



HAL
open science

Génération et tracé de structures décomposables

Francois Bertault

► **To cite this version:**

Francois Bertault. Génération et tracé de structures décomposables. Informatique [cs]. Université Henri Poincaré - Nancy I, 1997. Français. NNT: . tel-00557784

HAL Id: tel-00557784

<https://theses.hal.science/tel-00557784v1>

Submitted on 20 Jan 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Génération et tracé de structures décomposables

THÈSE

présentée et soutenue publiquement le 24 septembre 1997

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1
(spécialité informatique)

par

François Bertault

Composition du jury

<i>Président :</i>	Philippe Flajolet	Directeur de recherches INRIA, Rocquencourt.
<i>Rapporteurs :</i>	Maylis Delest	Professeur, Université de Bordeaux I,
	Claude Godart	Professeur, Université Henri Poincaré, Nancy 1,
	Jeffrey S. Vitter	Professeur, Duke University, États-Unis.
<i>Examineurs :</i>	Peter D. Eades	Professeur, University of Newcastle, Australie,
	Pierre Lescanne	Professeur, École Normale Supérieure, Lyon,
	Paul Zimmermann	Chargé de recherches INRIA, Nancy.

Table des matières

Introduction	1
Chapitre 1 Dénombrement et génération aléatoire uniforme de structures décomposables	9
1.1 Structures combinatoires	9
1.1.1 Définitions	9
1.1.2 Spécifications	11
1.1.3 Les classes de spécification Ω et $\hat{\Omega}$	13
1.1.4 Ensembles de structures particuliers	14
1.2 Dénombrement	14
1.2.1 Univers non étiqueté	14
1.2.2 Univers étiqueté	19
1.2.3 Solutions particulières	22
1.2.4 Mise en œuvre	22
1.3 Génération aléatoire uniforme	25
1.3.1 Principe	25
1.3.2 Mise en œuvre	28
1.4 Extensions	29
1.4.1 Nouveaux constructeurs	29
1.4.2 Dénombrement	31
1.4.3 Génération aléatoire	32
1.4.4 Exemples d'utilisation	33
1.5 Conclusion	37
Chapitre 2 Méthode de rang inverse sur des ensembles de structures décomposables	41
2.1 Le problème du rang inverse	41
2.1.1 Définition	41
2.1.2 Arbres binaires planaires	41
2.1.3 Structures décomposables	43
2.2 Formes	43
2.2.1 Définitions	43
2.2.2 Énumération de formes	43
2.2.3 Comptage des structures dans une forme	48

2.3	Algorithme de comptage de structures	50
2.4	Conditions supplémentaires sur les structures	50
2.4.1	Arité des multi-constructeurs	51
2.4.2	Taille des composantes	51
2.5	Rang inverse d'une forme	52
2.5.1	Univers non étiqueté	53
2.5.2	Univers étiqueté	56
2.6	Algorithme de rang inverse	56
2.7	Algorithme de rang inverse incrémental	58
2.7.1	Formes <i>Union</i>	58
2.7.2	Formes <i>Prod</i>	59
2.7.3	Formes <i>Set</i>	59
2.7.4	Formes <i>Cycle</i>	60
2.8	Mise en œuvre	60
2.9	Conclusion	64
Chapitre 3 Représentation graphique de graphes orientés et non orientés		65
3.1	Le problème de la représentation graphique des graphes	65
3.2	Définitions	67
3.2.1	Graphes non-orientés	67
3.2.2	Graphes orientés	68
3.3	Tracé d'arbres enracinés planaires	69
3.3.1	Arbres binaires	69
3.3.2	Cas des arbres k -aires avec des nœuds de taille quelconque	70
3.3.3	Exemples	80
3.4	Arbres enracinés en « cascade »	83
3.5	Graphes non orientés	83
3.5.1	Algorithme FR	83
3.5.2	Principe	83
3.5.3	Algorithme FRCC	89
3.6	Graphes orientés	92
3.6.1	Algorithme à « ressorts »	92
3.6.2	Algorithme classique	96
3.7	Conclusion	101
Chapitre 4 Représentation graphique de graphes composés		105
4.1	Introduction	106
4.2	Définitions	106
4.2.1	Graphes composés	106
4.2.2	Notations	109
4.3	Algorithme de tracé de graphes composés imbriqués	109
4.3.1	Placement relatif des nœuds	110

4.3.2	Placement absolu des nœuds	110
4.3.3	Transformations de graphes en fonction des modes d'inclusion	111
4.4	Représentation des structures décomposables	114
4.4.1	Représentation standard	114
4.4.2	Représentations particulières	117
4.5	Algorithme de tracé de graphes composés enracinés	119
4.5.1	Principe de l'algorithme	122
4.5.2	Étapes de l'algorithme	124
4.6	Algorithme de tracé de graphes composés quelconques	129
4.7	Visualisation de graphes	129
4.7.1	Sélection visuelle de parties d'un graphe	129
4.7.2	Réduction du nombre d'arêtes d'un graphe	129
4.8	Conclusion	131
Chapitre 5 Réalisations logicielles		135
5.1	Padnon	135
5.2	CGraph	136
5.3	Conclusion	138
Bibliographie		139
Index		143

Table des figures

1	Le train aléatoire de P. Flajolet [23].	3
1.1	Dénombrement du nombre d'arbres généraux non planaires : comparaison des temps de calcul entre le programme C généré par Ccombstruct et la version Maple-combstruct . . .	24
1.2	Dénombrement du nombre d'arbres généraux non planaires : accélération du programme C généré par Ccombstruct , par rapport à la version Maple-combstruct	24
1.4	Principe de l'algorithme de génération aléatoire uniforme dans le cas d'une production de la forme : $A \rightarrow Prod(B, C)$	26
1.6	Les trois chemins de Dyck d'aire 8 et de longueur 8.	39
1.7	Les 6 ensembles de chemins de Dyck d'aire 4.	39
3.1	Le problème des ponts de la ville de Königsberg	66
3.2	Le graphe correspondant à la figure 3.1.	66
3.4	Principe de l'algorithme de tracé d'arbres binaires enracinés	69
3.6	Exemple de mise à jour des liens	74
3.7	Placement vertical des sous-arbres	75
3.8	Placement horizontal des sous-arbres : cas (1).	76
3.9	Placement horizontal des sous-arbres : cas (2).	76
3.10	Placement horizontal des sous-arbres : mise à jour des liens (1)	76
3.11	Placement horizontal des sous-arbres : mise à jour des liens (2)	76
3.12	Placement horizontal des sous-arbres : mise à jour des liens (3)	76
3.13	Placement de la racine des sous-arbres.	78
3.15	Temps de calcul des positions absolues des nœuds d'arbres généraux.	79
3.16	Temps de calcul des positions absolues des nœuds d'arbres binaires.	79
3.21	Principe de l'algorithme de tracé en « cascade ».	84
3.22	Calcul de la distance souhaitable d_{opt} entre les centres de deux nœuds en fonction de la longueur souhaitée δ_e de l'arête les reliant.	85
3.30	Mise à jour des déplacements autorisés.	89
3.31	Calcul des interactions entre nœuds et arêtes.	92
3.36	Ajout de nœuds temporaires.	99
4.2	Graphe orienté.	113
4.3	Transformation du graphe de la figure 4.2 en un arbre enraciné.	113
4.4	Schéma de conversion pour les structures de la forme $cycle(s_1, \dots, s_k)$	115
4.5	Conversion des structures de la forme $prod(s_1, prod(s_{2,1}, \dots, s_{2,l_2}), \dots, s_k)$	116
4.9	Graphe associé aux structures de la forme $prod(a, sequence(s_1, \dots, s_k))$	120
4.10	Graphe associé aux structures de la forme $cycle(s_1, prod(a, sequence(s_{2,1}, \dots, s_{2,k})), \dots, s_k)$ 120	120
4.14	Modification du graphe de la figure 4.13 (solution 1).	123
4.15	Modification du graphe de la figure 4.13 (solution 2).	123
5.1	Padnon : fenêtre de visualisation du graphe	135
5.2	CGraph : fenêtre principale	136
5.3	Choix des préférences graphiques et des modes de nœuds du graphe.	137

Table des figures réalisées avec CGraph

2	Le train de la figure 1 dessiné automatiquement.	3
3	Permutation cyclique de 20 éléments.	4
4	Un arbre de la spécification (3) de taille 300 généré de façon aléatoire uniforme.	6
1.5	Arbre de Motzkin de taille 1000 généré aléatoirement.	30
2.1	Représentation graphique du résultat de l'algorithme de rang inverse, pour les arbres généraux non planaires de taille 5	62
2.2	Représentation graphique des 48 arbres généraux non planaires de taille 7	62
2.3	Arbres généraux avec permutation cyclique des fils.	63
2.4	Les 16 arbres généraux non planaires de taille 7 dont les sous-arbres sont des feuilles ou de taille un nombre premier	63
2.5	Les 24 ensembles de taille 6 formés de colliers bicolores de longueur au moins 3.	64
3.3	Graphe orienté dessiné par niveau.	67
3.5	Arbre k -aire avec nœuds de différentes tailles : nœuds placés « au plus près » <i>vs</i> nœuds placés par niveaux.	71
3.14	Arbre général de taille 500 dessiné avec et sans le critère <i>vi</i>	78
3.17	La famille aliquote de l'entier 37.	81
3.18	Arbre général de taille 150 dont les nœuds ont des tailles différentes.	82
3.19	Arbre binaire récursif représentant une structure de taille 49 issue de la spécification (3.1).	82
3.20	Arbre enraciné suivant les conventions en « cascade » et classique.	84
3.23	Exemple d'application de l'algorithme FR pour un graphe ayant des nœuds de tailles différentes.	86
3.24	Graphe B.	87
3.25	Graphe de Cayley du groupe de permutations de générateurs $((1, 2))$ et $((1, 3))$	88
3.26	Graphe de Cayley du groupe de permutations de générateurs $((1, 2))$, $((1, 3))$ et $((5, 6))$ (algorithme FR).	90
3.27	Graphe de Cayley du groupe de permutations de générateurs $((1, 2))$ et $((1, 3, 4))$ (algorithme FR).	90
3.28	Graphe de Cayley du groupe de permutations de générateurs $((1, 2))$, $((1, 3))$ et $((5, 6))$ (algorithme FRCC).	91
3.29	Graphe de Cayley du groupe de permutations de générateurs $((1, 2))$ et $((1, 3, 4))$ (algorithme FRCC).	91
3.32	Exemples de tracés obtenus avec l'algorithme FRCC	93
3.33	Placement des nœuds par niveaux suivant un algorithme d'ordonnement de liste suivant le chemin critique.	98
3.34	Placement des nœuds par niveaux avec minimisation des longueurs des arêtes par la méthode du simplexe.	98
3.35	Hiérarchie Unix.	98
3.37	Graphe de toutes les dérivations issues du terme $ss0 \times ss0$, pour le système de réécriture (3.3).	103
3.38	Graphe orienté dont les nœuds sont de taille différente.	104
4.1	Représentation d'une structure de la spécification (4.1).	107
4.6	Conversion standard d'une structure décomposable issue de la spécification (4.2).	117

4.7	Graphe série-parallèle.	118
4.8	Représentation classique du graphe fonctionnel de la figure 4.6.	120
4.11	Représentation d'une structure A de taille 20 de la spécification (4.4).	121
4.12	Graphe série-parallèle de taille 10.	121
4.13	Graphe imbriqué ne vérifiant pas la propriété ii de la définition 4.2.7.	122
4.16	Exemple de restitution d'arêtes modifiées.	127
4.17	Exemple de restitution d'arêtes orientées.	128
4.18	Exemple de restitution d'arêtes orientées.	128
4.19	Graphe avant réduction.	130
4.20	Graphe de la figure 4.19, après réduction des parties grisées.	130
4.21	Graphe non orienté.	131
4.22	Graphe équivalent au graphe de la figure 4.21	131
4.23	Suite de transformations équivalentes.	132

Introduction

L'objet de cette thèse est la réalisation d'algorithmes et d'outils d'aide à l'étude des propriétés de structures combinatoires particulières, les structures décomposables. Nous nous intéressons pour cela à la génération aléatoire et systématique de structures décomposables, puis à leur représentation graphique automatique. Ce travail se situe à la frontière entre calcul mathématique et visualisation.

Les structures décomposables

Il existe deux approches pour décrire des ensembles d'objets combinatoires. La première consiste à caractériser les ensembles à partir des propriétés de leurs éléments : c'est une démarche intentionnelle. La seconde repose sur la définition de *constructeur*, que l'on applique pour former de nouvelles structures à partir de structures existantes. On décrit alors les ensembles de structures en explicitant les constructeurs à partir desquels les structures de l'ensemble sont construites : c'est une démarche extensionnelle. C'est cette dernière solution que nous retenons ici. Les *structures décomposables* sont les structures combinatoires qu'il est possible de former récursivement en utilisant des constructeurs aux propriétés particulières. Le point de vue est similaire à celui adopté dans la théorie des espèces de structures [1, 36], où l'on privilégie la description d'ensembles de structures à partir de transformations d'ensembles existants. Il est alors possible, grâce à des *spécifications*, de décrire une infinité d'ensembles de structures combinatoires parmi lesquels les permutations, les graphes fonctionnels, les arbres enracinés ou encore les hiérarchies. L'intérêt de cette démarche tient au fait que l'on sait résoudre des problèmes de dénombrement et de comportement asymptotique sur ces ensembles [23] et générer aléatoirement de façon uniforme des structures de ces ensembles. Les applications concernent le calcul de complexité en moyenne d'algorithmes [24, 46, 58, 62], et la génération de jeux de tests pour la validation expérimentale ou l'étalonnage d'algorithmes.

Pour illustrer la diversité des ensembles de structures qu'il est possible de décrire, nous pouvons reprendre l'exemple proposé par P. Flajolet dans l'article [23], qui correspond à un modèle de train combinatoire. L'ensemble des trains est défini par la spécification suivante :

$$\left\{ \begin{array}{ll}
 \textit{train} & \rightarrow \textit{Prod}(\textit{locomotive}, \textit{Sequence}(\textit{wagon})) \\
 \textit{locomotive} & \rightarrow \textit{Sequence}(\textit{slice}) \\
 \textit{slice} & \rightarrow \textit{Union}(\textit{Prod}(\mathbf{upper}, \mathbf{lower}), \textit{Prod}(\mathbf{upper}, \mathbf{lower}, \textit{wheel})) \\
 \textit{wheel} & \rightarrow \textit{Prod}(\mathbf{center}, \textit{Cycle}(\mathbf{wheel_element})) \\
 \textit{wagon} & \rightarrow \textit{Prod}(\textit{locomotive}, \textit{Set}(\textit{passenger})) \\
 \textit{passenger} & \rightarrow \textit{Prod}(\textit{head}, \textit{belly}) \\
 \textit{head} & \rightarrow \textit{Cycle}(\mathbf{passenger_element}) \\
 \textit{belly} & \rightarrow \textit{Cycle}(\mathbf{passenger_element}) \\
 \mathbf{upper} & \rightsquigarrow \textit{Atom} \\
 \mathbf{lower} & \rightsquigarrow \textit{Atom} \\
 \mathbf{center} & \rightsquigarrow \textit{Atom} \\
 \mathbf{wheel_element} & \rightsquigarrow \textit{Atom} \\
 \mathbf{passenger_element} & \rightsquigarrow \textit{Atom}.
 \end{array} \right. \quad (1)$$

Le constructeur *Union* forme l'union disjointe d'ensembles de structures, le constructeur *Prod* le produit cartésien d'ensembles de structures. Étant donné un ensemble *A*, les constructeurs *Sequence*, *Set* et *Cycle* forment l'ensemble formé respectivement par les suites, les ensembles ou les cycles orientés d'éléments de

l'ensemble A . Les règles de la forme $\mathbf{a} \rightsquigarrow Atom$ indiquent que l'ensemble \mathbf{a} contient un élément de taille un, la taille d'une structure étant déterminée par le nombre d'éléments de taille un la constituant.

