Bruno Courcelle

Structuration des graphes et logique

Les deux mots clefs de mon exposé seront la *logique* et la *structuration des graphes*. Les recherches concernant les relations entre ces deux sujets ont débuté indépendamment dans quatre directions différentes et se sont révélées de plus en plus intimement liées (je vais essayer de montrer comment elles se relient). Je vais présenter ces quatre directions de recherche dans l'ordre suivant : extension aux graphes de la théorie des langages formels ; algorithmes polynomiaux pour des problèmes NP-complets ; théorie des mineurs de graphes et enfin décidabilité de la logique du second ordre monadique.

Suivre l'ordre historique inciterait plutôt à commencer par les algorithmes polynomiaux pour des problèmes NP-complets. Des recherches débutant dans les années 1980 se sont attachées à la notion de *structuration des graphes*, laquelle permet de construire des algorithmes polynomiaux pour des problèmes NP-complets restreints à des classes particulières de graphes. C'est le premier thème qui a mobilisé les chercheurs dans ce domaine. Le deuxième est celui de la théorie des mineurs de graphes, développée par Robertson et Seymour. Il fait appel aux mêmes notions de structuration. Le troisième thème concerne la décidabilité des théories logiques. Il consiste à faire le lien entre la structure des graphes, leur proximité avec le fait d'être des arbres et le rapport entre cette proximité et la décidabilité de certaines théories logiques. Le quatrième thème est l'extension aux graphes de la *théorie des langages formels* qui traite des descriptions d'ensembles de mots et d'arbres par des grammaires et des automates.

Extension aux graphes de la théorie des langages formels

Trois notions importantes en théorie des langages vont trouver, d'une manière ou d'une autre, des extensions dans le cas des graphes. Tout d'abord la notion de *grammaire* dont l'extension aux graphes est

assez facile, ensuite la notion d'*automate* et la notion de *transducteur* (c'est-à-dire de transformation de mots et d'arbres au moyen d'automates). Dans ces deux derniers cas, l'extension aux graphes n'est pas du tout immédiate, mais elle peut se faire par le détour de la logique du second ordre monadique.

Grammaires

Commençons par les grammaires. Les langues naturelles sont couramment décrites au moyen de grammaires qui définissent des catégories grammaticales. Les phrases se décomposent en groupes-sujet, verbes, attributs, déterminants, substantifs etc. Ce type de structuration est important pour la vérification syntaxique, la traduction automatique, la compréhension automatique de textes. C'est au moyen de grammaires que l'on décrit les langues naturelles et les langages de programmation, avec en particulier la nécessité cruciale pour ces derniers d'une traduction rigoureuse puisqu'un programme n'est utilisable par l'ordinateur qu'après traduction dans son langage.

La *syntaxe abstraite* est la description arborescente des objets. Prenons une expression arithmétique $x \times y + z$, avec son analyse correcte. (Une mauvaise analyse conduirait à faire d'abord la somme et ensuite le produit.) Cette expression peut être écrite de manière non ambiguë avec des parenthèses, des virgules et des opérateurs placés avant les arguments : $+(\times(x,y),z)$, ou bien au moyen de la notation dite *polonaise préfixée* : $+ \times xyz$ (elle a pour avantage d'être concise et non ambiguë mais pour inconvénient d'être peu lisible).

Dans tous les cas, une structure est commune à ces trois notations, c'est l'arborescence qui décrit la signification du terme considéré (fig. 1) et qui sert de base à la traduction que doit faire le compilateur.

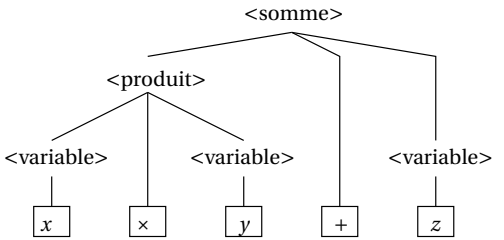


Fig. 1. Arborescence sous-jacente à l'expression algébrique $x \times y + z$

La notion de grammaire s'appuie sur l'arborescence qui structure un objet, ici un mot, et fournit le support d'un calcul. Dans le cas d'un processus de compilation, ces arborescences sont utilisées pour effectuer des calculs, par exemple des calculs *montants* qui permettent les vérifications de type : dans un langage de programmation les opérations telles que l'addition et la multiplication sont utilisables dans des expressions de types différents (entier, réel, complexe...) mais le compilateur doit distinguer les types des expressions pour les traduire correctement ; à partir de l'arbre de l'expression algébrique, et en effectuant un calcul montant dans cet arbre, il peut déterminer le type du résultat et donc traduire correctement l'expression. D'autres calculs se font en descendant dans l'arbre lorsqu'il est nécessaire d'utiliser des informations contextuelles. L'arbre est dans tous les cas le support d'un calcul.

J'en viens aux grammaires telles que vous pouvez les trouver dans les manuels des langages de programmation. Un terme peut être, soit un opérateur suivi de deux termes entre parenthèses séparés par une virgule, soit une variable, soit une constante.

<terme> ::= <opérateur>(<terme>, <terme>)

<terme> ::= <variable>

<terme> ::= <constante>

Les symboles <opérateur>, <variable> et <constante> renvoient à des descriptions d'opérateurs, de variables et de constantes qui sont données par ailleurs par d'autres équations de ce type. De manière plus synthétique, on pourra écrire :

$$\text{<terme>} ::= f(\text{<terme>}, \text{<terme>}) \mid a \mid b$$

Dans ce cas, un terme peut être, soit la constante a, soit la constante b, soit l'opérateur f avec deux arguments entre parenthèses séparés par une virgule qui sont aussi des termes. Cette écriture est plus concise. Avec les notations de la théorie des langages formels, on aboutit à cette écriture :

$$\mathrm{L} = f(\mathrm{L}, \mathrm{L}) \cup \{a\} \cup \{b\},$$

qui s'interprète de la façon suivante : L est un ensemble de mots, cet ensemble est formé des mots a et b (chaque lettre, a ou b, est un mot) et de mots qui sont formés par combinaison de l'opérateur f, de parenthèses, de virgules et de mots de L déjà construits. Cette équation a une *plus petite solution* (un plus petit ensemble pour l'inclusion qui

satisfait cette égalité) dans l'ensemble des langages dont les mots sont écrits avec les lettres a, b, f, (,), ,.

Cette approche grammaticale a l'intérêt de s'étendre très facilement à d'autres objets que les mots. Dès que des opérations sont définies sur les objets considérés (opérations qui généralisent la concaténation des mots), on peut écrire des équations comme ci-dessus et définir des ensembles d'objets engendrés à partir d'objets de base. La figure 2 montre un graphe obtenu à partir de l'équation qui définit les *cographes*.

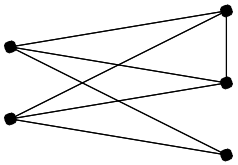


Fig. 2. Graphe obtenu à partir de l'équation : $L = L \oplus L \cup L \otimes L \cup 1$

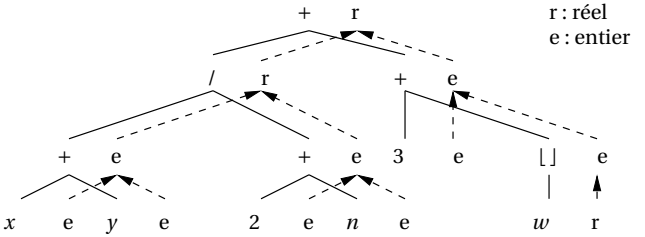
Un graphe à un sommet est noté 1. L'opération qui à partir de deux graphes en fait la somme disjointe est notée $\oplus$ (on prend des copies disjointes) et l'opération notée comme un produit ($\otimes$) consiste à prendre deux graphes (disjoints) et à les réunir par toutes les arêtes possibles entre l'un et l'autre. Avec ce type d'écriture, on spécifie un ensemble de graphes dont celui de la figure 2. Les cographes sont construits au moyen d'un nombre fini d'applications des opérations $\oplus$ et $\otimes$.

