algorithme suivant comme exemple.

Require : cc
1 : **repeat**
2 : Choisir aléatoirement l'indice de manière équiprobable $i \in \{1,2,\ldots,N\}$
3 : **until** $A[i] \neq a$
4 : **return** i

Le temps d'exécution le plus défavorable de cet algorithme est l'infini. En effet, il existe une probabilité non nulle qu'à chaque étape l'algorithme choisisse une cellule contenant la lettre b, ce qui signifie qu'on ne peut pas limiter le nombre d'itérations. Cependant, l'algorithme s'arrête en un nombre fini d'étapes avec probabilité 1. De plus, le nombre moyen d'itérations est

$$A \stackrel{\text{def}}{=} \sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=1}^{\infty} \frac{i+1}{2^i} - \sum_{i=1}^{\infty} \frac{1}{2^i} =$$
$$2\left(\sum_{i=1}^{\infty} \frac{i}{2^i} - \frac{1}{2}\right) - 1 = 2\left(A - \frac{1}{2}\right) - 1 = 2A - 2 \Rightarrow A = 2,$$

c'est-à-dire l'algorithme *fait 2 itérations en moyenne.*

Un algorithme de Monte-Carlo peut donner une réponse approchée dans un certain intervalle de confiance. Dans ce cas, on cherche à avoir un petit intervalle de confiance tout en conservant une complexité en temps faible, par exemple polynomiale en la taille de l'entrée. Considérons l'algorithme suivant comme exemple.

Le temps d'exécution dans le pire des cas de cet algorithme est k. La probabilité d'obtenir la bonne réponse est égale à $1 - \frac{1}{2^k}$. De plus, le

Require : Tableau de lettres A de taille N ; nombre entier positive k

1 : $i := 0$
2 : **repeat**
3 : Choisir aléatoirement l'indice de manière équiprobable $i \in \{1,2,\ldots,N\}$
4 : **until** $k = i$ ou $A[i] \neq a$
5 : **return** i

nombre moyen d'itérations effectuées est

$$B \stackrel{\text{def}}{=} \sum_{i=1}^{k} \frac{i}{2^i} = \sum_{i=1}^{k} \frac{i+1}{2^i} - \sum_{i=1}^{k} \frac{1}{2^i} = 2\left(\sum_{i=1}^{k} \frac{i}{2^i} - \frac{1}{2} - \frac{k}{2^k}\right) - 1 = \\ = 2\left(B - \frac{1}{2} + \frac{k+1}{2^{k+1}}\right) - 1,$$

Par conséquent,

$$B = 2 - \frac{k+1}{2^{k+1}},$$

l'algorithme *fait en moyenne moins de 2 itérations.*

Table des matières