Un train de taille 88 est dessiné à la main dans [23], que l'on peut voir reproduit sur la figure 1. Le train est composé d'une locomotive suivie de wagons dans un ordre précis. Les roues du train sont définies à un ordre cyclique près, et les passagers ne sont pas ordonnés entre eux (ils peuvent se déplacer librement dans un même wagon). Nous pouvons voir sur la figure 2 une représentation possible, obtenue automatiquement à l'aide des techniques exposées dans cette thèse, du train de la figure 1.

L'exemple du train combinatoire est anecdotique, mais illustre les difficultés de représentation du fait de la variété des structures étudiées. Nous nous intéressons dans cette thèse à l'étude des propriétés de structures combinatoires. Une représentation graphique obtenue automatiquement permet, après affichage de plusieurs structures générées automatiquement, de faire ressortir intuitivement les propriétés combinatoires de structures de grande taille. Il peut être également intéressant de générer toutes les structures d'une taille donnée d'un ensemble de structures, afin de voir si un comportement extrême existe. Cela nous conduit à nous intéresser aux problèmes suivants :

- i. connaissant une spécification d'un ensemble de structures décomposables, comment générer de façon aléatoire uniforme des structures de grande taille?
- ii. comment construire, connaissant une spécification d'un ensemble de structures, l'ensemble de toutes les structures décomposables ayant une taille donnée?
- iii. comment représenter graphiquement les structures décomposables de façon naturelle, de manière automatique?

Ces trois questions correspondent aux trois points abordés dans cette thèse, qui concernent la combinatoire énumérative et la génération aléatoire, les problèmes de rang inverse, et enfin la représentation graphique de structures combinatoires.

Combinatoire énumérative, génération aléatoire

L'énumération et la génération aléatoire de structures décomposables sont des problèmes qui ont été résolus en partie par P. Flajolet, P. Zimmermann et B. Van Cutsem dans [26]. Ce travail s'est concrétisé par la réalisation par P. Zimmermann puis E. Murray de la bibliothèque `Maple combstruct` (appelée aussi `Gaia`) [63]. En pratique, les structures que l'on peut générer en temps raisonnable (un jour CPU) sont de taille inférieure à 2000. Notre contribution dans ce domaine est la réalisation d'une bibliothèque `Maple` fournissant automatiquement des programmes écrits en langage `C` qui permettent de compter et de générer de façon exacte des structures décomposables plus rapidement, et ce jusqu'à des structures de taille 5000 environ. Compter de façon exacte le nombre d'arbres généraux non planaires de taille 1000 prend ainsi moins de deux minutes avec le programme écrit en `C`, contre plus de vingt avec `combstruct`.

Rang inverse

Les méthodes de rang inverse consistent à mettre en bijection un ensemble de structures et un intervalle d'entiers, et, étant donné un entier de cet intervalle, à exhiber la structure associée. Leur intérêt est double. Elles permettent de réaliser la génération aléatoire de structures, en choisissant simplement un entier dans l'intervalle de façon uniforme, puis en construisant la structure associée. Elles permettent surtout de générer une à une toutes les structures de l'ensemble. Si le problème est bien connu pour les structures arborescentes [43] et les grammaires hors-contexte, le cas des structures décomposables utilisant des constructeurs *Set* (multi-ensemble) ou *Cycle* est plus difficile. Les formules d'énumération des structures deviennent alors complexes, et il n'est plus possible d'en déduire directement un algorithme de rang inverse. Nous proposons dans cette thèse d'introduire la notion de forme d'une structure, ce qui nous permet de concevoir un algorithme de rang inverse unique quels que soient les constructeurs utilisés. Concevoir un algorithme de rang inverse pour des ensembles de structures utilisant un constructeur particulier revient alors à décrire des opérations simples portant sur des formes spécifiques à ce constructeur.

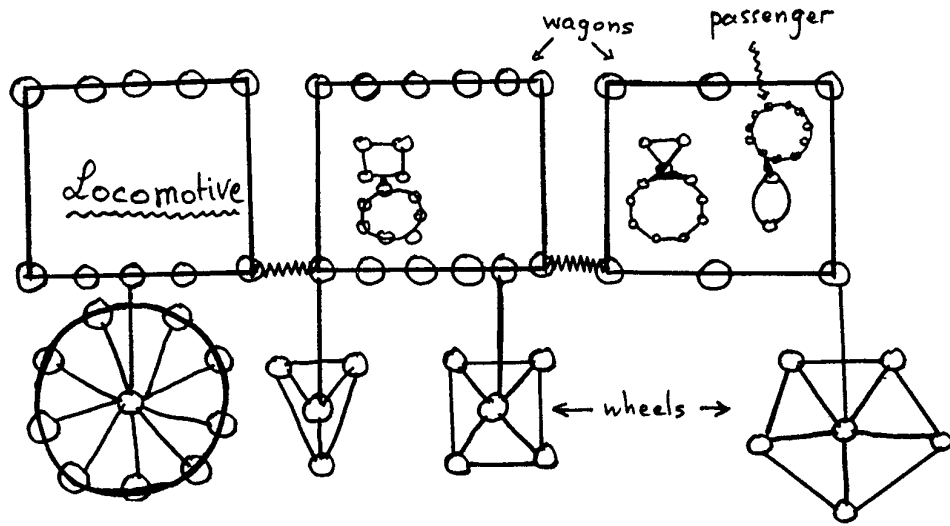


Fig. 1: *Le train aléatoire de P. Flajolet [23].*

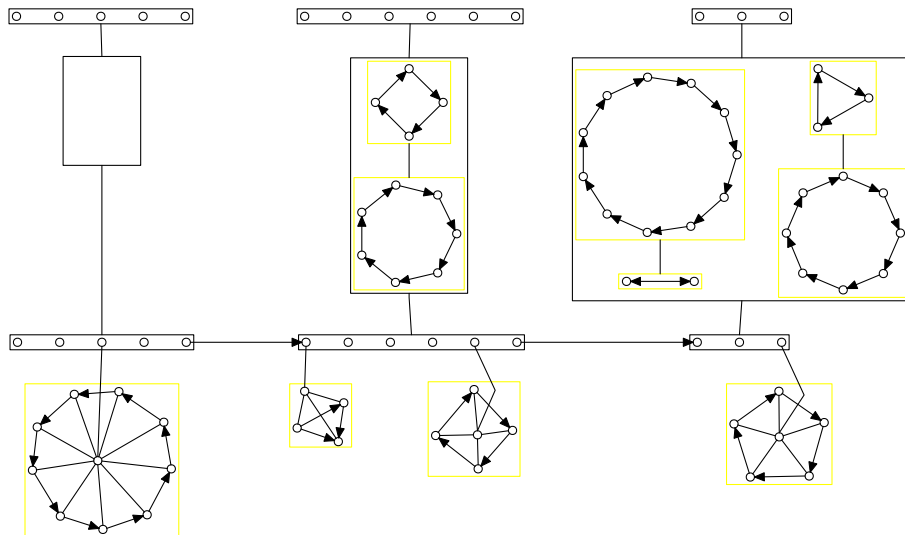


Fig. 2: *Le train de la figure 1 dessiné automatiquement.*

Le problème général se ramène à une succession de sous problèmes plus simples. Nous présentons une version incrémentale d'un algorithme de rang inverse utilisant ce principe, et nous décrivons les opérations nécessaires à son utilisation dans le cas d'ensembles de structures décrits par des constructeurs *Union*, *Prod*, *Set* et *Cycle*.

Représentation de structures combinatoires

Il existe deux approches différentes pour la représentation de structures combinatoires. La première consiste à écrire des modules de représentation spécifiques à chaque classe de structure combinatoire, la seconde consiste à considérer les structures manipulées comme des graphes.

Le logiciel CalCo [12, 45] développé au LaBRI rassemble des modules de représentation et de manipulation de structures combinatoires variées, parmi lesquelles les arbres enracinés, les graphes, les polyominos, les tresses, les chemins et les permutations. Ces différents modules peuvent communiquer entre eux ou avec d'autres programmes comme **Maple**. L'intérêt de cette approche est que la représentation des structures combinatoires peut être choisie spécifiquement pour la classe manipulée, ce qui permet des représentations naturelles et intuitives. Dans le cas particulier des structures décomposables, il n'est pas réaliste d'écrire un module pour chaque nouvelle spécification utilisée car elles sont en nombre potentiellement infini. Il faut donc concevoir un outil permettant de manipuler toutes les classes de structures décomposables, tout en laissant une certaine liberté quant aux représentations choisies. Pour cela, nous nous ramenons aux problèmes généraux de représentation de graphes.

Les algorithmes de tracé de graphes actuels sont dédiés à des classes particulières comme les arbres, les graphes planaires, les graphes généraux, les graphes dirigés par exemple. Un état de l'art a été réalisé dans [13]. Ramener le problème de la représentation des structures décomposables au tracé de l'une de ces classes particulières n'est pas satisfaisant. Regardons par exemple le cas de la spécification suivante, qui correspond aux permutations d'éléments données par leur décomposition cyclique :

$$\left\{ \begin{array}{l} P \rightarrow Set(Cycle(\mathbf{e})) \\ \mathbf{e} \rightsquigarrow Atom \end{array} \right.$$

Le tracé le plus intuitif d'une structure issue de cette spécification, serait certainement effectué à l'aide d'un algorithme de tracé de graphe généraux, comme par exemple sur la figure 3.

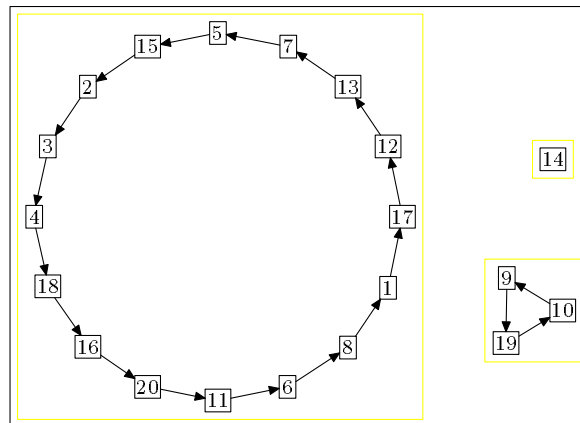


Fig. 3: *Permutation cyclique de 20 éléments.*

La spécification suivante utiliserait un algorithme de tracé d'arbres enracinés :

$$\left\{ \begin{array}{l} B \rightarrow Union(\mathbf{e}, Prod(B, B)) \\ \mathbf{e} \rightsquigarrow Atom \end{array} \right. \quad (2)$$

Mais comment représenter les structures de la spécification suivante, correspondant à un arbre binaire dont les nœuds sont substitués par des permutations?

$$\left\{ \begin{array}{l} B \rightarrow \text{Union}(\mathbf{e}, \text{Prod}(P, B, B)) \\ P \rightarrow \text{Set}(\text{Cycle}(\mathbf{e})) \\ \mathbf{e} \rightsquigarrow \text{Atom} \end{array} \right. \quad (3)$$

Une représentation naturelle de ces structures pourrait être celle de la figure 4. L'idée développée dans cette thèse pour représenter les structures décomposables, consiste à considérer le tracé de graphes particuliers, à savoir les graphes composés, pour lesquels il existe à la fois des relations d'adjacence et d'inclusion entre les nœuds, et de tracer ces graphes en faisant cohabiter plusieurs algorithmes de tracé classiques sur le même dessin. La conversion entre structure décomposable et graphe composé peut être réalisée par une conversion *standard* qui a l'avantage de faire ressortir les propriétés combinatoires des constructeurs utilisés dans la spécification, ou peut être effectuée de façon spécifique par l'utilisateur. Ceci permet d'obtenir la souplesse suffisante pour traiter des exemples comme celui du train combinatoire de la spécification (1). Cette méthode est au final un compromis entre l'écriture d'interfaces pour la représentation de classes de structures combinatoires spécifiques, et la représentation systématique par des graphes quelle que soit la classe considérée.

Plan de la thèse

Chapitre 1 – Dénombrement et génération aléatoire uniforme de structures décomposables

Le chapitre 1 introduit la notion de structure décomposable. Nous décrivons en détail les techniques, proposées par P. Flajolet, P. Zimmermann et B. Van Cutsem [26], de comptage et de génération aléatoire uniforme d'ensembles de structures. Nous présentons une implantation de ces techniques, qui permet de produire automatiquement, quelles que soient les spécifications, des programmes de comptage et de génération aléatoire écrits en langage `C`. Nous proposons également une extension de la notion de taille des structures, par l'introduction de nouveaux constructeurs, ce qui permet de traiter de nouveaux problèmes.

Chapitre 2 – Méthode de rang inverse sur des ensembles de structures décomposables

Le chapitre 2 propose un algorithme incrémental de rang inverse, basé sur l'utilisation de la notion de forme d'une structure, pour les ensembles de structures décomposables définis à l'aide des constructeurs *Prod*, *Union*, *Set* et *Cycle*.

Chapitre 3 – Représentation graphique de graphes orientés et non orientés

Dans le chapitre 3, nous étudions le problème du tracé de graphes pour des classes particulières de graphes. Nous présentons des algorithmes permettant le tracé d'arbres enracinés, de graphes orientés acycliques, ainsi que de graphes généraux. Nous présentons deux algorithmes originaux, l'un de tracé de graphes non orientés qui permet de conserver des propriétés de partitionnement des nœuds et des arêtes d'un graphe, l'autre de tracé de graphes orientés. Les autres algorithmes correspondent à des adaptations souvent directes d'algorithmes classiques de tracé, connus pour être performants, au cas de graphes dont les nœuds sont de tailles arbitrairement grandes. Nous présentons en particulier un algorithme de tracé d'arbres enracinés dont la particularité est d'utiliser une méthode classique qui semble au premier abord inadapté au cas de nœuds de tailles différentes.

Chapitre 4 – Représentation graphique de graphes composés

Le chapitre 4 contient la description de l'algorithme permettant le tracé des structures décomposables proprement dit. Nous définissons la notion de graphe composé, qui permet de représenter des relations

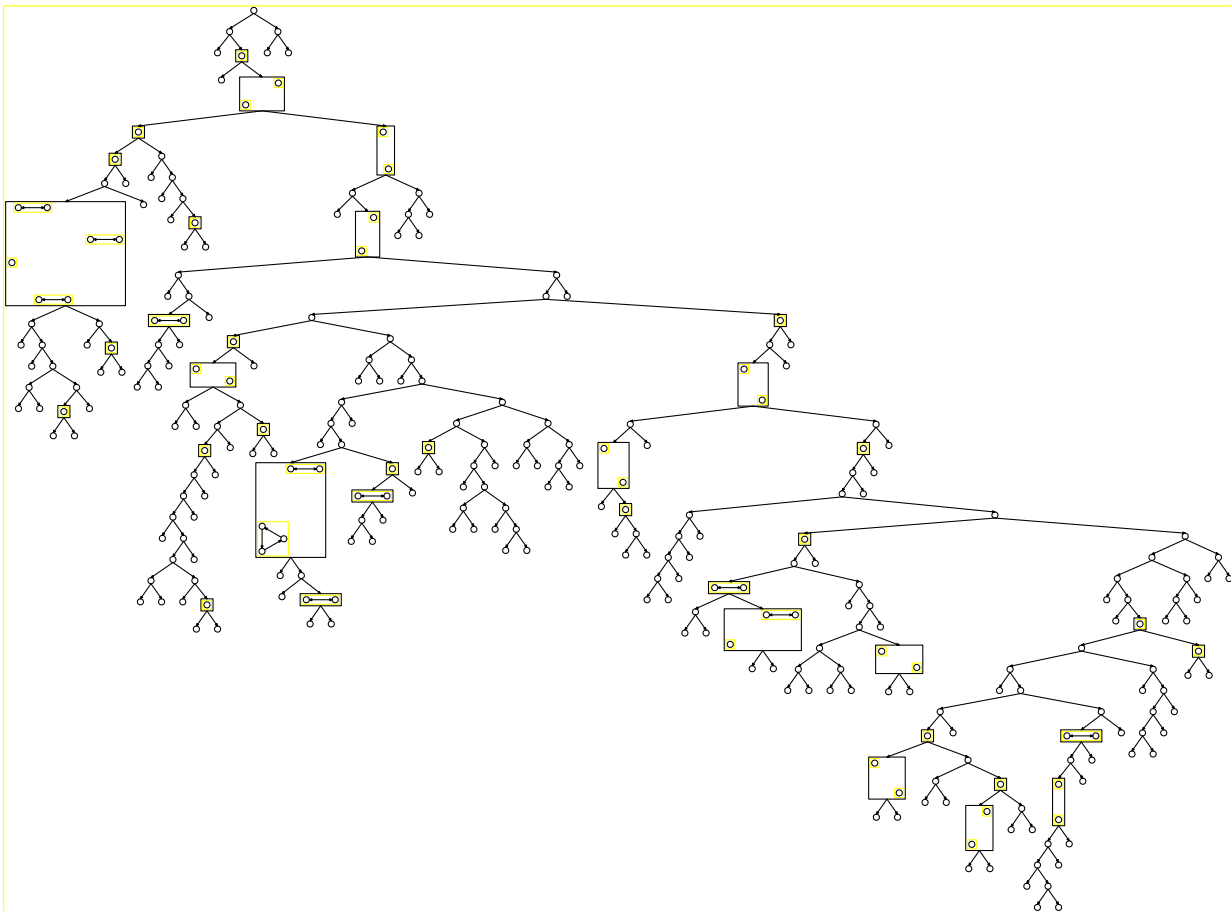


Fig. 4: Un arbre de la spécification (3) de taille 300 généré de façon aléatoire uniforme.

d'adjacence et d'inclusion entre des nœuds, et nous détaillons un algorithme permettant de les représenter. La particularité de cet algorithme est qu'il permet d'utiliser plusieurs algorithmes de tracé sur une même représentation, et en particulier ceux présentés au chapitre 3. Nous présentons également une méthode de conversion standard de structures décomposables en graphes composés, ainsi que des méthodes de conversion spécifiques sur des cas particuliers.