Automates

Parmi les notions importantes en théorie des langages formels, il y a ensuite la notion d'*automate qui opère sur les mots ou sur les termes*. Je vais commencer par présenter les automates finis qui opèrent sur les termes, et qui correspondent à des calculs inductifs montants.

Prenons l'exemple d'un *terme algébrique* (dit *expression arithmétique* en programmation) que l'on veut calculer, et pour lequel on cherche également le *type* (réel ou entier). Les différentes opérations sont l'addition, la division et l'arrondi à la partie entière inférieure. Les règles de base disent comment se propagent les types des objets : la somme de deux entiers est un entier, la somme d'un réel et d'un entier

ou d'un réel est un réel, etc. Ces règles de base sont utilisées itérativement sur l'arbre syntaxique de l'expression pour déterminer son type.



Exemple de règles de calcul locales en un nœud :

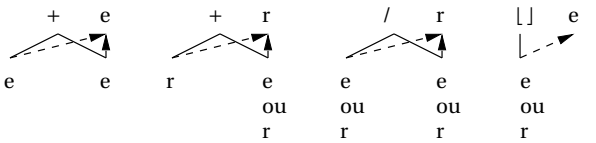


Fig. 3. Exemple de calcul montant pour déterminer le type d'une expression

L'expression arithmétique de la figure 3 est mise sous forme d'un arbre sur lequel un calcul montant fournit le type de l'expression complète, lequel dépend des types des variables $x, y, z\ldots$ fixés par les déclarations. Je me contenterai de cet exemple d'automate sur les termes. J'aurai besoin par la suite d'automates semblables opérant sur des termes représentant des graphes.

Transductions

Pour ce qui est de la notion de traducteur, je me contenterai d'un exemple concernant les automates traducteurs de mots. Je prends l'exemple d'un décodage. Quatre lettres a, b, c et d sont codées au moyen de 0 et 1. Le décodage est défini par le tableau ci-dessous :

$$\begin{aligned} 00 &\to a \\ 01 &\to b \\ 10 &\to c \\ 11 &\to d \end{aligned}$$

La notion d'automate formalise un calcul consistant à parcourir de

gauche à droite un mot donné sur 0 et 1. À chaque étape, un dispositif mémorise un *état* et l'utilise pour faire un calcul. Ces calculs aboutissent à écrire sur un fichier de sortie les lettres obtenues. Un 0, à lui seul, ne détermine pas si la lettre est un a ou un b et le 1 suivant indique que c'est un b, on écrit alors ce b et on continue.

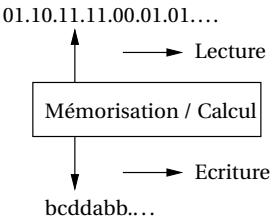


Fig. 4. Exemple d'automate traducteur de mots

C'est l'exemple typique d'une *transduction* définie par un automate traducteur. Comment peut-on étendre cette notion aux graphes ? Aussi bien pour les automates qui opèrent sur les arbres que pour ceux qui opèrent sur les mots, il y a une notion de sens de parcours. Autrement dit, l'objet est parcouru dans un certain sens, et on ne repasse pas plusieurs fois sur les mêmes occurrences de lettres. Pour un graphe, il faudrait qu'un sens de parcours soit disponible. La difficulté pour les graphes est qu'il n'y a pas de structuration naturelle, ni linéaire ni arborescente, et donc pas de parcours « naturel ».

Composition de graphes, largeur arborescente

Après cette longue introduction, entrons un petit peu dans le détail des définitions et regardons quelques opérations de composition de graphes et les grammaires associées. Cette idée de considérer les graphes comme des objets et de les combiner afin d'effectuer des calculs remonte à des études de la fin du XIX^{e} siècle relatives aux circuits électriques. Les premières opérations qui ont été introduites dans ce but ont été les opérations de *mise en parallèle* et de *mise en série*, appellations classiques issues de cette origine. Ces opérations concernent des graphes, avec des sommets particuliers, qui sont des points d'entrée et de sortie que j'appellerai des *sources*. L'opération de mise en parallèle recolle deux graphes G et H en identifiant les deux entrées et les

deux sorties. Dans le cas de la mise en série, la sortie de G est identifiée avec l'entrée de H (fig. 5).

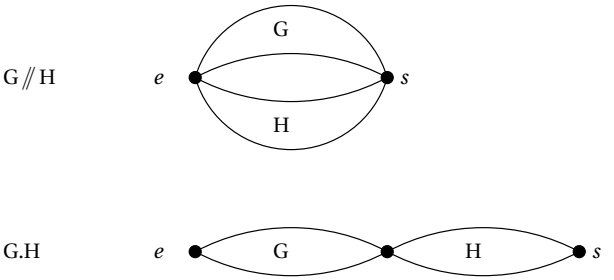


Fig. 5. Opérations de mise en parallèle et de mise en série de deux graphes

Les graphes que l'on peut fabriquer à partir de ces deux opérations sont de structure assez limitée. Le remède consiste en l'extension des notions d'entrée et de sortie : on oublie l'idée de circulation d'information de l'entrée vers la sortie et on se donne des sommets de recollement nommés *sources* et désignés par des étiquettes en nombre fini pour chaque classe de graphes considérée, mais non borné.

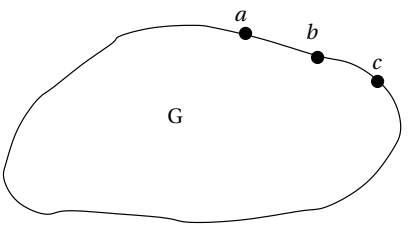


Fig. 6. Graphe muni de ses points de recollement

Dans l'exemple de la figure 6, trois lettres *a*, *b* et *c* désignent trois sources. « L'intérieur » du graphe peut contenir d'autres choses. De « l'extérieur », on ne connaît de ce graphe que les trois sources. Les sources permettent de définir des opérations dont l'une généralise simultanément la mise en parallèle et la mise en série.

Soient par exemple deux graphes, G muni de quatre sources *a*,

b, *c* et *d*, et H muni de quatre sources *b*, *c*, *d* et *e*. L'opération de recollement, notée G // H, nommée *composition parallèle* se fait en identifiant le sommet *b* de G avec le sommet *b* de H, le sommet *c* de G avec le sommet *c* de H, et enfin les sommets *d* de chacun des graphes. À partir de deux graphes avec quatre sources, nous obtenons un graphe avec cinq sources (fig. 7).

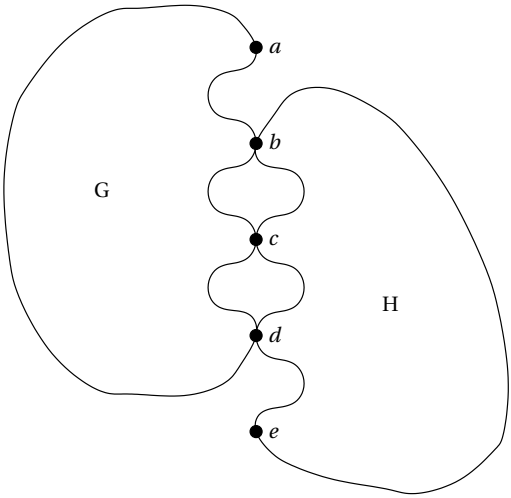


Fig. 7. Graphes G et H recollés

Outre cette opération de recollement, on utilisera des opérations qui suppriment les étiquettes de source. L'effet de l'opération Suppr_a est de rendre « ordinaire » le sommet source auparavant étiqueté par *a*. Ce sommet n'est plus accessible de l'extérieur ; on ne peut plus faire sur lui d'opération de recollement. La dernière opération est le renommage des sources qui consiste à remplacer une étiquette *a* par une étiquette *b*.