Chapitre 5 – Réalisations logicielles

Nous présentons dans ce dernier chapitre deux logiciels qui mettent en application les algorithmes de tracé présentés au cours de cette thèse. Le logiciel **Padnon** permet la représentation de graphes orientés et non orientés en trois dimensions. Il implante deux algorithmes de tracé présentés au chapitre 3. Le logiciel **CGraph** est un outil complet permettant de représenter les structures décomposables. Il met en œuvre l'algorithme de tracé de graphes composés présenté au chapitre 4. Ce logiciel est conçu pour être interfacé avec d'autres logiciels, et en particulier avec le système de calcul formel **Maple** et la bibliothèque de génération aléatoire **combstruct**, pour former un système intégré de génération et de représentation de structures décomposables.

Chapitre 1

Dénombrement et génération aléatoire uniforme de structures décomposables

DANS CE CHAPITRE nous nous intéressons aux structures combinatoires, à la façon de décrire des ensembles de structures, au problème qui consiste à compter les structures d'un même ensemble, ainsi qu'à la génération aléatoire de structures éléments de ces ensembles. Nous présentons en détail les techniques et algorithmes utilisés pour résoudre ces problèmes, dans le cadre de la théorie des structures décomposables. Ces algorithmes sont actuellement utilisés dans la bibliothèque `combstruct`, écrite en `Maple` par P. Zimmermann et E. Murray.

Notre contribution dans ce chapitre est double. La première est technique et se concrétise par la réalisation d'un logiciel, appelé `Ccombstruct`, qui permet de générer automatiquement, à partir d'une description d'un ensemble de structures, un programme écrit en langage `C` permettant de compter et de générer des structures. Cette méthode permet d'accélérer notablement les temps de calcul et de résoudre des problèmes pour des structures de plus grande taille. La seconde contribution réside en une extension de la théorie des structures décomposables, qui consiste à permettre de changer la notion de taille des structures par l'intermédiaire de nouveaux constructeurs « paramétrés ». Ceci permet de traiter des problèmes qu'il était impossible de traiter précédemment, sans changer dans la plupart des cas la complexité des algorithmes de comptage et de génération aléatoire. Cette extension est également implantée dans le programme `Ccombstruct` et nous illustrons son utilisation sur plusieurs exemples.

1.1 Structures combinatoires

1.1.1 Définitions

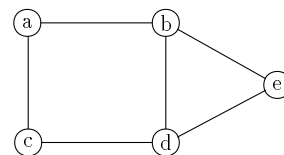
Commençons tout d'abord par définir la notion de structure, illustrée par un exemple simple, telle qu'elle est présentée dans [1].

Définition 1.1.1 (Structure)

Une *structure* s est un couple (γ, U) , où γ est une *construction* et U est l'*ensemble sous-jacent* de la structure s . .

Exemple. La structure dessinée ci-dessous, composée de cinq éléments a, b, c, d et e peut être représentée de façon ensembliste :

$$\begin{aligned} s &= (\gamma, U) \\ \gamma &= \{\{a, b\}, \{a, c\}, \{b, d\}, \{b, e\}, \{e, d\}, \{d, c\}\} \\ U &= \{a, b, c, d, e\} \end{aligned}$$



La difficulté liée à la définition des structures consiste à définir de façon correcte la notion de construction. Une méthode simple pour décrire une structure consiste à la construire comme étant un assemblage de structures déjà construites. Cette méthode, consistant à privilégier les transformations de structures plutôt que leurs propriétés, est présentée de façon rigoureuse dans [1] et [36]. Nous introduisons la notion de structure élémentaire et de constructeur :

Définition 1.1.2 (Constructeur de structures)

Un *constructeur de structures* est une règle ϕ , qui à partir d'une séquence de s structures o_1, \dots, o_s , construit une nouvelle structure notée $\phi(o_1, \dots, o_s)$. s est appelé l'*arité* du constructeur. La structure o_i est appelée le i^e *opérande* ou la i^e *composante directe* du constructeur ϕ . .

Définition 1.1.3 (Structures atomiques)

Les *structures atomiques* sont les structures de base définies sans constructeur. .

Nous disposons maintenant d'une méthode simple pour construire des structures. Nous pouvons utiliser une approche similaire pour décrire des ensembles de structures :

Définition 1.1.4 (Constructeur d'ensembles de structures)

Un *constructeur d'ensembles de structures* est une règle Φ , qui à partir d'une séquence de s ensembles de structures E_1, \dots, E_s construit un nouvel ensemble de structures :

$$\Phi(E_1, \dots, E_s) \stackrel{\text{def}}{=} \bigcup_{o_1 \in E_1, \dots, o_s \in E_s} \phi(o_1, \dots, o_s).$$

Un *multi-constructeur* est un constructeur d'arité multiple :

$$\Phi(E) \stackrel{\text{def}}{=} \bigcup_{s \in \mathbb{N}} \Phi_s(\underbrace{E, \dots, E}_s) = \bigcup_{\substack{s \geq 0 \\ o_1 \in E_1, \dots, o_s \in E_s}} \Phi_s(o_1, \dots, o_s).$$

Définition 1.1.5 (Ensembles élémentaires de structures)

Les *ensembles élémentaires* de structures sont l'ensemble des structures de *taille* 0, noté *Epsilon*, et l'ensemble des *structures atomiques* de *taille* 1, noté *Atom*. .

Notation. Par la suite nous utilisons le même terme « constructeur » pour désigner les constructeurs de structures ou d'ensemble. Nous noterons en lettres minuscules les constructeurs de structures, et avec la première lettre en majuscule le constructeur d'ensemble associé. .

Nous pouvons maintenant définir la notion de *taille* d'une structure :

Définition 1.1.6 (Fonction taille d'une structure |.|)

La *fonction taille* d'une structure s , élément d'un ensemble de structures, est la fonction à valeurs dans \mathbb{N} définie par les règles suivantes :

- pour tout élément $a \in \text{Epsilon}$: $|a| = 0$,
- pour tout élément $a \in \text{Atom}$: $|a| = 1$,
- pour tout constructeur ϕ appliqué à des structures $o_1, \dots, o_s, s \in \mathbb{N}$: $|\phi(o_1, \dots, o_s)| = \sum_{i=1}^s |o_i|$. .

Une structure combinatoire est simplement une structure à laquelle on associe la fonction permettant de calculer sa *taille* :

Définition 1.1.7 (Ensemble de structures combinatoires)

Un *ensemble E de structures combinatoires* est un ensemble fini ou dénombrable de structures, auquel on associe une fonction $|.|$ permettant de calculer la *taille* des structures de l'ensemble, et tel que pour tout $n \geq 0$, le nombre d'objets de *taille* n dans E est fini. Une structure élément de E est appelée *structure combinatoire*. .

1.1.2 Spécifications

1.1.2.1 Langage, spécification et structures décomposables.

Définition 1.1.8 (Grammaire)

Une *grammaire* est un quadruplet (T, N, E, P) , où T est un ensemble fini de *terminaux*, N un ensemble fini de *non terminaux* ($T \cap N = \emptyset$), E un *symbole d'entrée* élément de l'ensemble N , et P est un ensemble fini de *productions*. Une production portant sur un non terminal A est une règle de la forme :

- $A \rightarrow a$ ou $A \rightarrow B$, avec $A \in N, B \in N$ et $a \in T$. a est un *mot dérivé* de A .
- $A \rightarrow F(A_1, \dots, A_s)$, avec F une *opération*, $s \in \mathbb{N}$, $(A_1, \dots, A_s) \in N \times \dots \times N$. m est un mot dérivé de A s'il est de la forme $m = f(a_1, \dots, a_s)$, avec pour tout $i = 1, \dots, s$, a_i est un mot dérivé de A_i . a_1, \dots, a_s sont appelées les *composantes directes* du mot m . Les *composantes* de m sont ses composantes directes et les composantes de celles-ci. Si a_i est un terminal, a_i est une *composante terminale*. .

Notation. Par la suite, les non terminaux seront notés en lettres majuscules (A), les terminaux en lettres minuscules et en gras (\mathbf{a}).

Définition 1.1.9 (Langage d'une grammaire)

Soit une grammaire $G = (T, N, E, P)$. Le *langage* de la grammaire G est l'ensemble des mots dérivés du symbole d'entrée E . .

Définition 1.1.10 (Spécification d'un ensemble de structures)

Une *spécification* d'un ensemble de structures est un quintuplet (T, N, E, P, Q) , où (T, N, E, P) est une grammaire pour E , et Q des règles d'appartenance aux ensembles élémentaires de structures portant sur chaque terminal. De plus, les opérations dans les règles de production sont des constructeurs, et il existe exactement une règle par non terminal où il est membre gauche. La grammaire est alors de la forme :

$$\begin{aligned}
 T &= \{\mathbf{t}_i, i = 1, \dots, n_T, n_T \in \mathbb{N} \\
 N &= \{N_i, i = 1, \dots, n_N, n_N \in \mathbb{N} \\
 P &= \left\{ N_i \rightarrow \left. \begin{array}{l} \text{ou} \\ \Phi_i(E_1, \dots, E_{s_i}), i = 1, \dots, n_N, s_i \in \mathbb{N}, E_j \in N \cup T, j = 1, \dots, s_i. \end{array} \right\} \right.
 \end{aligned}$$

où Φ_i est un constructeur. Les règles d'appartenance aux ensembles élémentaires de structures sont de la forme :

$$Q = \{\mathbf{t}_i \rightsquigarrow \text{Epsilon}, i \in I_E\} \cup \{\mathbf{t}_j \rightsquigarrow \text{Atom}, j \in I_A \text{ avec } I_E \cap I_A = \emptyset \text{ et } I_E \cup I_A = \{1, \dots, n_T\}\}.$$

Dans la spécification d'un ensemble de structures, les mots dérivés d'un non terminal de la grammaire sont les structures appartenant à l'ensemble des structures associé à ce non terminal. Une structure appartient à l'ensemble de structures d'une spécification, si et seulement si la structure appartient au langage décrit par la grammaire associée à la spécification. En d'autres termes, les mots de la grammaire sont les structures, le langage de la grammaire l'ensemble des structures engendrée par la spécification. On dira qu'une structure s dérivée d'un non terminal A est une structure de l'ensemble A , et on notera $s \in A$. Les définitions portant sur les mots du langage sont étendues aux structures d'une spécification. On pourra par exemple parler des composantes d'une structure.

Notation. La seule donnée des règles de production permet de construire les ensembles T et N sans ambiguïté. N est l'ensemble des symboles apparaissant dans les membres gauches de P , T l'ensemble des symboles apparaissant dans les membres gauches de Q . Par la suite, une spécification sera donnée uniquement par les règles P et Q , et par le symbole d'entrée E . .

Définition 1.1.11 (Ensemble de structures décomposables)

Un *ensemble de structures décomposables* est un ensemble de structures combinatoires qui admet une spécification. .

1.1.2.2 Univers étiqueté et non étiqueté

Nous considérons maintenant deux façons différentes de compter le nombre de structures de taille n . Dans le premier cas, en *univers étiqueté*, nous associons à chaque composante terminale d'une structure, un entier unique compris entre 1 et la taille de la structure. Cet entier est appelé l'*étiquette* de l'élément. Deux éléments atomiques de deux structures différentes sont égaux si et seulement s'ils ont même nom et même étiquette. Dans le second cas, en *univers non étiqueté*, les éléments atomiques ne sont pas étiquetés. Deux éléments atomiques de deux structures différentes sont alors égaux si et seulement s'ils ont même nom.

Exemple. Il n'y a qu'un cycle orienté de longueur trois sur un alphabet d'une lettre en univers non étiqueté ($\text{cycle}(a, a, a)$), et deux en univers étiqueté ($\text{cycle}(a_1, a_2, a_3)$ et $\text{cycle}(a_1, a_3, a_2)$). \cdot

1.1.2.3 Spécifications bien fondées et constructions admissibles

Un des buts de ce chapitre est de compter le nombre de structures ayant une taille donnée, qui appartiennent à un ensemble de structures. Une première étape consiste tout d'abord à ne considérer que les spécifications pour lesquelles ce problème a un sens :

Définition 1.1.12 (Spécification bien fondée)

Une spécification est *bien fondée* si et seulement si pour chaque non terminal A :

1. il existe une structure de taille finie qui dérive de A ,
2. pour tout $n \in \mathbb{N}$, le nombre de structures de taille n est fini. \cdot

Exemple. Considérons la spécification suivante, de symbole d'entrée A et utilisant le constructeur $Prod$, qui forme les couples d'objets, et le constructeur $Union$ qui forme l'union disjointe d'ensembles de structures :

$$\left\{ \begin{array}{l} A \rightarrow Union(\mathbf{a}, B) \\ B \rightarrow Prod(A, A) \\ \mathbf{a} \rightsquigarrow Epsilon. \end{array} \right.$$

En univers étiqueté ou non, une infinité de structures de taille 0 dérivent de cette spécification. Elle n'est donc pas bien fondée d'après le point 2. De même, en univers étiqueté ou non, la spécification suivante, de symbole d'entrée A , n'est pas bien fondée d'après le point 1 :

$$\{ A \rightarrow \Psi(A, A) \}$$

En effet, aucune structure de taille finie ne peut être dérivée de A . \cdot

Nous supposons dans la suite que les spécifications sont bien fondées. Le caractère bien fondé d'une spécification peut être déterminé à l'aide de l'algorithme proposé dans [62].

Définition 1.1.13 (Valuation)

La valuation d'un ensemble de structures décomposables A , notée $valuation(A)$, est la plus petite taille telle qu'une structure dérive de A :

$$valuation(A) = \min_{i \in \mathbb{N}} \text{tel que } \{s \in A, |s| = i\} \neq \emptyset.$$

Exemple. Considérons la spécification suivante :

$$\begin{aligned} A &\rightarrow Prod(C, C, C) \\ C &\rightarrow Union(A, B) \\ B &\rightarrow \mathbf{b} \\ \mathbf{b} &\rightsquigarrow Atom \end{aligned}$$

La valuation du non terminal A est 3, celle des non terminaux B et C est 1. \cdot

Nous supposerons dans la suite que les valuations des non terminaux sont connues. On peut pour cela utiliser l'algorithme décrit dans [62].

Pour permettre de compter de façon simple les structures d'une certaine taille, les constructeurs utilisés dans la spécification doivent vérifier certaines propriétés :

Définition 1.1.14 (Constructeurs admissibles)

La suite de dénombrement associée à un ensemble C de structures combinatoires est la suite $((c)_n)_{n \in \mathbb{N}}$, où $(c)_n$ est le nombre de structures de taille n de C . Un constructeur Φ d'arité s est *admissible* si la suite de dénombrement de $C = \Phi(C_1, \dots, C_s)$ ne dépend que des suites de dénombrement de ses composantes directes C_1, \dots, C_s . ▪

1.1.3 Les classes de spécification Ω et $\hat{\Omega}$

Nous nous proposons maintenant de décrire deux classes particulières de spécifications, la classe Ω et la classe $\hat{\Omega}$. Ces classes étendent les grammaires hors-contexte en ajoutant de nouveaux constructeurs représentant les ensembles et les cycles d'objets.

Définition 1.1.15 (Classes de spécification Ω et $\hat{\Omega}$)

La classe de spécification Ω en univers non étiqueté (respectivement $\hat{\Omega}$ en univers étiqueté), est la classe des spécifications définies à l'aide des constructeurs suivants :

- le constructeur *Union* forme l'union disjointe d'objets : une structure a appartient à l'ensemble de structures $Union(B_1, \dots, B_s)$ si et seulement s'il existe un entier i dans l'intervalle $[1, s]$ tel que a appartient à B_i ,
- le constructeur *Prod* forme les n -uplets d'objets : cela signifie que deux structures $prod(o_1, \dots, o_s)$ et $prod(o'_1, \dots, o'_{s'})$ sont égales si et seulement si $s = s'$ et si pour tout i , $o_i = o'_i$,
- le multi-constructeur *Sequence* forme les séquences d'objets : deux structures $sequence(o_1, \dots, o_s)$ et $sequence(o'_1, \dots, o'_{s'})$ sont égales si et seulement si $s = s'$ et si pour tout i , $o_i = o'_i$. La formulation $A \rightarrow Sequence(B)$ correspond par définition exactement à :

$$\left\{ \begin{array}{l} A = Union(\mathbf{t}, T) \\ \mathbf{t} \rightsquigarrow Epsilon \\ T \rightarrow Prod(B, A) \end{array} \right.$$

- le multi-constructeur *Set* forme les multi-ensembles d'objets (avec répétitions éventuelles en univers non étiqueté) : deux structures $set(o_1, \dots, o_s)$ et $set(o'_1, \dots, o'_{s'})$ sont égales si et seulement si $s = s'$ et s'il existe une permutation σ telle que pour tout i de l'intervalle $[1, s]$, $o_i = o'_{\sigma(i)}$,
- le multi-constructeur *Cycle* forme les cycles orientés d'objets : deux structures $cycle(o_1, \dots, o_s)$ et $cycle(o'_1, \dots, o'_{s'})$ sont égales si et seulement si $s = s'$ et s'il existe t dans l'intervalle $[1, s]$ tel que pour tout $i \in [1, s]$, $o_i = o'_{1+(i+t) \bmod s}$,
- le multi-constructeur *Powerset* (défini uniquement en univers non étiqueté) forme les ensembles d'objets sans répétition : deux structures $powerset(o_1, \dots, o_s)$ et $powerset(o'_1, \dots, o'_{s'})$ sont égales si et seulement si $s = s'$ et s'il existe une permutation σ telle que pour tout i de l'intervalle $[1, s]$, $o_i = o'_{\sigma(i)}$. ▪

Remarque. Il est également possible d'ajouter des conditions portant sur l'arité des multi-constructeurs définis ci-dessus. Nous noterons, pour chaque multi-constructeur Φ , $\Phi(A, card \leq k)$ l'ensemble des structures

$$\phi(a_1, \dots, a_s), s \leq k, a_i \in A, i = 1, \dots, s.$$

1.1.4 Ensembles de structures particuliers

Nous pouvons maintenant définir les spécifications pour des structures combinatoires utilisées couramment.

Définition 1.1.16 (Arbre enraciné)

Un arbre enraciné T est soit une *feuille* \circ , soit construit comme étant une *racine* \blacksquare à laquelle on associe des arbres enracinés T_1, \dots, T_s . Les racines des arbres T_1, \dots, T_s sont les *filles* de \blacksquare , et \blacksquare est le père des racines des arbres T_1, \dots, T_s . Les *nœuds internes* de T sont la racine, ainsi que les nœuds internes des T_1, \dots, T_s . Les arbres pour lesquels le nombre de fils de chaque nœud interne est k , sont les arbres k -aires. Quand il n'y a pas de contrainte sur le nombre de fils, nous parlerons d'arbres généraux. Un arbre dont l'ordre des fils des nœuds internes est distingué est dit *planaire*. \cdot

L'ensemble des arbres binaires (*i.e.* 2-aires) plans peut être décrit à l'aide de la spécification suivante :

$$\begin{cases} B \rightarrow \text{Union}(\mathbf{f}, C) \\ C \rightarrow \text{Prod}(\mathbf{n}, B, B) \\ \mathbf{f} \rightsquigarrow \text{Atom} \\ \mathbf{n} \rightsquigarrow \text{Atom} \end{cases}$$

Dans cette spécification de symbole d'entrée B , \mathbf{f} correspond aux feuilles de l'arbre, et \mathbf{n} aux nœuds internes. La taille d'un arbre correspond ici au nombre de nœuds internes et de feuilles. En modifiant simplement la production $\mathbf{n} \rightsquigarrow \text{Atom}$ en $\mathbf{n} \rightsquigarrow \text{Epsilon}$, on considère que la taille d'un arbre est donnée par son nombre de feuilles.

1.2 Dénombrement

Le problème de dénombrement consiste à déterminer combien il existe de structures différentes ayant une taille donnée. Nous avons vu précédemment que ce nombre dépendait de l'univers, étiqueté ou non étiqueté, dans lequel on se plaçait. Le but de cette section est de présenter les méthodes classiques de dénombrement basées sur l'association de séries génératrices aux constructeurs admissibles [1, 26, 36], et d'en déduire un algorithme efficace de dénombrement dans les cas étiqueté et non étiqueté.

Notation. Dans toute cette section, nous supposons donnée une spécification bien fondée. Nous notons $(a)_n$ le nombre de structures de taille n associées à un non terminal A de cette spécification.

1.2.1 Univers non étiqueté

1.2.1.1 Fonctions génératrices

Nous commençons tout d'abord par introduire la notion classique de série génératrice :

Définition 1.2.1 (Série génératrice ordinaire)

La série génératrice ordinaire associée à un non terminal A est la série :

$$A(z) = \sum_{a \in A} z^{|a|} = \sum_{n=0}^{\infty} (a)_n z^n.$$

\cdot

Il est possible d'associer directement à une spécification un système d'équations portant sur les fonc-

tions génératrices [58]. Les preuves de ces règles peuvent être trouvées dans [62]:

Théorème 1.2.2 (Zimmermann [62])

Dans le cas d'une spécification de Ω , on a les règles suivantes :

$$\begin{array}{l}
 \frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Epsilon}}{A(z) = 1} \qquad \frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Atom}}{A(z) = z} \\
 \frac{A \rightarrow \text{Union}(B_1, \dots, B_s)}{A(z) = \sum_{i=1}^s B_i(z)} \qquad \frac{A \rightarrow \text{Prod}(B_1, \dots, B_s)}{A(z) = \prod_{i=1}^s B_i(z)} \\
 \frac{A \rightarrow \text{Sequence}(B)}{A(z) = \frac{1}{1 - B(z)}} \qquad \frac{A \rightarrow \text{Cycle}(B)}{A(z) = \sum_{k \geq 1} \frac{\phi(k)}{k} \log \frac{1}{1 - B(z^k)}} \\
 \frac{A \rightarrow \text{Set}(B)}{A(z) = \exp\left(\sum_{i=1}^{\infty} \frac{B(z^i)}{i}\right)} \qquad \frac{A \rightarrow \text{Powerset}(B)}{A(z) = \exp\left(\sum_{i=1}^{\infty} (-1)^{i+1} \frac{B(z^i)}{i}\right)}
 \end{array}$$

où ϕ est la fonction d'Euler : $\phi(k) = \text{Card}\{i \leq k, i \text{ premier avec } k\}$.

Remarque. Le caractère bien fondé d'une spécification se traduit simplement en termes des fonctions génératrices associées : une spécification est bien fondée si et seulement si il existe une seule solution ayant des coefficients positifs au système d'équations. .

Exemple (Arbres de Motzkin). Les arbres de Motzkin sont les arbres planaires dont le nombre de fils de chaque nœud interne est soit un, soit deux. Une spécification de ces arbres est :

$$\left\{ \begin{array}{l}
 B \rightarrow \text{Union}(\mathbf{a}, D, E) \\
 D \rightarrow \text{Prod}(\mathbf{a}, B) \\
 E \rightarrow \text{Prod}(\mathbf{a}, B, B) \\
 \mathbf{a} \rightsquigarrow \text{Atom}
 \end{array} \right.$$

En appliquant les règles du théorème 1.2.2, on obtient le système d'équations suivant :

$$\left\{ \begin{array}{l}
 B(z) = a(z) + D(z) + E(z) \\
 D(z) = a(z).B(z) \\
 E(z) = a(z).B(z).B(z) \\
 a(z) = z
 \end{array} \right.$$

On obtient deux solutions de ce système d'équations, dont l'une seulement possède un développement de Taylor à l'origine :

$$B(z) = \frac{1 - z - \sqrt{1 - 2z - 3z^2}}{2z}.$$

Le développement de Taylor à l'origine est de la forme :

$$z + z^2 + 2z^3 + 4z^4 + 9z^5 + 21z^6 + 51z^7 + 127z^8 + 323z^9 + O(z^{10})$$

Il y a donc 323 arbres de Motzkin de taille 9 en univers non étiqueté. .

1.2.1.2 Forme standard

Il n'est pas toujours possible d'obtenir une forme explicite de la solution du système d'équations sur les séries génératrices. Pour résoudre ce problème, P. Zimmermann [62] propose une méthode de calcul des coefficients basée sur des équations récurrentes portant sur les coefficients.

Nous introduisons pour cela l'opérateur Θ de pointage d'un des éléments atomiques d'une structure. Étant donnée une structure, cela revient à distinguer l'une de ses composantes terminales. Cet opérateur se traduit en terme de fonction génératrice par l'opération :

$$\Theta A(z) = z \frac{d}{dz} A(z) = \sum_{i=1}^{\infty} i(a)_i z^i.$$

Nous remarquons que lors de l'application de l'opération Θ , nous perdons l'information sur le nombre de structures dans A de taille nulle. Pour permettre de restituer cette information, et donc de définir l'opération inverse qui consiste à enlever la marque sur un objet, nous n'autorisons l'application de l'opération Θ que lorsque le nombre d'objets de taille zéro est nul. L'opération inverse Θ^{-1} se traduit alors en terme de fonction génératrice :

$$\Theta^{-1} A(z) = \sum_{i=1}^{\infty} \frac{(a)_i}{i} z^i.$$

Pour tenir compte du fait que les composantes des objets en univers non étiqueté peuvent être égales, nous introduisons un nouveau constructeur, le constructeur diagonal. Le constructeur diagonal simple $\Delta^{(k)}(B)$, $k \geq 1$, forme toutes les séquences d'éléments identiques de longueur k , formées d'éléments de B . En terme de fonctions génératrices, on a :

$$A(z) = \Delta^{(k)} B(z) = B(z^k).$$

Étant donnée une suite $u(k)_{k=1}^{\infty}$, le constructeur diagonal généralisé $\Delta_{\{u(k)\}}$ correspond à la série :

$$\Delta_{\{u(k)\}} = \sum_{k=1}^{\infty} u(k) \Delta^{(k)}.$$

Proposition 1.2.3 (Forme standard d'une spécification de Ω (Zimmermann [62]))

Soit S une spécification de Ω , et \underline{S} la spécification, n'utilisant que les constructeurs *Prod* binaire, *Union*, Θ , Θ^{-1} et Δ , obtenue en appliquant les règles suivantes portant sur les règles de production de S :

$$\frac{A \rightarrow \text{Prod}(A_1, \dots, A_s), s > 2}{A \rightarrow \text{Prod}(A_1, T_1), T_1 \rightarrow \text{Prod}(A_2, T_2), \dots, T_{s-2} \rightarrow \text{Prod}(A_{s-1}, A_s)} \quad (1.1)$$

$$\frac{A \rightarrow \text{Sequence}(B)}{A \rightarrow \text{Union}(\mathbf{t}, T_1), \mathbf{t} \rightsquigarrow \text{Epsilon}, T_1 \rightarrow \text{Prod}(B, A)} \quad (1.2)$$

$$\frac{A \rightarrow \text{Set}(B)}{A \rightarrow \text{Union}(\mathbf{t}, T_1), \mathbf{t} \rightsquigarrow \text{Epsilon}, T_1 \rightarrow \Theta^{-1} T_2, T_2 \rightarrow \text{Prod}(A, T_3), T_3 \rightarrow \Delta_{\{1\}} T_4, T_4 \rightarrow \Theta B} \quad (1.3)$$

$$\frac{A \rightarrow \text{Powerset}(B)}{A \rightarrow \text{Union}(\mathbf{t}, T_1), \mathbf{t} \rightsquigarrow \text{Epsilon}, T_1 \rightarrow \Theta^{-1} T_2, T_2 \rightarrow \text{Prod}(A, T_3), T_3 \rightarrow \Delta_{\{(-1)^{k+1}\}} T_4, T_4 \rightarrow \Theta B} \quad (1.4)$$

$$\frac{A \rightarrow \text{Cycle}(B)}{A \rightarrow \Theta^{-1} T_1, T_1 \rightarrow \Delta_{\{\phi(k)\}} T_2, T_2 \rightarrow \text{Prod}(T_3, T_4), T_4 \rightarrow \Theta B, T_3 \rightarrow \text{Union}(\mathbf{t}, T_5), \mathbf{t} \rightsquigarrow \text{Epsilon}, T_5 \rightarrow \text{Prod}(B, T_3)} \quad (1.5)$$

où les \mathbf{t} sont des terminaux, les T_i des non terminaux, différents à chaque application d'une règle. Alors :

- i. les séries génératrices associées à chaque non terminal des spécifications S et \underline{S} sont égales,

ii. dans le cas de spécifications n'utilisant pas le constructeur *Powerset*, les spécifications S et \underline{S} sont équivalentes.

Remarque. Il convient de noter que l'opérateur $\Delta_{\{(-1)^{k+1}\}}$ n'a pas d'interprétation combinatoire. Il ne permet de construire qu'une « pseudo-classe » d'objets combinatoires, les coefficients de la série génératrice associée n'étant pas forcément à valeur positive.

Démonstration.

i. Nous montrons que les séries génératrices des deux spécifications sont égales. Nous détaillons le calcul pour chaque constructeur, bien que la propriété est immédiate d'après le point ii pour les constructeurs autres que *Powerset*.

- règles (1.1) et (1.2) : se déduisent directement de leur définition.
- règle (1.3) : $A \rightarrow \text{Set}(B)$ correspond, d'après le théorème 1.2.2, à l'équation :

$$A(z) = \exp\left(\sum_{i=1}^{\infty} \frac{B(z^i)}{i}\right).$$

Nous avons choisi de n'appliquer l'opérateur Θ que si le nombre de structures de taille nulle est nul. Il n'existe pas de structure de taille zéro dans B , sinon la spécification ne serait pas bien fondée. On pourrait en effet construire une infinité d'ensembles de taille zéro. Par contre, il existe un ensemble de taille nulle dans A , l'ensemble vide, que nous notons *Epsilon*. Nous introduisons le non terminal T_1 comme suit :

$$A = \text{Union}(T_1, \text{Epsilon})$$

T_1 vérifie :

$$\begin{cases} A(z) = 1 + T_1(z) \\ T_1(z) = \exp\left(\sum_{i=1}^{\infty} \frac{B(z^i)}{i}\right) - 1 \end{cases}$$

Nous pouvons maintenant appliquer l'opérateur $\Theta = z \frac{d}{dz}$ à T_1 , ce qui donne :

$$\begin{aligned} \Theta T_1(z) &= A(z) \cdot \sum_{i=1}^{\infty} \left(\frac{d}{dz} B\right)(z^i) \cdot z^i \\ &= A(z) \cdot \sum_{i=1}^{\infty} \Delta^{(i)}(\Theta B(z)) \\ &= A(z) \cdot \Delta_{\{1\}}(\Theta B(z)) \end{aligned}$$

La règle portant sur les constructeurs *Prod* du théorème 1.2.2 permet ensuite d'en déduire la règle proposée.