On peut ainsi définir des familles de graphes nettement plus substantielles qu'avec uniquement les mises en parallèle et en série. On peut utiliser une infinité d'étiquettes. Tout graphe peut être construit, mais il est important pour les applications algorithmiques d'utiliser un

nombre fini d'opérations décrites au moyen d'un ensemble fini d'étiquettes.

Prenons ainsi $D = \{a_1, a_2, \ldots, a_k\}$. Des graphes peuvent être définis par la seule équation :

$$L = L /\!\!/ L \cup \quad (1)$$

$$\mathrm{Suppr}_{a_1}(L) \cup \mathrm{Suppr}_{a_2}(L) \cup \cdots \cup \mathrm{Suppr}_{a_k}(L) \cup \quad (2)$$

$$\mathrm{Ren}_{h_1} \cup \mathrm{Ren}_{h_2} \cup \cdots \cup \mathrm{Ren}_{h_p} \cup \quad (3)$$

$$\{a_1 \to a_2\} \quad (4)$$

où $h_1, ..., h_p$ sont les permutations de D. Cette équation s'interprète de la façon suivante :

(1) deux graphes de la famille L peuvent être mis en parallèle ;
(2) une étiquette de source peut être supprimée ;
(3) les sources peuvent être renommées ;
(4) on prend des graphes de base constitués d'un arc allant d'un sommet étiqueté a_1 vers un sommet étiqueté a_2.

Grâce aux possibilités de renommage, l'arc de a_1 vers a_2 suffit car a_1 peut être changé en b et a_2 en c, pour toutes étiquettes b et c. On dispose donc d'arcs avec toutes les étiquettes possibles et on peut les recoller un par un pour former un graphe. Avec un nombre limité d'étiquettes, certains graphes ne peuvent pas être obtenus. On ne peut fabriquer que des graphes de *largeur arborescente* inférieure à la cardinalité de l'ensemble D.

La notion de largeur arborescente a été introduite simultanément par différents auteurs ayant des préoccupations différentes : construction d'algorithmes polynomiaux sur des classes particulières de graphes, définition de grammaires de graphes, théorie des mineurs de graphes et décidabilité des théories logiques. Elle peut être définie de façon algébrique à partir des opérations sur les graphes, mais également de façon combinatoire. La notion de *k-arbre partiel* a été tout d'abord introduite et caractérise les graphes simples non orientés de largeur arborescente au plus k. Le terme « largeur arborescente », en anglais *tree-width*, a été introduit par Robertson et Seymour [8] qui, je pense, ont redécouvert la notion. En termes de graphes l'intuition est la suivante : un graphe est obtenu par un recollement arborescent de graphes plus petits et qui ont tous au plus $k+1$ sommets.

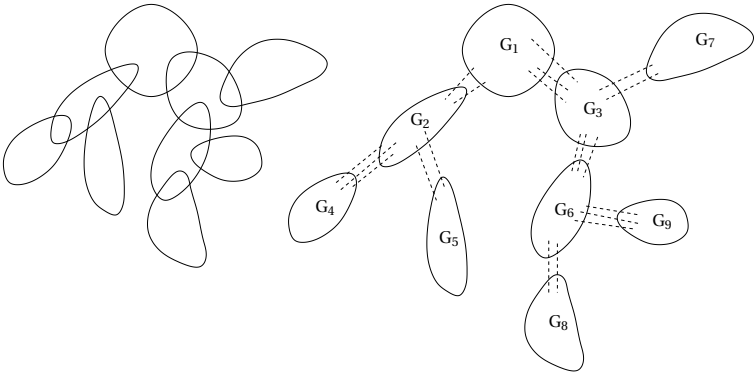


Fig. 8. Un graphe obtenu par recollement arborescent de « petits » graphes.

Une façon parlante d'illustrer ce recollement arborescent consiste à séparer les différentes composantes qui se recouvrent dans la partie gauche de la figure 8. Dans la partie droite, les composantes sont séparées par des arcs en pointillés qui ont vocation à être contractés. Cela fournit une présentation claire de l'arborescence sous-jacente à la décomposition.

La figure 9 illustre ce type de construction avec un graphe de largeur arborescente au plus trois. Une décomposition est *optimale* si elle minimise la taille de la plus grosse boîte. La décomposition de la figure 9 n'est pas optimale.

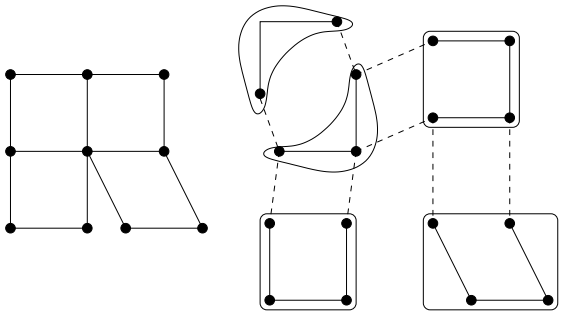


Fig. 9. Un exemple de décomposition arborescente.

Pourquoi la notion de largeur arborescente a-t-elle un intérêt algorithmique ? Prenons un graphe éventuellement très compliqué. On va d'abord le structurer de façon arborescente, donc le décomposer en « petits graphes ». Ces composantes sont organisées en un arbre, où $G_1, G_2, \ldots$ sont des « petits » sous-graphes comme le montre la figure 10 correspondant au graphe de la figure 8.

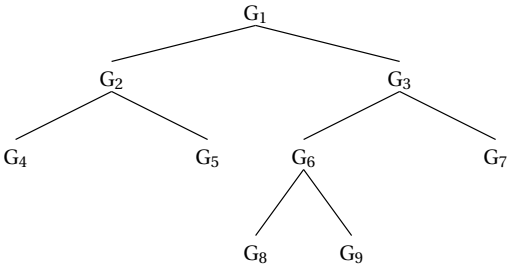


Fig. 10. Arborescence obtenue à partir du graphe de la figure 8

Nous pouvons utiliser l'arbre de la décomposition comme support d'un calcul relatif au graphe considéré. On peut dans certains cas tester une propriété du graphe global, par exemple une propriété de colorabilité, en commençant par la tester sur les graphes G_4, G_5, G_8, G_9 et G_7 qui sont aux feuilles de l'arbre. Les informations relatives aux possibilités de colorier G_8 et G_9 permettent d'obtenir des informations sur les coloriages de G_6, on en déduit des informations analogues sur G_3, etc. On obtient en fin de calcul (à la racine de l'arbre) l'information désirée relative au graphe considéré. On a utilisé pour ce faire un calcul montant basé sur l'arbre de la décomposition.

Grammaires de graphes

Revenons sur la présentation des grammaires comme systèmes d'équations. Les grammaires de mots sont habituellement représentées comme des *systèmes de réécriture,* c'est-à-dire des méthodes pour fabriquer des mots.

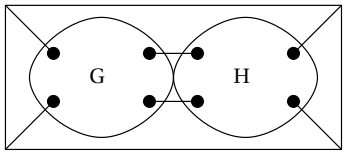
$$\begin{aligned} \mathrm{T} &\to a \\ \mathrm{S} &\to a\mathrm{ST} \\ \mathrm{S} &\to b \\ \mathrm{T} &\to c\mathrm{TTT} \end{aligned}$$

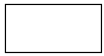
La première règle dit qu'à partir de T, on obtient le mot a, la deuxième règle dit qu'à partir de S on peut construire un mot défini comme a suivi de mots définis par S et T, etc. La présentation sous forme de système d'équations donnerait pour cette grammaire :

$$S = aST \cup b\,;\ T = a \cup cTTT$$

La notion de *système d'équations* a l'avantage sur les systèmes de réécriture de s'appliquer à des situations variées, dès que l'on a des opérations, ce qui dans notre cas, fournit une certaine classe de grammaires sur les graphes.

Grammaire : $S = f(S,S) \cup a$

$f(G,H) =$ 

$a =$ 

Exemple de graphe obtenu par cette grammaire :

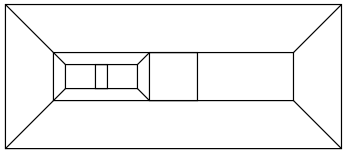


Fig. 11. Une grammaire de dessins de graphes

Les graphes comme celui de la figure 11 ont une structure emboîtée assez visible. Le graphe de base est un rectangle et l'opération de combinaison de deux graphes, notée f, consiste à insérer dans les deux places libres d'un cadre des graphes déjà construits. Ce sont des graphes avec quatre points de recollement. On peut en coller un à la place de G et un à la place de H. Les recollements appropriés peuvent être définis formellement avec des étiquetages.

La notion de grammaire, définie au départ pour les mots, permet de définir des ensembles de graphes et d'autres objets combinatoires

comme les graphes dessinés. Chaque objet défini par une grammaire est caractérisé par une structuration qui est l'historique de sa construction par applications successives des règles de la grammaire. Cette structuration est un arbre de syntaxe abstraite et elle représente la « signification » de cet objet. Elle peut être utilisée comme support d'une traduction, d'un dessin, d'une preuve par induction sur la profondeur de l'arbre ou d'algorithmes basés sur des parcours d'arbre.

Enfin, un dernier avantage est que l'on peut décrire les graphes engendrés par des termes, autrement dit des mots. Cet avantage est tout de même un peu théorique, car si un graphe est très compliqué, le terme qui le désignera a peu de chances d'être lisible.

D'autre part, l'analyse syntaxique pour une grammaire de graphes est un problème difficile, et dans certains cas NP-complet (cette notion est présentée plus loin), alors que pour les grammaires algébriques qui définissent des mots, l'analyse syntaxique se fait toujours en temps polynomial.

Deuxième difficulté, l'ensemble de tous les graphes finis n'est pas définissable par une seule grammaire. Alors que tous les mots, sur un alphabet donné, peuvent être définis par une unique grammaire, pour les graphes, nous sommes obligés de définir une hiérarchie fondée sur la largeur arborescente. En effet, une grammaire peut être construite pour les graphes de largeur arborescente au plus k. Mais dès qu'on a besoin de graphes qui sont de largeur arborescente plus grande, il faut changer de grammaire. Ce sont les deux difficultés principales de la notion de grammaire de graphes. Ces questions sont développées dans [2, 3].

Algorithmes polynomiaux pour des problèmes NP-complets

Algorithmes polynomiaux ; problèmes NP et NP-complets

La distinction entre problèmes polynomiaux et NP-complets relève de la *théorie de la complexité*. Le terme « complexité » est très mal choisi. Dans le jargon informaticien il veut dire « long à calculer ». Un programme très simple, c'est-à-dire construit d'un seul module mais qui nécessite un temps de calcul très long sera dit de complexité élevée, alors que par construction il n'est pas « complexe ». Cette terminologie très largement adoptée confond « complexité » et « difficulté ».

Un problème est dit *polynomial* s'il peut être programmé sur une *machine de Turing* et fournit une réponse correcte en un temps polynomial en la taille de la donnée. Une machine de Turing, c'est une sorte d'automate plus compliqué que ceux que j'ai montrés précédemment qui constitue le modèle de calcul de référence pour les questions de calculabilité et de complexité.

La notion de calculabilité en temps polynomial est *robuste* : si nous voulons appliquer cette notion à un problème de graphe, une première étape consiste à coder le graphe sous la forme d'une suite de caractères. Un graphe peut être représenté de plusieurs façons : comme une matrice carrée de 0 et de 1 qui indique pour chaque couple de sommets l'absence ou l'existence d'un arc entre eux ou bien comme une liste de paires de sommets. La notion de temps polynomial est invariante par rapport au choix de l'une ou l'autre de ces représentations.

Un problème est de *type NP* s'il peut être résolu en temps polynomial par une machine non déterministe. C'est un problème pour lequel la réponse est oui, s'il existe une affectation de valeurs à certaines variables qui satisfait une condition vérifiable en temps polynomial. Un problème de type NP comporte deux aspects, la nécessité de « deviner quelque chose » et la vérification qu'un candidat « deviné » est bon ou mauvais, ce qui se fait « facilement » (en temps polynomial). La difficulté est que le nombre d'essais à effectuer avant de trouver le bon candidat n'est pas borné par une fonction polynomiale en la taille des données.

Prenons l'exemple du 3-coloriage des graphes. Il s'agit de colorier les sommets d'un graphe avec trois couleurs de telle sorte que deux sommets adjacents ont des couleurs différentes. Un 3-coloriage candidat est une affectation des « couleurs » 1, 2 ou 3 à chacun des n sommets. Il reste ensuite à vérifier que deux sommets voisins ont des couleurs différentes. Si ce n'est pas le cas il faut faire un autre essai pour une autre affectation de couleurs aux sommets. Vérifier que les sommets adjacents ont des couleurs différentes se fait en temps polynomial, mais il y a 3^n candidats potentiels, donc au pire, si on ne tombe pas rapidement sur le bon 3-coloriage, on est obligé de tester 3^n possibilités. Si le graphe n'est pas 3 coloriable, on doit faire 3^n vérifications pour s'en assurer.

Une conjecture, classée comme l'un des 7 problèmes de mathématiques les plus importants (cf. [4]), consiste à prouver que $\mathrm{P} \neq \mathrm{NP}$: autrement dit, qu'il existe des problèmes qui sont de type NP mais qui ne

sont pas de type polynomial. La plupart des chercheurs sont convaincus que c'est vrai. Il y a eu tellement de travaux sur les problèmes de type NP que s'il y avait une solution polynomiale pour l'un d'entre eux, cela aurait certainement été trouvé. Mais la conjecture reste à prouver.

Parmi ces problèmes NP, il y a les *problèmes NP-complets* qui sont les plus difficiles. Ils sont représentatifs de la classe toute entière en ce sens que si un quelconque d'entre eux a un algorithme polynomial, tous les autres en ont aussi parce que ces différents problèmes NP-complets sont étroitement reliés les uns aux autres par des transformations d'algorithmes.

Comme problèmes NP-complets concernant les graphes, nous pouvons citer le 3-coloriage (qui est NP-complet même pour les graphes planaires) ; l'existence d'un circuit hamiltonien (un circuit qui parcourt tous les sommets du graphe sans passer deux fois par un même sommet) : ce problème est lié à celui du voyageur de commerce qui consiste à visiter un ensemble de villes en minimisant la distance totale parcourue. On peut également citer la recherche d'un sous-graphe complet de taille au moins k lorsque k fait partie de la donnée du problème. Pour chacun de ces problèmes, des algorithmes polynomiaux existent pour des classes particulières de graphes. Pour les graphes de largeur arborescente au plus k certains problèmes NP-complets deviennent résolubles en temps polynomial.

Problèmes restreints à des graphes de largeur arborescente bornée ; exemple du 3-coloriage

La problématique peut se formuler ainsi. Soient $\mathcal{G}$ une famille de graphes, F un ensemble fini d'opérations sur ces graphes et T(F) l'ensemble des termes sur F. À partir du terme t qui définit un graphe G de $\mathcal{G}$, la question qui se pose est : « Peut-on obtenir un résultat relatif au graphe G en travaillant uniquement sur le terme t ? » L'idée est de court-circuiter l'étape qui consisterait à construire le graphe G. Il est plus efficace de travailler directement sur le terme en utilisant un automate qui opère en un temps linéaire par rapport à la taille du terme.