- la règle (1.4) se démontre de la même façon que la règle (1.3).
- règle (1.5) : $A \rightarrow \text{Cycle}(B)$ correspond, d'après le théorème 1.2.2, à l'équation :

$$A(z) = \sum_{k \geq 1} \frac{\phi(k)}{k} \log \frac{1}{1 - B(z^k)}$$

Il n'y a pas de cycle de taille zéro. On peut appliquer l'opérateur Θ :

$$\begin{aligned} \Theta A(z) &= \sum_{k=1}^{\infty} \phi(k) \frac{1}{1 - B(z^k)} \frac{d}{dz} B(z) \cdot z^k \\ &= \Delta_{\{\phi(k)\}}(\Theta B(z) \cdot \frac{1}{1 - B(z)}) \end{aligned}$$

Le terme $\frac{1}{1 - B(z)}$ correspond, d'après le théorème 1.2.2, à une séquence d'objets de B , ce qui, en appliquant la règle (1.2), conduit à la formulation proposée.

□

ii. Dans le cas où le constructeur *Powerset* n'est pas utilisé, les deux spécifications engendrent des structures équivalentes, au sens où l'on peut définir une bijection entre les structures engendrées par les deux spécifications. La preuve de cette proposition peut être trouvée dans [26].

1.2.1.3 Calcul explicite

D'après la proposition 1.2.3, le problème consistant à compter les structures d'une taille donnée, engendrées par une spécification, peut donc être résolu en comptant les structures engendrées par la spécification standard associée.

Proposition 1.2.4 (Forme explicite des coefficients (Zimmermann [62]))

Les règles suivantes donnent les formes des coefficients des séries génératrices pour les constructeurs utilisés dans la forme standard des spécifications de Ω :

$$\begin{array}{c}
\frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Epsilon}}{(a)_0 = 1, (a)_i = 0, i \neq 0} \\
\frac{A \rightarrow \text{Union}(B_1, \dots, B_s)}{(a)_n = \sum_{i=1}^s (b)_n} \\
\frac{A \rightarrow \Theta B}{(a)_n = n \cdot (b)_n}
\end{array}
\qquad
\begin{array}{c}
\frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Atom}}{(a)_1 = 1, (a)_i = 0, i \neq 1} \\
\frac{A \rightarrow \text{Prod}(B_1, B_2)}{(a)_n = \sum_{k=0}^n (b_1)_k (b_2)_{n-k}} \\
\frac{A \rightarrow \Theta^{-1} B}{(a)_0 = 0, (a)_n = \frac{(b)_n}{n}, n \neq 0} \\
\frac{A \rightarrow \Delta_{\{u(k)\}} B}{(a)_n = \sum_{k|n} u(k) (b)_{\frac{n}{k}}}
\end{array}$$

où $k|n$ signifie que n est divisible par k . .

Corollaire 1.2.5

Les constructeurs de la classe Ω sont admissibles. En effet, pour chaque constructeur, les règles de calcul ci-dessus ne dépendent que du nombre de structures des composantes. .

1.2.1.4 Algorithme de dénombrement

Nous pouvons maintenant décrire l'algorithme proposé par P. Zimmermann [62] permettant de compter le nombre d'objets de taille inférieure à une taille N donnée, pour chaque non terminal A d'une spécification S bien fondée. Le principe consiste à mémoriser les valeurs des coefficients $(a)_i$ calculés. Nous ne calculons ainsi qu'une seule fois la valeur de chaque coefficient. Nous noterons $(a)_i := ?$ les coefficients non encore calculés. Dans le cas du calcul du terme $(a)_n$ des règles de production de la forme $A \rightarrow \text{Prod}(B, C)$, nous avons besoin des valuations de B et C , étant donné que la valeur de B_n peut dépendre de A_n . De même nous vérifions que l'application du constructeur Θ^{-1} se fait bien sur des structures pointées.

Algorithme

- *Données* : un entier N , une spécification bien fondée sous forme standard de Ω ,
- *Résultat* : le nombre de structures de taille i de chaque non terminal de la spécification, pour tout i inférieur ou égal à N .

```

PROCEDURE compte_tous(N: INTEGER)
  FOR ALL non terminal A ∈ S DO
    (a)i := ?, i = 0, ..., N
  END_FOR;
  FOR n FROM 0 TO N DO
    FOR ALL non terminal A ∈ S DO
      compte(A, n);
    END_FOR;
  END_FOR;

```

```

END_PROCEDURE;

FUNCTION compte(A: nonterminal, n: INTEGER) : INTEGER
  IF (a)n =? THEN
    IF (A → a, a ∼ Atom) ∈ S THEN
      IF n = 1 THEN (a)n := 1; ELSE (a)n := 0; END_IF;
    ELSE_IF (A → a, a ∼ Epsilon) ∈ S THEN
      IF n = 0 THEN (a)n := 1; ELSE (a)n := 0; END_IF;
    ELSE_IF (A → Union(B1, ..., Bk)) ∈ S THEN
      s := 0;
      FOR i FROM 1 TO k DO
        s := s + compte(Bi, n);
      END_FOR;
      (a)n := s;
    ELSE_IF (A → Prod(B, C)) ∈ S THEN
      s := 0;
      FOR k FROM valuation(B) TO n - valuation(C) DO
        s := s + compte(B, k) * compte(C, n - k);
      END_FOR;
      (a)n := s;
    ELSE_IF (A → ΘB) ∈ S THEN
      IF compte(B, 0) ≠ 0 THEN ERROR("LA SPÉCIFICATION N'EST PAS VALIDE"); END_IF;
      (a)n := n * compte(B, n);
    ELSE_IF (A → Θ-1(B)) ∈ S THEN
      IF n = 0 THEN
        (a)n := 0;
      ELSE_IF compte(B, n) mod n ≠ 0 THEN
        ERROR("LA SPÉCIFICATION N'EST PAS VALIDE");
      ELSE
        (a)n := compte(B, n) / n;
      END_IF;
    ELSE_IF (A → Δ{u(k)}(B)) ∈ S THEN
      s := 0;
      FOR k FROM 1 TO n DO
        IF n mod k = 0 THEN
          s := s + u(k) * compte(B, n/k);
        END_IF;
      END_FOR;
      (a)n := s;
    END_IF;
  END_IF;
  RETURN (a)n;
END_FUNCTION;

```

Le coût de dénombrement de n coefficients, si l'on considère que le coût de la multiplication de deux entiers est constant, est en $O(n^2)$. En pratique, nous faisons des calculs sur de grands entiers, et la croissance des coefficients doit être prise en compte. La taille des coefficients est en $O(n)$, et le coût « réel » de l'algorithme est en $O(n^4)$.

1.2.2 Univers étiqueté

1.2.2.1 Fonctions génératrices

Définition 1.2.6 (Série génératrice exponentielle)

La série génératrice exponentielle associée à un non terminal A est la série :

$$\hat{A}(z) = \sum_{a \in A} \frac{z^{|a|}}{|a|!} = \sum_{n=0}^{\infty} (a)_n \frac{z^n}{n!}.$$

Comme dans le cas non étiqueté, il est possible d'associer à une spécification un système d'équations portant sur les fonctions génératrices. La preuve de ces règles peut être trouvée dans [62] :

Théorème 1.2.7 (Zimmermann [62])

Dans le cas d'une spécification de $\hat{\Omega}$, on a les règles suivantes :

$$\begin{array}{c}
\frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Epsilon}}{\hat{A}(z) = 1} \\
\frac{A \rightarrow \text{Union}(B_1, \dots, B_s)}{\hat{A}(z) = \sum_{i=1}^s \hat{B}_i(z)} \\
\frac{A \rightarrow \text{Sequence}(B)}{\hat{A}(z) = \frac{1}{1 - \hat{B}(z)}} \\
\frac{A \rightarrow \text{Set}(B)}{\hat{A}(z) = \exp(\hat{B}(z))} \\
\frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Atom}}{\hat{A}(z) = z} \\
\frac{A \rightarrow \text{Prod}(B_1, \dots, B_s)}{\hat{A}(z) = \prod_{i=1}^s \hat{B}_i(z)} \\
\frac{A \rightarrow \text{Cycle}(B)}{\hat{A}(z) = \log \frac{1}{1 - \hat{B}(z)}}
\end{array}$$

Remarque. En univers étiqueté, le constructeur *Prod* correspond au produit partitionnel.

1.2.2.2 Forme standard

Nous introduisons, comme dans le cas non étiqueté, l'opérateur Θ de pointage d'un des éléments atomiques d'une structure. Cet opérateur se traduit en terme de fonction génératrice par l'opération :

$$\Theta \hat{A}(z) = z \frac{d}{dz} \hat{A}(z) = \sum_{i=1}^{\infty} i \frac{(a)_i}{i!} z^i.$$

Nous posons les mêmes conditions d'application que dans le cas étiqueté, à savoir que le terme correspondant aux objets de taille nulle de A doit être nul pour pouvoir lui appliquer l'opérateur Θ . L'opération inverse Θ^{-1} est alors définie par :

$$\Theta^{-1} \hat{A}(z) = \sum_{i=1}^{\infty} \frac{(a)_i}{i!} z^i = \int_0^z \frac{\hat{A}(t) dt}{t}.$$

Proposition 1.2.8 (Forme standard d'une spécification de $\hat{\Omega}$ (Zimmermann [62]))

Soit \hat{S} une spécification de $\hat{\Omega}$, et \underline{S} la spécification, n'utilisant que les constructeurs *Prod* binaire, *Union*, Θ et Θ^{-1} , obtenue en appliquant les règles suivantes portant sur les règles de production de \hat{S} :

$$\frac{A \rightarrow \text{Prod}(A_1, \dots, A_s), s > 2}{A \rightarrow \text{Prod}(A_1, T_1), T_1 \rightarrow \text{Prod}(A_2, T_2), \dots, T_{s-2} \rightarrow \text{Prod}(A_{s-1}, A_s)} \quad (1.6)$$

$$\frac{A \rightarrow \text{Sequence}(B)}{A \rightarrow \text{Union}(\mathbf{t}, T_1), \mathbf{t} \rightsquigarrow \text{Epsilon}, T_1 \rightarrow \text{Prod}(B, A)} \quad (1.7)$$

$$\frac{A \rightarrow \text{Set}(B)}{A \rightarrow \text{Union}(\mathbf{t}, T_1), \mathbf{t} \rightsquigarrow \text{Epsilon}, T_1 \rightarrow \Theta^{-1} T_2, T_2 \rightarrow \text{Prod}(A, T_3), T_3 \rightarrow \Theta B} \quad (1.8)$$

$$\frac{A \rightarrow \text{Cycle}(B)}{A \rightarrow \Theta^{-1} T_1, T_1 \rightarrow \text{Prod}(T_2, T_3), T_3 \rightarrow \Theta B, T_2 \rightarrow \text{Union}(\mathbf{t}, T_4), \mathbf{t} \rightsquigarrow \text{Epsilon}, T_4 \rightarrow \text{Prod}(B, T_2)} \quad (1.9)$$

où les \mathbf{t} représentent des terminaux, les T_i des non terminaux, différents à chaque application d'une règle. Alors les spécifications S et \underline{S} sont équivalentes.

Démonstration.

La démonstration se fait selon les mêmes arguments que dans le cas non étiqueté [26]. \square

D'après la proposition 1.2.8, le problème consistant à compter les structures d'une taille donnée, engendrées par une spécification, peut donc être résolu en comptant les structures engendrées par la spécification standard associée.

1.2.2.3 Calcul explicite

À partir de la spécification standard, on peut déduire simplement les coefficients des séries génératrices associées à chaque non terminal.

Proposition 1.2.9 (Forme explicite des coefficients (Zimmermann [62]))

Les règles suivantes donnent les formes des coefficients des séries génératrices pour les constructeurs utilisés dans la forme standard des spécifications de $\hat{\Omega}$:

$$\frac{\frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Epsilon}}{(a)_0 = 1, (a)_i = 0, i \neq 0}}{A \rightarrow \text{Union}(B_1, \dots, B_s)} \quad \frac{\frac{A \rightarrow \mathbf{a}, \mathbf{a} \rightsquigarrow \text{Atom}}{(a)_1 = 1, (a)_i = 0, i \neq 1}}{A \rightarrow \text{Prod}(B_1, B_2)}}{\frac{(a)_n = \sum_{i=1}^s (b_i)_n}{A \rightarrow \Theta B} \quad \frac{(a)_n = \sum_{k=0}^n \binom{n}{k} (b_1)_k (b_2)_{n-k}}{A \rightarrow \Theta^{-1} B}}{(a)_n = n(b)_n \quad (a)_0 = 0, (a)_n = \frac{(b)_n}{n}, n \neq 0}$$

Remarque. Le nombre de réétiquetages compatibles avec l'ordre initial entre deux objets de taille i et j est $\frac{(i+j)!}{i!j!} = \binom{i+j}{j}$, ce qui conduit au terme $\binom{n}{k}$ dans le cas de l'opérateur *Prod*.

Corollaire 1.2.10

Les constructeurs de la classe $\hat{\Omega}$ sont admissibles.

1.2.2.4 Algorithme de dénombrement

L'algorithme de dénombrement est similaire au cas non étiqueté, avec simplement des règles de dénombrement des coefficients différentes :

Algorithme

- *Données* : un entier n , une spécification sous forme standard S de $\hat{\Omega}$,
- *Résultat* : le nombre d'objets de taille inférieure ou égale à N pour les non terminaux de la spécification S .

```

FUNCTION compte(A: nonterminal, n: INTEGER) : INTEGER
  IF (a)_n =? THEN
    IF (A → a, a ∼ Atom) ∈ S THEN
      IF n = 1 THEN (a)_n := 1; ELSE (a)_n := 0; END_IF;
    ELSE_IF (A → a, a ∼ Epsilon) ∈ S THEN
      IF n = 0 THEN (a)_n := 1; ELSE (a)_n := 0; END_IF;
    ELSE_IF (A → Union(B1, ..., Bk)) ∈ S THEN
      s := 0;
      FOR i de 1 à k DO
        s := s + compte(Bi, n);
      END_FOR;
      (a)_n := s;
    ELSE_IF (A → Prod(B, C)) ∈ S THEN
      s := 0;
  
```



```

FOR k FROM valuation(B) TO n - valuation(C) DO
  s := s +  $\binom{n}{k}$  * compte(B, k) * compte(C, n - k);
END_FOR;
(a)n := s;
ELSE_IF (A → ΘB) ∈ S THEN
  IF compte(B, 0) ≠ 0 THEN ERROR("LA SPÉCIFICATION N'EST PAS VALIDE"); END_IF;
  (a)n := n * compte(B, n);
ELSE_IF (A → Θ-1B) ∈ S THEN
  IF n = 0 THEN
    (a)n := 0;
  ELSE_IF compte(B, n) mod n ≠ 0 THEN
    erreurla spécification n'est pas valide;
  END_IF;
  (a)n := compte(B, n) / n;
END_IF;
END_IF;
RETURN (a)n;
END_FUNCTION;

```

1.2.3 Solutions particulières

Dans le cas de spécifications ne contenant que les constructeurs *Prod* et *Union*, les séries génératrices associées aux non terminaux sont holonomes. Il est alors possible de décrire les coefficients de la série génératrice sous forme d'une équation récurrente. Nous pouvons pour cela utiliser la bibliothèque **Maple GFUN**, écrite par B. Salvy et P. Zimmermann [47].

Exemple. Reprenons l'exemple des arbres de Motzkin. Nous avons vu que la fonction génératrice associée vérifiait l'équation :

$$B(z) = z + z.B(z) + z.B(z)^2$$

GFUN permet de convertir cette équation en équation différentielle, puis d'obtenir l'équation récurrente sur les coefficients :

```

> with(gfun);
> algeqtodiffeq(B=z*(1+B+B^2),B(z));

```

$$2z^2 + (-1 + z)B(z) + (-z + 2z^2 + 3z^3)D(B)(z), B(0) = 0$$

```

> diffeqtorec("B(z),b(n));

```

$$b(0) = 0, 3nb(n) + (3 + 2n)b(1 + n) + (-3 - n)b(n + 2), b(1) = 1$$

On peut vérifier que l'on obtient bien le même résultat que précédemment :

```

> p:=rectoproc("b(n)");
> seq(p(i),i=0..9);
0, 1, 1, 2, 4, 9, 21, 51, 127, 323

```

1.2.4 Mise en œuvre

L'algorithme de dénombrement de structures décomposables, présenté dans ce chapitre, est utilisé dans la bibliothèque **comstruct**, intégrée au système de calcul formel **Maple**. La première version de cette bibliothèque, appelée **Gaia**, a été réalisée par P. Zimmermann [63], la version actuelle étant maintenue par E. Murray.

Nous illustrons l'utilisation de la bibliothèque **comstruct** sur l'exemple suivant. Nous souhaitons compter le nombre d'arbres généraux non planaires de taille 500, en univers non étiqueté. Une spécification

de ces arbres, en termes de structures décomposables, est la suivante :

$$\left\{ \begin{array}{l} A \rightarrow \text{Prod}(N, B) \\ B \rightarrow \text{Set}(A) \\ N \rightarrow \mathbf{n} \\ \mathbf{n} \rightsquigarrow \text{Atom} \end{array} \right.$$

En **Maple**, nous définissons la spécification, que nous notons **spec**, de la façon suivante :

```
> spec:={A=Prod(n,B), B=Set(A), n=Atom};
```

Pour compter le nombre de structures de taille 500, nous utilisons ensuite la fonction **count** de la bibliothèque **combstruct**. Les instructions **time** permettent de connaître le temps CPU, en secondes, utilisé pour l'exécution de l'instruction **count**. Les tests de cette section sont effectués sur machine Sun Ultra Sparc 1.

```
> with(combstruct);                # charge la bibliothèque combstruct
> t1:=time();
> count([A,spec,unlabelled],size=500);
```

```
851104420578030830171766543802662283351533556217730718125750010602468532453176\
706329189027371547883574966253369264258013782888526879605945287645314671717388\
220308429708770104103735878788902078937174928693455204507540870849529634520
```

```
> time()-t1;
137.333
```

Il faut plus de deux minutes pour compter le nombre de structures de taille 500 sur cet exemple. Il faut plus de 20 minutes de temps CPU pour obtenir le nombre de structures de taille 1000. Cela illustre le coût en $O(n^4)$ de l'algorithme de dénombrément, du fait de la croissance des coefficients. Pour permettre de compter le nombre de structures pour de plus grandes tailles en un temps raisonnable, il devient indispensable de diminuer ces temps de calcul.

Pour diminuer le temps de calcul nécessaire, nous avons réalisé une bibliothèque **Maple**, appelée **Ccombstruct**, qui permet de générer automatiquement, à partir d'une spécification quelconque, un programme écrit en langage **C** qui répond au problème. Le problème de dénombrément nécessite de pouvoir manipuler de grands entiers. Pour ce faire, le programme généré automatiquement utilise la bibliothèque multi-précision **GMP** [33].

Regardons l'utilisation la bibliothèque **Ccombstruct**, dans le cas de l'exemple précédent. La définition de la spécification est identique. Nous utilisons ensuite la fonction **create_count_file** pour générer le fichier **npgtree.c** :

```
> spec:={A=Prod(n,B), B=Set(A), n=Atom};
> create_count_file(A, spec, 'npgtree.c');
```

La génération et la compilation du fichier **C** s'effectue en moins de deux secondes de temps CPU.

```
% gcc -O npgtree.c -DGMP -o npgtree -lgmp
```

On peut ensuite obtenir le nombre de structures cherché :

```
% npgtree 500
A(500)=85110442057803083017176654380266228335153355621773071812575001060246853
245317670632918902737154788357496625336926425801378288852687960594528764531467
171738822030842970877010410373587878890207893717492869345520450754087084952963
4520;
```

Pour comparer le gain apporté par l'utilisation de programmes **C** au lieu de **Maple**, nous pouvons nous référer aux mesures suivantes, effectuées dans le cas des arbres généraux non planaires, résumées sur les figures 1.1 et 1.2. L'accélération correspond au rapport du temps de la version **Maple** et de la version **C**, en tenant compte des temps de génération et de compilation.

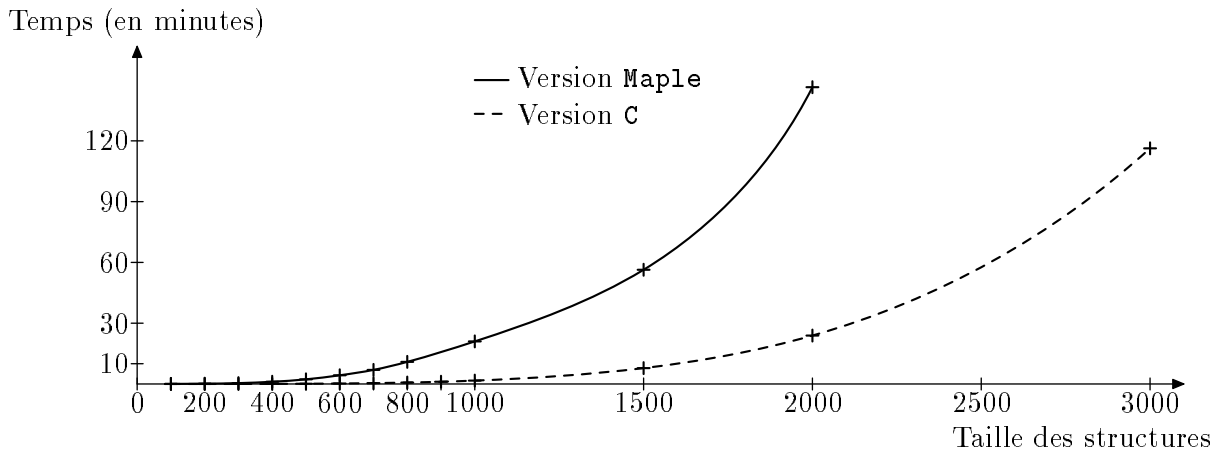


Fig. 1.1: Dénombrement du nombre d'arbres généraux non planaires : comparaison des temps de calcul entre le programme `C` généré par `Ccombstruct` et la version `Maple-combstruct`.

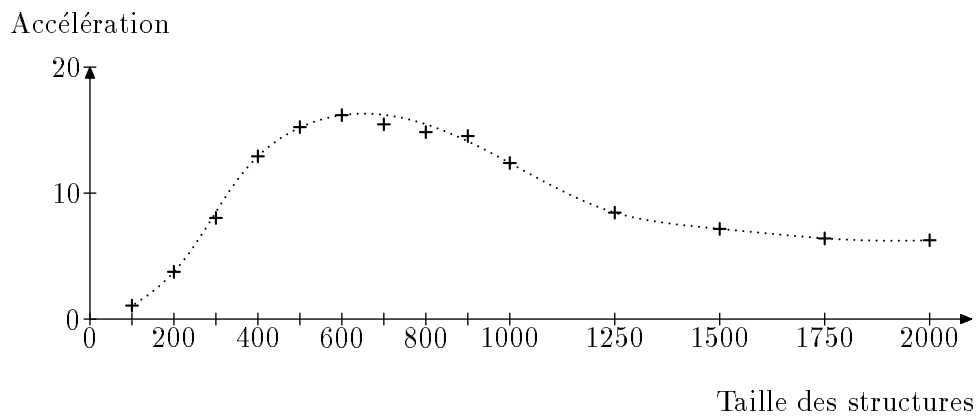


Fig. 1.2: Dénombrement du nombre d'arbres généraux non planaires : accélération du programme `C` généré par `Ccombstruct`, par rapport à la version `Maple-combstruct`.

Avec `combstruct`, le calcul n'aboutit pas, pour des raisons de place mémoire, pour des structures de taille 3000. On remarque une baisse relative de l'efficacité, pour des tailles de structures supérieures à 600, qui finit par se stabiliser autour d'un facteur 6. Cette baisse tient sans doute au fait que `Maple` est un système interprété. `Maple` est pénalisé (boucles `for`) pour de petites tailles, mais l'est moins quand n devient grand du fait de la croissance des coefficients et de l'utilisation par `Maple` d'une arithmétique compilée. Cette baisse d'efficacité se retrouve également sur des programmes de tests qui n'effectuent que des multiplications au sein d'une boucle `for`, sans allocation ni libération de mémoire.

Si l'on souhaite obtenir uniquement une approximation du nombre de structures, il est possible d'utiliser des calculs approchés. Étant donnée la taille des nombres manipulés, les nombres flottants machine (`double` ou `long double`) ne suffisent pas. Nous choisissons donc de représenter un entier n sous la forme $n = m \times 10^e$, avec m un nombre flottant compris entre 1 et 10, et e un entier compris entre 0 et 2^{31} . On obtient alors les temps de calcul indiqués sur la figure 1.3.

Taille	Temps version C approché(en s)	Temps C + gmp	Temps version Maple (en s)	Accélération
100	0.04	0.07	2.22	
500	1.26	7.01	137.33	108.99
1000	11.39	99.74	1260.65	110.68
2000	69.58	1400.94	8789.48	126.32
3000	206.42	6978.40	–	–
5000	861.37	42189.05	–	–

Fig. 1.3: Dénombrement du nombre d'arbres généraux non planaires : mesure des temps de calcul des versions C avec calcul approché, et `Maple`

Dans le cas où la solution peut être trouvée sous la forme d'une équation récurrente simple, `Ccombstruct` permet de générer directement le code C :

```
create_count_file_proc_rec(A, a(0)=0, 3*n*a(n)+(3+2*n)*a(1+n)+(-3-n)*a(n+2), a(1)=1, a(n),
    'npgtree_rec.c');
```

1.3 Génération aléatoire uniforme

Dans cette section, nous nous intéressons au problème de la génération aléatoire uniforme de structures décomposables. Le principe de l'algorithme de génération aléatoire est basé sur la spécification standard des ensembles de structures. À chaque non terminal A de cette spécification, on associe une fonction `gener_A`, qui génère de façon aléatoire uniforme une structure de A de la taille cherchée. Cette fonction peut utiliser les fonctions de dénombrement de la section précédente, ainsi que les fonctions de génération d'autres non terminaux.

1.3.1 Principe

1.3.1.1 Univers non étiqueté

1.3.1.1.1 Constructeur *Prod*. Regardons le cas d'une spécification standard contenant une règle de production de la forme $A \rightarrow Prod(B, C)$. Pour générer une structure de taille n , de façon uniforme parmi toutes les structures appartenant à l'ensemble A , on procède comme suit. Le nombre de structures de taille n dans A est donné par la formule :

$$(a)_n = \sum_{k=0}^n (b)_k \times (c)_{n-k}.$$

Nous définissons la probabilité p_k pour une structure de A d'avoir une première composante de taille k . On a alors :

$$p_k = \frac{(b)_k \times (c)_{n-k}}{(a)_n}.$$

L'algorithme de génération aléatoire, dans le cas du constructeur *Prod*, consiste à choisir un nombre réel aléatoire x , selon une loi uniforme, dans l'intervalle $[0, 1]$. Ensuite, on cherche k minimum tel que $\sum_{i=0}^k p_i \geq x$. Cela détermine la taille des composantes de la structure. On appelle ensuite récursivement le procédé sur chaque composante. La figure 1.4 illustre ce principe dans le cas du constructeur *Prod*.

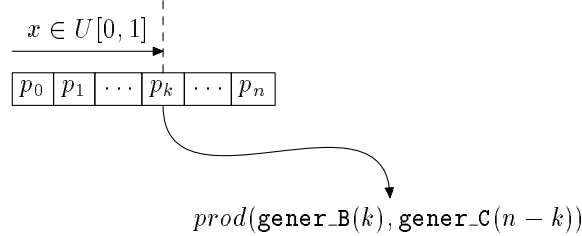


Fig. 1.4: Principe de l'algorithme de génération aléatoire uniforme dans le cas d'une production de la forme $A \rightarrow Prod(B, C)$

L'algorithme suivant utilise la fonction `uniforme(a, b)`, qui retourne un entier aléatoire de l'intervalle $[a, b]$. Chercher k minimum tel que $\sum_{i=0}^k p_i \geq x$, pour $x \in [0, 1]$, revient à chercher k minimum tel que $\sum_{i=0}^k (b)_k \times (c)_{n-k} \geq x'$, pour $x' \in [1, (a)_n]$.

Algorithme (Génération de structures de l'ensemble A défini par $A \rightarrow Prod(B, C)$)

- *Données* : la taille n de la structure à générer aléatoirement,
- *Résultat* : une structure de l'ensemble A .

```

FUNCTION gener_A(n: INTEGER)
  x:=uniforme(1,compte(A,n));
  k:=-1;
  s:=0
  WHILE s < x DO
    k:=k+1;
    p_k:=compte(B,k)*compte(C,n-k);
    s:=s+p_k;
  END_WHILE;
  RETURN prod(generator_B(k),generator_C(n-k));
END_FUNCTION;

```

Si l'on note $T_D(n)$ le temps maximal mis pour générer une structure de taille n appartenant à un ensemble D , le temps d'exécution dans le cas le pire de l'algorithme précédent, en comptant un coût unitaire pour chaque exécution des instructions de la boucle `while`, conduit à l'équation :

$$T_A(n) = \max_{0 \leq k \leq n} (k + 1 + T_B(k) + T_C(n - k))$$

En posant $T(n) = \max_A T_A(n)$, on obtient comme complexité pour l'algorithme de génération :

$$T(n) = O(n^2).$$

Il est possible d'améliorer l'algorithme précédent en utilisant une méthode de recherche dite *Boustrophédon* [26]. Au lieu de chercher simplement k minimum tel que $x \leq \sum_{i=0}^k p_i$, on cherche simultanément le k' maximum tel que $1 - x \leq \sum_{i=n-k'}^n p_i$. On arrête la recherche dès que l'une des deux conditions cherchées est vérifiée. L'algorithme devient alors :

```

FUNCTION gener_A(n: INTEGER)
  x:=uniforme(1,compte(A,n));

```

```

k:=-1;
s:=0;
trouve:=FALSE;
WHILE NOT trouve DO
  k:=k+1;
  pk:=compte(B,k)*compte(C,n-k);
  s:=s+pk;
  trouve:=s≥x;
  IF NOT trouve THEN
    pk:=compte(B,n-k)*compte(C,k);
    s:=s+pk;
    trouve:=s≥x;
    IF trouve THEN
      k:=n-k;
    END_IF
  END_IF;
END_WHILE;
RETURN prod(gener_B(k),gener_C(n-k));
END_FUNCTION;

```

Le temps d'exécution vérifie alors l'équation de récurrence suivante :

$$T(n) = \max_k [2 \cdot \min(n-k, k) + 2 + T(k) + T(n-k)]$$

On obtient la complexité asymptotique suivante pour l'algorithme Boustrophédon [26] :

$$T_A(n) = O(n \cdot \log n).$$

1.3.1.1.2 Constructeur Union. La fonction de génération pour les règles de production de la forme $A \rightarrow \text{Union}(B, C)$ est :

```

FUNCTION gener_A(n: INTEGER)
  x:=uniforme(1,compte(A,n));
  IF x < compte(b,n) THEN
    RETURN gener_B(n);
  ELSE
    RETURN gener_C(n);
  END_IF
END_FUNCTION

```

1.3.1.1.3 Constructeur Θ . On note $\theta(s, k)$ le fait de marquer le k^e terminal de la structure s . L'ordre sur les terminaux appartenant à une structure peut être choisi arbitrairement comme étant par exemple l'ordre d'apparition des terminaux lors de l'écriture de la structure. Pour une structure s de taille n , il y a n marquages possibles de la structure. On en déduit la fonction de génération suivante pour $A = \Theta B$:

```

FUNCTION gener_A(n: INTEGER)
  x:=uniforme(1,n);
  RETURN  $\theta(\text{gener}_B(n), x)$ ;
END_FUNCTION;