Soient P une propriété de graphe et t un terme décrivant un graphe G. Supposons G décrit comme composition parallèle de deux graphes G_1 et G_2. La question cruciale est celle-ci : « Peut-on déduire la propriété P(G) à partir de la connaissance de $P(G_1)$ et de $P(G_2)$? » Prenons une analogie numérique. Supposons que l'on ait une expression compliquée sur les entiers qui utilise les opérations habituelles d'addition

et de multiplication. On veut savoir si le résultat est pair. Il n'y a pas besoin de calculer le résultat pour savoir s'il est pair. On peut déduire des sous-expressions la parité du résultat parce que la parité d'une somme ou d'un produit se déduit de la parité des deux arguments. Le principe est le même dans le cas des graphes. Prenons par exemple la propriété qu'un graphe est 2-coloriable et regardons comment elle se comporte par rapport à la mise en parallèle. Soient deux graphes G et H. Est-ce que $G /\!\!/ H$ est 2-coloriable ? Il est clair que si $G /\!\!/ H$ est 2-coloriable, alors G et H sont 2-coloriables, G et H étant des sous-graphes. Mais la réciproque est fausse, comme le montre l'exemple de la figure 12.

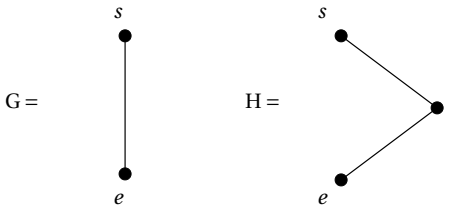


Fig. 12. Exemple de graphes 2-coloriables qui une fois mis en parallèle forment un graphe qui ne l'est pas

Chacun de ces graphes est 2-coloriable, mais une fois mis en parallèle, ils forment un graphe qui n'est pas 2-coloriable parce que G et H sont 2-coloriables mais par des 2-coloriages qui sont *incompatibles au niveau des sources (les sommets de recollement)*. En effet, le 2-coloriage de G attribue aux deux extrémités des couleurs différentes alors que les deux extrémités de H sont nécessairement de la même couleur. Le remède consiste à renforcer l'hypothèse de récurrence en utilisant des propriétés auxiliaires.

Montrons que le 3-coloriage des graphes de largeur arborescente au plus k est faisable en temps linéaire à condition que le graphe soit donné par une décomposition arborescente de largeur bornée. Soit G un graphe construit avec les opérations de composition parallèle, de renommage et d'oubli de source au moyen de k étiquettes. Appelons Col(G) l'ensemble des fonctions de l'ensemble des sommets du graphe vers les couleurs 1, 2 et 3 qui définissent un 3-coloriage. Cet ensemble n'est pas de taille bornée indépendamment de G. Pour remédier à cette difficulté, on va remplacer les coloriages par leurs « réductions » aux sources. Pour chaque fonction C dans Col(G) on note $C^{\#}$ sa restriction aux sources. Plus précisément, $C^{\#}$ est la restriction du coloriage C à

l'ensemble D des étiquettes des sources, chaque source étant identifiée par une étiquette (D est fixé, de taille au plus k). Les fonctions $C^{\#}$ restent dans un ensemble fini.

Notons $Col^{\#}(G)$ l'ensemble des $C^{\#}$ pour tous les 3-coloriages (cet ensemble est de cardinalité finie bornée). La récurrence marche bien. On peut calculer l'ensemble des coloriages « réduits » pour $G /\!\!/ H$ en fonction des coloriages « réduits » pour G et pour H. On prend les couples formés d'un coloriage réduit de G et d'un coloriage réduit de H et on examine s'ils sont compatibles (si les mêmes étiquettes portent la même couleur). Si c'est le cas on peut les combiner en un coloriage réduit de $G /\!\!/ H$. Nous avons bien la propriété d'inductivité souhaitée pour l'opération de composition parallèle, ainsi que pour le renommage et l'oubli de sources.

Supposons que l'on ait un graphe G construit au moyen de ces opérations à partir de graphes de base en nombre fini (à isomorphisme près). On calcule les 3-coloriages réduits pour les graphes de base, et, au moyen d'un calcul montant dans l'arbre qui combine à chaque étape les ensembles de 3-coloriages « réduits », on obtient l'ensemble des 3-coloriages « réduits » pour G. Si cet ensemble est vide, le graphe n'est pas 3-coloriable. Lorsque l'on a obtenu un 3-coloriage « réduit » à la racine de l'arbre, donc relatif au graphe entier, on peut, en revenant en arrière dans le calcul, rassembler les résultats partiels et construire un 3-coloriage du graphe.

Le problème du 3-coloriage peut être présenté de telle façon qu'il soit vérifiable inductivement à partir d'un terme qui représente un graphe. Cette méthode peut être appliquée à tous les problèmes que l'on peut spécifier par une formule de la *logique du second ordre monadique*.

Logique du second ordre monadique

La logique permet d'écrire formellement des propriétés de graphes. L'écriture d'une propriété dans tel ou tel langage logique fournit des informations sur la complexité du problème de vérification correspondant. Un graphe G est représenté comme la structure logique relationnelle :

$$G = <S, a(.,.)>$$

L'ensemble S est l'ensemble des sommets et a est une relation binaire qui exprime simplement l'existence d'un arc entre deux sommets.

Commençons par un exemple de propriété exprimable en logique du premier ordre :

$$\mathrm{G} \vDash \forall x, \exists y, \exists z\,(a(y,x) \wedge a(x,z))$$

Implicitement, x, y et z sont des sommets car dans la structure relationnelle associée à un graphe, le domaine est constitué de l'ensemble S des sommets. La formule dit que quel que soit le sommet x, il existe un sommet y et un sommet z tels que l'on ait un arc (y,x) et un arc (x,z). Autrement dit, tout sommet est incident à un arc entrant et à un arc sortant (cela peut être une boucle, y et z peuvent être égaux). Le signe $\vDash$ indique la validité de la formule dans la structure logique considérée, ici le graphe G.

La logique du premier ordre n'est pas très expressive en termes de propriétés de graphes. La logique du second ordre monadique est bien plus expressive et fournit des résultats intéressants pour l'algorithmique et la théorie des langages formels étendue aux graphes. L'expression « second ordre » signifie que les quantifications portent sur des relations et pas seulement sur des objets individuels. « Monadique » veut dire que ces quantifications ne portent que sur des relations unaires, c'est-à-dire des ensembles. (La logique du second ordre est encore plus expressive, mais elle ne possède pas de bonnes propriétés algorithmiques.)

Prenons l'exemple de la formule :

$$\begin{aligned}\exists \mathrm{X} \quad & \{(\exists x,\ x \in \mathrm{X}) \wedge (\exists y,\ y \notin \mathrm{X}) \wedge \\ & \forall u,\ \forall v\ [(a(u,v) \vee a(v,u)) \Rightarrow (u \in \mathrm{X} \Rightarrow v \in \mathrm{X})]\}\end{aligned}$$

Elle exprime qu'un graphe n'est pas connexe, car il existe un ensemble X de sommets, non vide ($\exists x, x \in \mathrm{X}$), dont le complémentaire n'est pas vide ($\exists y, y \notin \mathrm{X}$) et tel que quels que soient u et v, si on a un arc entre u et v et si u est dans X, alors v est aussi dans X. On a donc au moins deux sous-graphes non reliés l'un à l'autre et, en conséquence, le graphe n'est pas connexe. En prenant la négation de cette formule, on exprime la connexité.