```

1.3.1.1.4 Constructeur Θ^{-1} . Le constructeur Θ^{-1} est défini comme le constructeur qui permet d'enlever la marque sur une structure marquée. Il vérifie la propriété, pour toute structure s et tout entier k :

$$\theta^{-1}(\theta(s, k)) = s$$

La fonction de génération s'écrit donc directement, pour $A = \Theta^{-1}B$:

```
FUNCTION gener_A(n: INTEGER)
  RETURN  $\theta^{-1}$  (gener_B(n));
END_FUNCTION;
```

1.3.1.1.5 Constructeur Δ . On procède de la même façon que dans le cas du produit. On cherche la taille que doivent avoir les composantes de la structure que l'on génère en fonction d'un entier choisi aléatoirement. Si $A \rightarrow \Delta_{\{u(k)\}}(B)$, on a :

```
FUNCTION gener_A(n: INTEGER)
  x:=uniforme(1,compte(A,n));
  k:=1;
  s:=u(k)*compte(B, $\frac{n}{k}$ );
  WHILE s < x DO
    k:=k+1; WHILE n mod k  $\neq$  0 DO k:=k+1; END_WHILE;
    s:=s+u(k)*compte(B, $\frac{n}{k}$ );
  END_WHILE;
  b:=gener_B( $\frac{n}{k}$ );
  RETURN  $\underbrace{(b, \dots, b)}_k$ ;
END_FUNCTION;
```

1.3.1.2 Univers étiqueté

Le principe de l'algorithme est exactement le même que dans le cas non étiqueté. Les algorithmes sont légèrement modifiés pour tenir compte des différences de dénombrement. Regardons le cas du constructeur *Prod*. L'algorithme de génération Boustrophédon devient :

```
FUNCTION gener_A(n: INTEGER)
  x:=uniforme(1,compte(A,n));
  k:=-1;
  s:=0;
  trouve:=FALSE;
  WHILE NOT trouve DO
    k:=k+1;
    pk :=  $\binom{n}{k}$ *compte(B,k)*compte(C,n-k);
    s:=s+pk;
    trouve := s  $\geq$  x;
  IF NOT trouve THEN
    pk :=  $\binom{n}{k}$ *compte(B,n-k)*compte(C,k);
    s:=s+pk;
    trouve := s  $\geq$  x;
  IF trouve THEN
    k:=n-k;
  END_IF
  END_IF;
  END_WHILE;
  RETURN prod(gener_B(k),gener_C(n-k));
END_FUNCTION;
```

Dans le cas étiqueté, nous devons tenir compte des étiquettes à ajouter aux terminaux de la structure. Nous ajoutons simplement les étiquettes en fin de génération, de façon aléatoire [26].

1.3.2 Mise en œuvre

Cet algorithme de génération aléatoire est utilisé dans la bibliothèque **combstruct**, présentée en 1.2.4. Nous pouvons illustrer l'algorithme dans le cas de l'exemple des arbres généraux non planaires présenté en 1.2.4 :

```
> with(combstruct):
```

```
> spec:={A=Prod(n,B), B=Set(A), n=Atom}:
> draw([A,spec,unlabelled],size=10);
```

```
Prod(N,Set(Prod(N,Set(Prod(N,Epsilon),Prod(N,Set(Prod(N,Epsilon),Prod(N,
Epsilon))),Prod(N,Epsilon))),Prod(N,Set(Prod(N,Epsilon),Prod(N,Epsilon))))
```

De la même façon que pour le problème de dénombrement, le programme `Cconstruct` permet de créer des programmes en langage `C` permettant de générer de façon aléatoire uniforme des structures, étant donnée leur spécification. Il est possible d'utiliser soit les entiers longs et faire du calcul exact, soit d'effectuer des calculs approchés. En pratique, il devient ainsi possible de générer en temps raisonnable de grandes structures de façon quasi-uniforme.

L'arbre de Motzkin de la figure 1.5 est de taille 1000, et a été généré de façon quasi-uniforme en 10 secondes, sachant que les coefficients sont calculés sans utiliser la solution particulière utilisant des équations récurrentes :

1.4 Extensions

Il est possible d'apporter quelques modifications simples aux méthodes de dénombrement et de génération aléatoire que nous avons présentées, sans changer leur complexité. Ces modifications ne changent pas le principe général des algorithmes précédents, mais permettent simplement de changer la notion de taille des structures. La taille d'une structure n'est plus forcément égale à la somme des tailles de ses composantes, et se rapproche de ce point de vue des techniques utilisées dans le cadre des grammaires d'objets [14].

1.4.1 Nouveaux constructeurs

Définition 1.4.1 (Constructeur Δ_f)

Le constructeur Δ_f est un constructeur qui modifie la notion de taille d'une structure. $f : \mathbb{N} \rightarrow \mathbb{N}$ est appelée la *fonction taille* du constructeur. ▪

Définition 1.4.2 (Constructeur $Prod_{(f,c)}$)

Le constructeur $Prod_{(f,c)}$ forme les couples de structures. L'ensemble de structures $A = Prod_{(f,c)}(B_1, B_2)$ est l'ensemble formé par les structures de la forme :

$$a = prod_{(f,c)}(b_1, b_2), b_1 \in B_1, b_2 \in B_2,$$

avec la condition $c(|b_1|, |b_2|)$ vérifiée. $f : \mathbb{N} \rightarrow \mathbb{N}$ est appelée la *fonction taille* du constructeur, c la *fonction de condition* du constructeur. ▪

La fonction taille d'une structure est redéfinie comme suit pour tenir compte des nouveaux constructeurs introduits :

Définition 1.4.3 (Fonction taille d'une structure $| \cdot |$)

La *fonction taille* d'une structure s , élément d'un ensemble de structures d'une spécification, est la fonction à valeurs dans \mathbb{N} définie par les règles suivantes :

- pour tout élément $a \in Epsilon$: $|a| = 0$,
- pour tout élément $a \in Atom$: $|a| = 1$,
- pour tout constructeur Δ_f appliqué à une structure o , $|\delta_f(o)| = f(|o|)$,
- pour tout constructeur $prod_{(f,c)}$ appliqué à des structures o_1 et o_2 , $|prod_{(f,c)}(o_1, o_2)| = f(|o_1|, |o_2|)$,
- pour tout constructeur ϕ , différent des constructeurs Δ_f et $Prod_{(f,c)}$, appliqué à des structures o_1, \dots, o_s , $s \in \mathbb{N}$: $|\phi(o_1, \dots, o_s)| = \sum_{i=1}^s |o_i|$. ▪

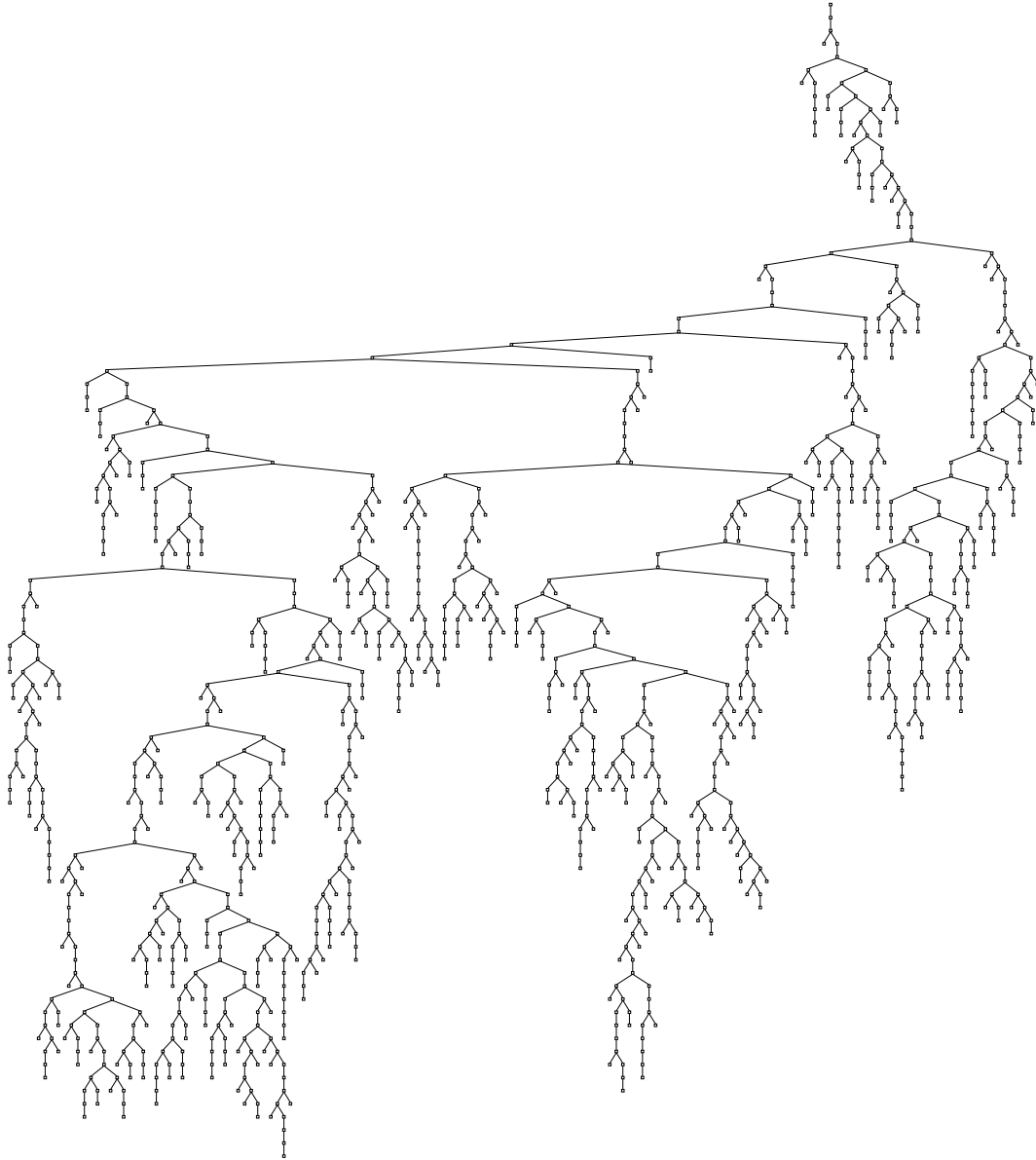


Fig. 1.5: *Arbre de Motzkin de taille 1000 généré aléatoirement.*

1.4.2 Dénombrement

1.4.2.1 Constructeur Δ_f

Nous nous limitons au cas où il est possible de définir deux fonctions

$$\begin{cases} a_f : \mathbb{N} \rightarrow \mathbb{N}, \\ b_f : \mathbb{N} \rightarrow \mathbb{N}, \end{cases}$$

telles que l'on ait :

$$\frac{A = \Delta_f(B)}{(a)_n = \sum_{\substack{n=f(k) \\ k \geq 0}} (b)_k = \sum_{k=a_f(n)}^{b_f(n)} (b)_k \cdot \chi_{(f(k)=n)}},$$

où $\chi_{(a=b)}$ vaut 1 si $a = b$, et 0 sinon. a_f , b_f et g_f sont appelées les *fonctions de calcul* du constructeur. Quand $b(n) - a(n) = O(n)$, la complexité globale de l'algorithme de dénombrement n'est pas modifiée.

Remarque. On a $\Delta_{\{1\}} = \Delta_f$ si $\begin{cases} a_f : x \mapsto 0 \\ b_f : x \mapsto x \\ f : x \mapsto n + n \pmod{x} \end{cases}$.

1.4.2.2 Constructeur $Prod_{(f,c)}$

Dans le cas du constructeur $Prod_{(f,c)}$, nous nous limitons au cas où nous pouvons déterminer, pour des fonctions f et c données, les trois fonctions suivantes, appelées les *fonctions de calcul* du constructeur :

$$\begin{cases} a_{(f,c)} : \mathbb{N} \rightarrow \mathbb{N}, \\ b_{(f,c)} : \mathbb{N} \rightarrow \mathbb{N}, \\ g_{(f,c)} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}, \end{cases}$$

et un entier $i_{(f,c)} \in \{1, 2\}$ tels que l'on ait la propriété suivante :

- i. si $i_{(f,c)} = 1$: pour tout $n \in \mathbb{N}$ et tout $x \in \mathbb{N}$ tels que $n = f(x, g_f(n, x))$, on ait x dans l'intervalle $[a_f(n), b_f(n)]$ et la condition $c(x, g_f(n, x))$ vérifiée,
- ii. si $i_{(f,c)} = 2$: pour tout $n \in \mathbb{N}$ et tout $x \in \mathbb{N}$ tels que $n = f(g_f(n, x), x)$, on ait x dans l'intervalle $[a_f(n), b_f(n)]$ et la condition $c(g_f(n, x), x)$ vérifiée.

Les fonctions $a_{(f,c)}$ et $b_{(f,c)}$ déterminent l'intervalle des tailles valides pour la $i_{(f,c)}$ ^e composante. La fonction $g_{(f,c)}$ permet ensuite de calculer la taille de l'autre composante. Quand $b(n) - a(n) = O(n)$, ces limitations nous permettent de compter les structures d'une taille donnée en gardant la même complexité que dans le cas du produit simple, et donc de ne pas modifier la complexité globale de l'algorithme de dénombrement. Nous obtenons alors les fonctions de dénombrement suivantes, en notant implicitement h les fonctions $h_{(f,c)}$ et i l'entier $i_{(f,c)}$:

1.4.2.2.1 Univers non étiqueté.

$$\frac{A = Prod(B_1, B_2)}{(a)_n = \sum_{\substack{n=f(k,k') \\ k, k' \geq 0}} (b_1)_k \cdot (b_2)_{k'} = \sum_{k=a(n)}^{b(n)} (b_i)_k \cdot (b_{3-i})_{g(n,k)}}$$

1.4.2.2 Univers étiqueté.

$$\frac{A = \text{Prod}(B_1, B_2)}{(a)_n = \sum_{k=a(n)}^{b(n)} \binom{k + g(n, k)}{k} (b_i)_k \cdot (b_{3-i})_{g(n, k)}}$$

1.4.3 Génération aléatoire

1.4.3.1 Constructeur Δ_f

L'algorithme de génération aléatoire est simplement :

```

FUNCTION gener_A(n: INTEGER)
  k:=a_f(n);
  WHILE s < x DO
    k:=k+1;
    WHILE n≠f(k) DO k:=k+1; END_WHILE;
    s:=s+compte(B, k);
  END_WHILE
END_FUNCTION

```

1.4.3.2 Constructeur $\text{Prod}_{(f,c)}$

Nous connaissons les formules explicites pour le dénombrement des structures d'une taille donnée. Nous pouvons en déduire un algorithme de génération aléatoire, de même coût que dans le cas du produit simple, en utilisant une méthode *Boustrophédon*. Si $A \rightarrow \text{Prod}_{(f,c)}$, et en notant h les fonctions $h_{(f,c)}$ pour $h = a, b, f, g$, et i l'entier $i_{(f,c)}$, on obtient en univers non étiqueté :

Algorithme

```

FUNCTION gener_A(n: INTEGER)
  x:=uniforme(1,compte(A,n));
  k:=a(n)-1;
  s:=0;
  trouve:=FALSE;
  WHILE NOT trouve DO
    k:=k+1;
    p_k:=compte(B,k)*compte(C,g(n,k));
    s:=s+p_k;
    trouve:=s≥x;
    IF NOT trouve THEN
      k':=b(n)-k;
      p_k':=compte(B,k')*compte(C,g(n,k'));
      s:=s+p_k';
      trouve:=s≥x;
      IF trouve THEN
        k:=k';
      END_IF
    END_IF;
  END_WHILE;
  RETURN prod(gener_B(k),gener_C(n-k));
END_FUNCTION;

```

En univers étiqueté, on remplace simplement les lignes de dénombrement de p_k et p'_k par :

$$p_k := \binom{n}{k} \text{compte}(B, k) \text{compte}(C, g(n, k));$$

$$p_{k'} := \binom{n}{k'} \text{compte}(B, k') \text{compte}(C, g(n, k'));$$

1.4.4 Exemples d'utilisation

1.4.4.1 Arbres binaires de même hauteur

Nous pouvons traiter de nouveaux problèmes dans lesquels on considère la taille des structures comme étant fonction de la composante de taille maximale. Nous pouvons par exemple étudier les arbres binaires, en considérant que la taille d'un arbre correspond à sa hauteur.