Théorème (B. Courcelle, 1990 : [1, 2, 3, 5]). *Toute propriété de la logique du second ordre monadique est vérifiable en temps linéaire sur les graphes de largeur arborescente au plus k, pour k fixé.*

Le temps de calcul est linéaire par rapport à la taille du graphe, la constante dépendant exponentiellement de la syntaxe de la propriété

(hauteur de quantification) et de la valeur de k considérée. Quelles propriétés sont concernées par ce résultat ? Tout d'abord des propriétés (NP-complètes) de colorabilité. On écrit ainsi qu'un graphe est 3-coloriable :

$$\begin{aligned}\exists X, Y, Z \quad & \{X \cap Y = \varnothing \wedge X \cap Z = \varnothing \wedge Y \cap Z = \varnothing \wedge \\ & \forall x,\ [x \in X \vee x \in Y \vee x \in Z] \wedge \\ & \quad \forall u, v,\ [a(u,v) \Rightarrow (\quad \neg(u \in X \wedge v \in X) \wedge \\ & \qquad\qquad\qquad\qquad \neg(u \in Y \wedge v \in Y) \wedge \\ & \qquad\qquad\qquad\qquad \neg(u \in Z \wedge v \in Z))]\}\end{aligned}$$

Intuitivement X, Y, Z sont les ensembles de sommets de couleurs 1,2,3. Ces ensembles sont deux à deux disjoints, ce qui évite qu'un même élément ait deux couleurs à la fois. Enfin, si on a un arc entre u et v, alors u et v ne sont pas tous deux dans X ni dans Y ni dans Z et sont donc de couleurs différentes.

Voici quelques problèmes typiques exprimables en logique du second ordre monadique : la k-colorabilité, pour chaque k fixé ; les propriétés de connexité ; la planarité, grâce au critère de Kuratowski (voir ci-dessous) ; les propriétés qui portent sur l'existence de chemins de telle ou telle forme d'un sommet vers un autre, en particulier le fait d'être un arbre.

Je vais expliquer pourquoi la propriété d'inductivité (vue dans l'exemple du 3-coloriage) s'étend à toutes les formules du second ordre monadique. Les notions de base sont les suivantes, soit G = <S, a(.,.)> un graphe, φ une formule et X, Y, Z des variables ensemblistes sur lesquelles porte la formule, ce que l'on notera φ(X, Y, Z). Définissons *l'ensemble de satisfaction de la formule* φ dans G comme l'ensemble des triplets (A, B, C) pour lesquels la formule est vraie :

$$SAT(G, \varphi, X, Y, Z) = \{(A, B, C) \mid A \subseteq S,\ B \subseteq S,\ C \subseteq S,\ G \vDash \varphi(A, B, C)\}$$

Si φ est sans variable libre :

$$\begin{aligned} SAT(G, \varphi) &= \varnothing \text{ si } G \nvDash \varphi \\ &= \{\epsilon\} \text{ (suite vide) si } G \vDash \varphi \end{aligned}$$

Il existe deux méthodes pour calculer cet ensemble de satisfaction. La première résulte des définitions : on fait une induction sur la structure de la formule (si la formule est la conjonction de deux formules φ_1

et φ_2, l'ensemble de satisfaction est l'intersection des ensembles correspondants pour φ_1 et φ_2 ; si c'est une disjonction, c'est la réunion). Mais cela ne donne rien de plus que la définition de la validité d'une formule dans une structure et il est plus intéressant de faire une induction sur un terme définissant le graphe considéré (c'est la seconde méthode). Pour cela, nous avons besoin de trois lemmes fondamentaux.

Premier lemme. Soit f une opération de graphe unaire définissable sans quantificateurs. Le renommage des sources en est un exemple. Nous voulons déterminer l'ensemble de satisfaction d'une formule φ dans un graphe $f(\mathrm{G})$ à partir de l'ensemble de satisfaction dans G d'une autre formule déterminée en fonction de f et de φ. Le premier lemme dit que :

$$\mathrm{SAT}(f(\mathrm{G}), \varphi, \mathrm{X}, \mathrm{Y}, \mathrm{Z}) = \mathrm{SAT}(\mathrm{G}, f^{\#}(\varphi), \mathrm{X}, \mathrm{Y}, \mathrm{Z})$$

où $f^{\#}(\varphi)$ est une formule de hauteur de quantification inférieure ou égale à celle de la formule φ. Un autre exemple d'opération f est la complémentation des arcs : on prend les mêmes sommets, et chaque fois qu'il n'y a pas d'arc entre deux sommets, on en ajoute un et chaque fois qu'il y en a un, on l'enlève ; en termes logiques, cela peut s'écrire sans quantificateur de la façon suivante :

$$a'(x, y) = \neg a(x, y)$$

Pour construire dans ce cas la formule $f^{\#}(\varphi)$ équivalente dans G à φ dans $f(\mathrm{G})$, on remplace chaque relation de φ par sa définition. Dans notre cas, la relation de base a est remplacée par sa négation. La hauteur de quantification (l'imbrication maximum des quantificateurs) n'est pas modifiée par la transformation.

Deuxième lemme. Le deuxième lemme, basé sur un théorème de Feferman et Vaught, consiste à vérifier une formule φ dans un graphe obtenu comme union disjointe de deux graphes. Soit un graphe défini comme l'union disjointe $\mathrm{G} \oplus \mathrm{H}$ de G et H. Soit X un ensemble vérifiant une formule $\varphi(\mathrm{X})$ dans l'union disjointe. Cet ensemble X se décompose en deux ensembles, l'un dans G noté X_1 et l'autre dans H noté X_2. Savoir si $\varphi(\mathrm{X}_1 \cup \mathrm{X}_2)$ est vraie dans $\mathrm{G} \oplus \mathrm{H}$ se réduit à vérifier la validité de formules auxiliaires indépendamment : des formules auxiliaires $\theta_i(\mathrm{X}_1)$ dans G et $\psi_i(\mathrm{X}_2)$ dans H. Autrement dit, on ramène la validité d'une formule dans un graphe composé comme une union de deux graphes

à des validités de formules auxiliaires à vérifier dans ses deux composantes, de manière indépendante. Cela s'écrit de la façon suivante (q étant la hauteur de quantification) :

$$\mathrm{SAT}(\mathrm{G}\oplus\mathrm{H},\varphi,\mathrm{X},\mathrm{Y},\mathrm{Z}) = \bigcup_{1\leqslant i\leqslant p} \mathrm{SAT}(\mathrm{G},\theta_i,\mathrm{X},\mathrm{Y},\mathrm{Z}) \bigodot \mathrm{SAT}(\mathrm{H},\psi_i,\mathrm{X},\mathrm{Y},\mathrm{Z})$$

où $q(\theta_i), q(\psi_i) \leqslant q(\varphi)$.

L'ensemble de satisfaction pour $\mathrm{G}\oplus\mathrm{H}$ est une combinaison des ensembles qui satisfont θ_i dans G et de ceux qui satisfont ψ_i dans H. L'opération $\bigodot$ consiste à prendre tous les $(\mathrm{X}_1,\mathrm{Y}_1,\mathrm{Z}_1)$ qui satisfont la formule θ_i dans G et les $(\mathrm{X}_2,\mathrm{Y}_2,\mathrm{Z}_2)$ qui satisfont la formule ψ_i dans H : on prend toutes les unions $(\mathrm{X}_1\cup\mathrm{X}_2,\mathrm{Y}_1\cup\mathrm{Y}_2,\mathrm{Z}_1\cup\mathrm{Z}_2)$ possibles de $(\mathrm{X}_1,\mathrm{Y}_1,\mathrm{Z}_1)$ avec $(\mathrm{X}_2,\mathrm{Y}_2,\mathrm{Z}_2)$ et on obtient le résultat recherché. Les formules auxiliaires θ_i,ψ_i n'ont pas de hauteur de quantification plus grande que φ.

Troisième lemme. Le troisième lemme permet de compléter les bases dont nous avons besoin : pour tout entier n l'ensemble des formules de hauteur de quantification au plus n est fini car il n'y a pas de symbole de fonction. Si on a une fonction unaire, par exemple la fonction successeur dans le cas de l'arithmétique, on peut fabriquer une infinité de termes 0, $s(0)$, $s(s(0)),\ldots$ et nécessairement une infinité de formules. Dans notre cas, nous n'avons que des relations. On a bien une infinité de formules, parce que pour toute formule A, on a les formules $\mathrm{A}\vee\mathrm{A}$, $\mathrm{A}\vee\mathrm{A}\vee\mathrm{A},\ldots$ mais elles sont toutes équivalentes. Nous pouvons les ramener à une formule minimale et les formules minimales sont en nombre fini. Pour tout n, l'ensemble des formules minimales de hauteur de quantification au plus n, sur un alphabet de relations fini et un ensemble fini de variables libres est fini.

Soit un graphe G appartenant à une famille à laquelle on s'intéresse. La théorie de niveau k de ce graphe, notée $\Theta_k(\mathrm{G})$, est l'ensemble des formules de hauteur de quantification au plus k qui sont vraies dans ce graphe.

$$\Theta_k(\mathrm{G}) = \{\varphi \mid q(\varphi)\leqslant k, \mathrm{G}\vDash\varphi\}$$

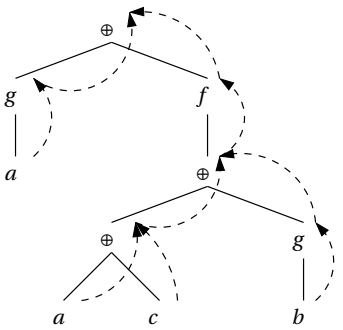
Les lemmes précédents montrent que la théorie de l'union disjointe $\mathrm{G}\oplus\mathrm{H}$ de deux graphes est une combinaison des théories des graphes G et H.

$$\Theta_k(\mathrm{G}\oplus\mathrm{H}) = \oplus^{\flat}(\Theta_k(\mathrm{G}),\Theta_k(\mathrm{H}))$$

Cette fonction $\oplus^\flat$ associée à la somme disjointe n'est qu'une reformulation de l'expression de $\mathrm{SAT}(G \oplus H, \varphi, X, Y, Z)$ vue plus haut. On va donc manipuler en bloc les théories : ce sont des ensembles finis (mais très grands) de formules associées à des graphes. Pour les opérations unaires sans quantificateurs, c'est encore plus simple. La théorie de niveau k de G est une transformation de la théorie de niveau k de H si $G = f(H)$:

$$\Theta_k(f(H)) = f^\flat(\Theta_k(H))$$

Maintenant supposons qu'un graphe G soit donné par un terme t.



Terme t

Fig. 13. Terme correspondant à un graphe et illustration du calcul montant associé

Les théories sont calculées au plus bas niveau et on monte dans l'arborescence en calculant la théorie associée à chaque nœud, au moyen d'un automate fini qui opère sur des termes. Cet automate est fini parce que l'ensemble de toutes les théories possibles est un ensemble fini. Le temps de calcul est linéaire par rapport à la taille du terme et donc, par rapport à la taille du graphe. La constante est très importante. Elle dépend de la largeur arborescente maximum autorisée et de la structure de la formule, principalement de sa hauteur de quantification.

La difficulté liée à la complexité en temps (la constante a un nombre non borné de niveaux d'exponentiation) est incontournable, elle n'est pas liée à une faiblesse de la méthode. Frick et Grohe [6] ont

montré que sous la condition P ≠ NP, il n'est pas possible de décider qu'un mot satisfait une formule φ en un temps polynomial par rapport à la longueur du mot, avec une constante bornée par une exponentielle itérée un nombre fixé de fois. La hauteur de la tour d'exponentielles doit dépendre de la formule φ.

On peut présenter ainsi le schéma général de cette méthode. On doit d'abord formaliser le problème considéré en logique du second ordre monadique (mais certaines propriétés de graphe ne peuvent pas être formalisées dans ce langage) ; il faut trouver une borne k à la largeur arborescente des graphes que l'on va considérer. En utilisant ces données, on fait appel à une sorte de *compilateur de formules logiques* qui produit un automate selon la méthode indiquée plus haut ou une méthode améliorée. C'est bien une *compilation* car le travail correspondant peut être fait une fois pour toutes. Soit maintenant un graphe à traiter. On doit le faire passer par un *analyseur syntaxique* qui prend aussi comme donnée l'entier k et qui doit répondre, ou bien que le graphe n'a pas la bonne largeur arborescente (il faut alors recommencer pour la valeur $k+1$), ou bien que le graphe a la bonne largeur arborescente, et dans ce cas, on obtient un terme qui définit le graphe. Nous pouvons alors utiliser sur ce terme l'automate construit qui fournira la réponse vrai ou faux, ou éventuellement une valeur, car ce dispositif peut aussi s'adapter à des situations où on ne veut pas seulement une réponse oui ou non, mais une valeur comme la taille maximum d'un sous-graphe planaire d'un graphe donné, et même l'un de ces sous-graphes. Si le graphe est donné par son terme, on évite la phase d'analyse syntaxique. Le nombre très important d'états de l'automate utilisé est une difficulté majeure. Mais la méthode est applicable à des cas particuliers, à défaut d'être réellement applicable à toutes les formules de la logique du second ordre monadique.

Configurations interdites et théorie des mineurs de graphes

Un graphe est *planaire* si on peut le dessiner dans le plan sans croisements d'arcs. Le graphe complet à 5 sommets K_5 et le graphe biparti complet $K_{3,3}$ ne sont pas planaires, on peut le démontrer assez facilement avec la relation d'Euler. Le graphe K_5 peut être dessiné sans croisements d'arcs sur le tore (fig. 14).

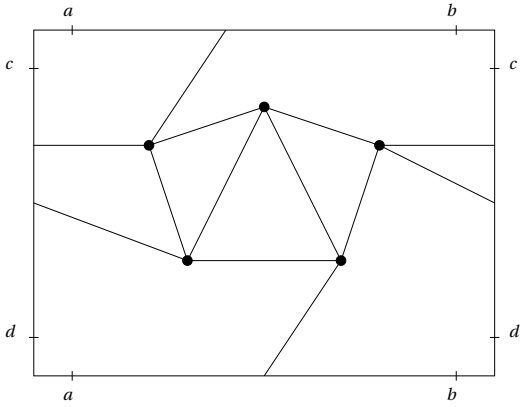


Fig. 14. Le graphe complet K_5 dessiné sur le tore

Le tore est représenté sur cette figure comme un rectangle dont on recolle les côtés opposés (les points marqués a sont identifiés, de même pour b, c, d). On a bien un dessin du graphe complet K_5 où les arêtes ne se croisent pas. Le graphe $K_{3,3}$ peut être dessiné sur le plan projectif, représenté comme un cercle dont on identifie les points diamétralement opposés.

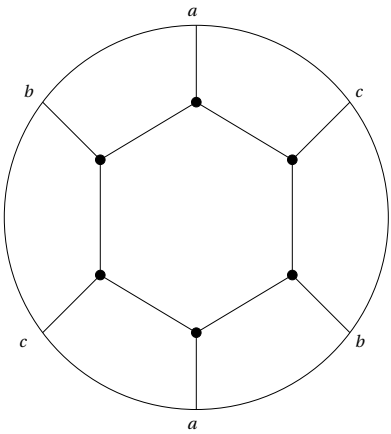


Fig. 15. Le graphe biparti complet $K_{3,3}$ dessiné sur le plan projectif

Le théorème de Kuratowski (cf. [7]) dit qu'un graphe est planaire *si et seulement si* il ne contient ni K_5, ni $K_{3,3}$ en tant que *sous-graphe topologique*. Si un graphe est planaire, ses sous-graphes sont également planaires, donc K_5 et $K_{3,3}$ ne peuvent pas apparaître comme sous-graphes topologiques d'un graphe planaire. La réciproque est plus difficile à montrer.

Un très beau théorème dû à Robertson et Seymour [9] énonce que pour toute surface, tore, plan projectif ou surface d'ordre supérieur, il y a un nombre fini de sous-graphes topologiques interdits. C'est une généralisation du théorème de Kuratowski. Ce résultat était connu pour le plan projectif : il y a 103 configurations interdites. Dans le cas du tore par contre, on ne connaît pas le nombre exact de configurations interdites, on sait qu'il y en a au moins 2200, mais on ne peut même pas dire, par exemple, si elles ont toutes moins d'une centaine d'arcs [1].

La notion de configuration interdite étudiée par Robertson et Seymour est celle de *mineur*. G est un mineur de H si et seulement si on peut construire G à partir de H en supprimant des arcs, des sommets et en contractant des arcs (fig. 16).

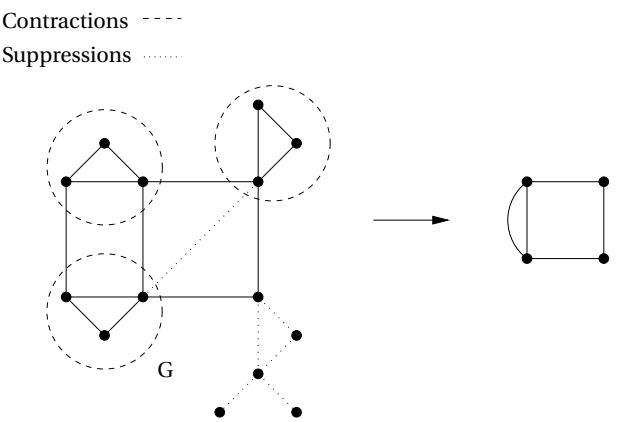


Fig. 16. Un graphe G et un mineur de G

1. En novembre 2005, j'apprends que P. Seymour [12] a déterminé, pour chaque surface, une majoration de la taille des configurations interdites. Le résultat n'est pas encore publié. (N.d.A.)

Pour tout graphe fini H, il existe une formule φ_H de la logique du second ordre monadique telle qu'un graphe satisfait cette formule *si et seulement si* il contient H comme mineur. Robertson et Seymour ont montré que pour toute famille de graphes fermée par mineur (typiquement la famille des graphes que l'on peut dessiner sur une surface, mais il y en a d'autres, par exemple pour chaque k la famille des graphes de largeur arborescente au plus k) est caractérisée par un nombre fini de mineurs exclus (la preuve a été publiée en 20 articles échelonnés sur 20 ans, voir [10]). On en déduit donc que ces familles sont définissables en logique du second ordre monadique.

Décidabilité de la logique du second ordre monadique

Je vais présenter pour finir un résultat dû à D. Seese. Il concerne la variante de la logique du second ordre monadique qui autorise les quantifications sur les ensembles d'arcs : je la note MS_2 (elle est plus expressive que le langage présenté initialement).

Théorème (D. Seese, 1991 : [11]) *Si un ensemble de graphes $\mathcal{C}$ a un problème de satisfaisabilité décidable pour la logique MS_2 (c'est-à-dire s'il existe un algorithme qui prend comme donnée une formule de la logique MS_2 et qui dit s'il existe un graphe dans $\mathcal{C}$ qui la satisfait), alors les graphes de cet ensemble ont une largeur arborescente bornée.*

Les seuls ensembles de graphes qui ont ce type de décidabilité sont les ensembles d'arbres et les ensembles de graphes qui en sont proches en ce sens qu'ils peuvent être codés par des arbres. La preuve s'appuie sur la notion de *transduction MS_2-définissable*, c'est-à-dire de transformation d'un ensemble de graphes que l'on peut spécifier par des formules de la logique MS_2. On peut ainsi transformer un graphe en l'ensemble de ses mineurs.

Soit $\mathcal{C}$ un ensemble de graphes pour lequel le problème de satisfaisabilité de la logique MS_2 est décidable. Alors l'ensemble de ses mineurs est définissable à partir de $\mathcal{C}$ par des formules de MS_2, il en résulte que le problème de satisfaisabilité pour les mineurs est décidable. Un résultat de Robertson et Seymour [8] dit que si un ensemble de graphes finis a une largeur arborescente non bornée, alors l'ensemble de leurs mineurs contient des grilles carrées de taille arbitrairement grande. Les *configurations d'une machine de Turing*, c'est-à-dire les étapes d'un calcul, forment une suite de mots qui comprend la donnée

et les calculs annexes faits par la machine de Turing. Dans des grilles de taille non bornée et au moyen d'une formule de la logique MS_2 on peut coder les calculs d'une machine de Turing qui se terminent. Cela veut dire que sur un ensemble de graphes qui contient une infinité de grilles carrées, la logique du second ordre monadique est indécidable du fait que l'on peut y coder l'arrêt d'une machine de Turing. En conséquence, une classe de graphes pour laquelle la logique MS_2 a un problème de satisfaisabilité décidable est de largeur arborescente bornée.

C'est un résultat qui relie des notions de logique et de calculabilité à des notions de structuration des graphes. Il s'appuie sur un résultat difficile de nature combinatoire.

Conclusion

Cet exposé est un aperçu rapide d'une recherche très vivante qui concerne plusieurs thématiques : les structurations des graphes, les « configurations interdites », la théorie des langages formels et son extension aux graphes, le pouvoir d'expression et la décidabilité de langages logiques et la construction d'algorithmes polynomiaux pour certains problèmes difficiles. Les liens sont nombreux et étroits entre ces différents sujets. Je développe cette théorie dans un livre à paraître prochainement [3]. La bibliographie ci-dessous est volontairement très limitée.

Bibliographie

[1] B. Courcelle, *The Monadic Second-Order Logic of Graphs. I : Recognizable Sets of Finite Graphs*, Information and Computation **85** (1990), p. 12-75.

[2] B. Courcelle, *The expression of graph properties and graph transformations in monadic second-order logic*, In G. Rozenberg, editor, Handbook of graph grammars and computing by graph transformation, vol. 1 : Foundations, World Scientific, 1997 : chapter 5, p. 313-400.

[3] B. Courcelle, *Graph structure and monadic second-order logic*, livre à paraître chez Cambridge University Press. Consultable en ligne : `http://www.labri.fr/perso/courcell/ActSci.html`

[4] J.-P. Delahaye, *Un algorithme à un million de dollars ?*, Pour La Science, n°**334** (août 2005), p. 90-95.

[5] R. G. Downey and M. R. Fellows, *Parameterized Complexity*, Springer, 1999 [Très complet sur les décompositions arborescentes et leurs applications algorithmiques ; le théorème de D. Seese y est aussi présenté.]

[6] M. Frick and M. Grohe, *The Complexity of First-order and Monadic Second-order Logic Revisited*, Annals of Pure and Applied Logic **130** (2004), p. 3-31.

[7] B. Mohar and C. Thomassen, *Graphs on Surfaces*, John Hopkins University Press, Baltimore, 2001.

[8] N. Robertson and P. Seymour, *Graph minors* v *: Excluding a planar graph*, J. Combin. Theory, ser. B **41** (1986), nº1, p. 92-114.

[9] N. Robertson and P. Seymour, *Graph minors* viii *: A Kuratowski theorem for general surfaces*, J. Combin. Theory, ser. B **48** (1990), nº2, p. 255-288.

[10] N. Robertson and P. Seymour, *Graph minors* xx *: Wagner's conjecture*, J. Combin. Theory, ser. B **92** (2004), nº2, p. 325-357.

[11] D. Seese, *The structure of the models of decidable monadic theories of graphs*, Ann. Pure Appl. Logic, **53** (1991), nº2, p. 169-195.

[12] P. Seymour, *A bound on the excluded minors for a surface* (1995), soumis pour publication.