Soit un arbre T binaire. On note \blacksquare les nœuds internes de T et \circ ses feuilles. La hauteur de l'arbre T est notée $h(T)$ et correspond à la fonction :

$$\left\{ \begin{array}{l} h(\circ) = 0 \\ h \left(\begin{array}{c} \blacksquare \\ \swarrow \quad \searrow \\ T.g \quad T.d \end{array} \right) = 1 + \max(h(T.g), h(T.d)) \end{array} \right.$$

Si l'on note \mathcal{T}_h l'ensemble des arbres binaires de hauteur h , et $\mathcal{T}_{<h}$ l'ensemble des arbres binaires de hauteur strictement inférieure à h , on obtient alors la relation :

$$\mathcal{T}_h = \begin{array}{c} \blacksquare \\ \swarrow \quad \searrow \\ \mathcal{T}_{h-1} \quad \mathcal{T}_{h-1} \end{array} \cup \begin{array}{c} \blacksquare \\ \swarrow \quad \searrow \\ \mathcal{T}_{h-1} \quad \mathcal{T}_{<h-1} \end{array} \cup \begin{array}{c} \blacksquare \\ \swarrow \quad \searrow \\ \mathcal{T}_{<h-1} \quad \mathcal{T}_{h-1} \end{array} \quad (1.10)$$

Pour traiter ce problème, nous pouvons utiliser le constructeur $Prod_{(p_1,=)}$, où p_i est la fonction de projection définie par $p_i : (x_1, x_2) \rightarrow x_i$. Nous ne considérons alors que les structures pour lesquelles la taille de la première composante est égale à celle de la seconde, en prenant comme taille celle de la première composante. Nous utilisons également les constructeurs $Prod_{(p_1,>)}$ et $Prod_{(p_2,<)}$, qui ne considèrent que les structures pour lesquelles les composantes sont de tailles différentes, en prenant comme taille celle de la plus grande composante.

Remarque. La spécification

$$\left\{ \begin{array}{l} A \rightarrow Union(A_1, A_2, A_3) \\ A_1 \rightarrow Prod_{(p_1,>)}(B, C) \\ A_2 \rightarrow Prod_{(p_1,=)}(B, C) \\ A_3 \rightarrow Prod_{(p_2,<)}(B, C) \end{array} \right.$$

revient à compter les structures définies par une spécification de la forme :

$$\{ A \rightarrow Prod_{(max,TRUE)}(B, C) \}$$

où TRUE la condition toujours vérifiée. Nous pouvons donc traiter les fonctions taille utilisant des fonctions max, si l'on peut trouver les fonctions de calcul des constructeurs $Prod_{(p_1,>)}$, $Prod_{(p_1,=)}$ et $Prod_{(p_2,<)}$.

Nous pouvons maintenant donner une spécification pour les arbres binaires dont la taille correspond à la hauteur, déduite directement de la relation 1.10, et utilisant les constructeurs produits introduits ci-dessus :

$$\left\{ \begin{array}{l} B \rightarrow Union(F, C) \\ C \rightarrow Union(B_1, B_2, B_3) \\ B_1 \rightarrow Prod(N, C_1) \\ B_2 \rightarrow Prod(N, C_2) \\ B_3 \rightarrow Prod(N, C_3) \\ C_1 \rightarrow Prod_{(p_1,=)}(B, B) \\ C_2 \rightarrow Prod_{(p_1,>)}(B, B) \\ C_3 \rightarrow Prod_{(p_2,<)}(B, B) \\ F \rightarrow f \\ f \rightsquigarrow Atom \\ N \rightarrow n \\ n \rightsquigarrow Atom \end{array} \right.$$

Il reste à exhiber les fonctions de calcul pour chacun des constructeurs introduits. Dans le cas du constructeur $Prod_{(p_1,=)}$, on obtient simplement :

$$\begin{cases} a_{(p_1,=)} : n \mapsto n, \\ b_{(p_1,=)} : n \mapsto n, \\ g_{(p_1,=)} : (n, k) \mapsto n, \\ i_{(p_1,=)} = 1. \end{cases}$$

Dans le cas du constructeur $Prod_{(p_2,<)}$, on trouve :

$$\begin{cases} a_{(p_2,<)} : n \mapsto 0, \\ b_{(p_2,<)} : n \mapsto n - 1, \\ g_{(p_2,<)} : (n, k) \mapsto n, \\ i_{(p_2,<)} = 1. \end{cases}$$

Le cas du constructeur $Prod_{(p_1,>)}$ est symétrique.

Pour créer un programme C permettant de compter le nombre d'arbres binaires ayant une hauteur donnée, on utilise la bibliothèque **Cconstruct** présentée en 1.2.4. La variable **defs** permet de spécifier les fonctions de dénombrement des constructeurs utilisés dans la spécification.

```
> read(Cconstruct);

> defs:={Prod=ProdExtd[0,size,size-k,1,1],
        ProdEqual=ProdExtd[size,size,size,1,1],
        ProdLower=ProdExtd[0,size-1,size,1,1],
        ProdGreater=ProdExtd[0,size-1,size,1,2]};

> spec:=subs(defs,{B = Union(F,C),
                  C = Union(B1, B2, B3),
                  B1 = Prod(N,C1),
                  B2 = Prod(N,C2),
                  B3 = Prod(N,C3),
                  C1 = ProdEqual(B,B),
                  C2 = ProdGreater(B,B),
                  C3 = ProdLower(B,B),
                  F = Atom,
                  N = Atom});

> create_count_file(B, spec, 'htree.c');
> quit;
```

Après compilation, on obtient bien la suite correspondant au nombres d'arbres binaires de hauteur n , qui correspond à la suite **M3087** du livre [49] :

```
% htree 0 7
[B(0)=1, B(1)=1, B(2)=3, B(3)=21, B(4)=651, B(5)=457653, B(6)=210065930571];
```

1.4.4.2 Arbre de taille particulière

Nous présentons dans cet exemple l'utilisation d'une fonction taille particulière. Soit un arbre T binaire. On note \blacksquare les nœuds internes de T et \circ ses feuilles. La taille de l'arbre T est notée $ch(T)$ et correspond à la fonction :

$$\begin{cases} ch(\circ) = 0, \\ ch \left(\begin{array}{c} \blacksquare \\ \swarrow \quad \searrow \\ T.g \quad T.d \end{array} \right) = 2 \times (1 + ch(T.g) + ch(T.d)). \end{cases}$$

Nous pouvons maintenant étudier les chemins de Dyck en fonction de plusieurs paramètres. Le dénombrement des chemins de Dyck dont la taille correspond à la longueur du chemin conduit aux mêmes techniques que celles employées dans les exemples précédents. Il y a une infinité de chemins d'une hauteur donnée, nous ne pouvons pas étudier ce paramètre seul. Nous choisissons naturellement dans cet exemple de compter le nombre de chemins de Dyck ayant une aire donnée. Nous obtenons les conditions suivantes portant sur les fonctions de longueur et d'aire :

$$\left\{ \begin{array}{l} \text{aire}(\triangle) = 1 \\ \text{aire}(\triangle \triangle) = \text{aire}(\triangle) + \text{longueur}(\triangle) + 1 \\ \text{aire}(\triangle_1 \triangle_2) = \text{aire}(\triangle_1) + \text{aire}(\triangle_2) \\ \text{longueur}(\triangle) = 2 \\ \text{longueur}(\triangle \triangle) = \text{longueur}(\triangle) + 2 \\ \text{longueur}(\triangle_1 \triangle_2) = \text{longueur}(\triangle_1) + \text{longueur}(\triangle_2) \end{array} \right.$$

Le problème de dénombrement dépend de deux paramètres, l'aire et la longueur des chemins. Nous devons donc, pour résoudre le problème de dénombrement, prendre en compte ces deux paramètres dans la notion de taille des chemins. Pour cela, nous choisissons de résoudre le problème uniquement pour des aires et longueurs inférieures à un entier N . La taille d'un chemin est alors donnée par l'équation :

$$\text{taille}(\triangle \triangle) = \text{longueur}(\triangle \triangle) \times N + \text{aire}(\triangle \triangle) \quad (1.11)$$

On en déduit les relations :

$$\left\{ \begin{array}{l} \text{taille}(\triangle) = 2 \times N + 1 \\ \text{taille}(\triangle_1 \triangle_2) = \text{taille}(\triangle_1) + \text{taille}(\triangle_2) \\ \text{taille}(\triangle \triangle) = \text{taille}(\triangle) + 2.N + \lfloor \frac{\text{taille}(\triangle \triangle)}{N} \rfloor + 1 \end{array} \right.$$

Une spécification des chemins de Dyck non vides dont la taille est donnée par l'équation 1.11 est :

$$\left\{ \begin{array}{l} A \rightarrow \text{Union}(Z, B_1, B_2, B_3) \\ B_1 \rightarrow \text{Prod}(Z, A) \\ B_2 \rightarrow \Delta_p(Z, A) \\ B_3 \rightarrow \text{Prod}(B_2, A) \\ Z \rightarrow \Delta_p(E) \\ E \rightarrow e \\ e \rightsquigarrow \text{Epsilon} \end{array} \right.$$

avec $a_p : x \mapsto 0$, $b_p : x \mapsto x$, et $p : n \in \mathbb{N} \mapsto n + \lfloor \frac{n}{N} \rfloor + 2.N + 1$. Le constructeur Δ_p correspond à l'opération $\triangle \triangle$ sur les chemins. Le constructeur Prod correspond à l'opération de concaténation des chemins. Le chemin de longueur deux correspond au non terminal Z de la spécification. Il faut maintenant vérifier que les opérations $\triangle \triangle \triangle$ et $\triangle \triangle \triangle$ construisent toujours des structures dont la taille est inférieure à $N^2 + N$, pour pouvoir extraire les données concernant l'aire et la longueur. On pourra résoudre le problème pour des tailles n telle que :

$$\left\{ \begin{array}{l} n \leq N^2 + N, \\ 2.n \leq N^2 + N, \\ 2.N + 1 + n + \lfloor \frac{n}{N} \rfloor \leq N^2 + N. \end{array} \right.$$

On note n_{max} la taille maximale obtenue. Prenons par exemple $N = 20$, il est alors possible de résoudre le problème pour des tailles inférieures à $n_{max} = 210$. En **Maple**, nous écrivons, en renommant **concat** l'opération $\triangle \triangle \triangle$, et **base** l'opération $\triangle \triangle$:

```
N:=20;
ap:=x -> '0';
```

```

bp:=x -> convert(x,string);
gp:=(n,x) -> cat('2*',convert(N,string),'+',convert(x,string),
               '(int) (' ,convert(x,string),'/',convert(N,string),'+1');

spec:={A = Union(B,S),
      S = ProdExtd[2*N+1,size-(2*N+1),size-k,concat](B,A),
      B = Union(Z,C),
      Z = DeltaExtd[ap,bp,gp,base](E),
      E = Epsilon,
      C = DeltaExtd[ap,bp,gp,base](A)};

```

Dans la spécification, nous avons explicité la valuation de A ($2N + 1$) en définissant le produit, ce qui permet d'accélérer les calculs. L'expression des fonctions de dénombrement ci-dessus est faite suivant la syntaxe du langage \mathbb{C} , d'où leur aspect peu accueillant.

Après compilation, nous comptons le nombre de chemins de Dyck d'aire 8 et de longueur 8, puis générons 10 structures de façon aléatoire. La taille des structures cherchées est $8 \times 20 + 8 = 168$:

```

% dyck 168 10
A(168)=3;
[prod(base(E),base(prod(base(E),base(E)))) ,
 prod(base(base(E)),base(base(E))),
 prod(base(E),base(prod(base(E),base(E)))) ,
 prod(base(E),base(prod(base(E),base(E)))) ,
 prod(base(E),base(prod(base(E),base(E)))) ,
 prod(base(base(E)),base(base(E))),
 prod(base(base(E)),base(base(E))),
 prod(base(base(E)),base(base(E))),
 prod(base(base(E)),base(base(E))),
 prod(base(prod(base(E),base(E))),base(E))];

```

$\mathbf{delta}(E)$ représente le chemin \triangleleft . Ces structures sont représentées sur la figure 1.6.

Nous pouvons aller plus loin, en étudiant par exemple les ensembles de chemins de Dyck ayant une aire donnée, et tels que la taille est inférieure à n_{max} (pour $N = 20$, l'aire et la longueur doivent être inférieurs à 10). Nous ajoutons pour cela les règles de production suivantes à la spécification :

$$\begin{cases} K \rightarrow Set(F) \\ F \rightarrow \Delta_q(A) \end{cases}$$

avec $a_q : x \mapsto 0$, $b_q : x \mapsto n_{max}$, et $q : x \mapsto x \bmod N$. On obtient les 6 ensembles de chemins de Dyck d'aire 4, représentées sur la figure 1.7 :

```

% Sdyck 4 6
K(4)=6;
{Set(delta(prod(base(E),base(E))),delta(base(E)),delta(base(E))),
 Set(delta(prod(base(E),base(E))),delta(prod(base(E),base(E)))) ,
 Set(delta(base(E)),delta(base(E)),delta(base(E))),
 Set(delta(prod(base(E),prod(base(E),prod(base(E),base(E))))),
 Set(delta(prod(base(E),prod(base(E),base(E)))) ,delta(base(E))),
 Set(delta(base(base(E))))}

```

1.5 Conclusion

Nous avons présenté dans ce chapitre la théorie des structures décomposables. Nous avons proposé une implantation de la méthode de génération aléatoire de structures décomposables décrite dans [26], qui permet, en créant des programmes écrit en langage \mathbb{C} , de générer plus rapidement des structures de plus grande taille que précédemment. Nous avons également abordé l'emploi de nouveaux constructeurs qui permettent de modifier la notion de taille des structures, et nous avons présenté leur utilisation sur

plusieurs exemples. Cette extension offre cependant potentiellement moins de puissance que les constructeurs initiaux. Nous ne pouvons en effet pas associer de séries génératrices aux spécifications, ce qui empêche par exemple de connaître le comportement asymptotique du nombre de structures en fonction de leur taille.

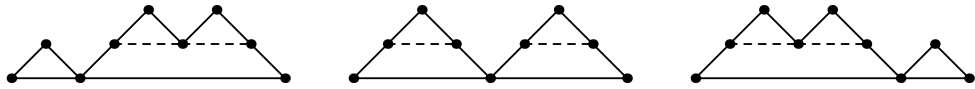


Fig. 1.6: Les trois chemins de Dyck d'aire 8 et de longueur 8.

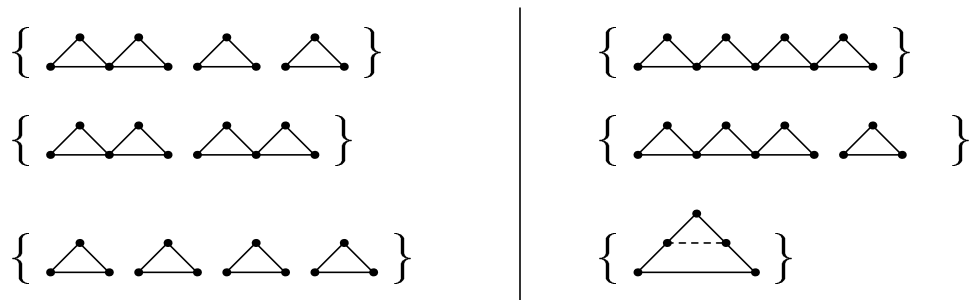


Fig. 1.7: Les 6 ensembles de chemins de Dyck d'aire 4.

Chapitre 2

Méthode de rang inverse sur des ensembles de structures décomposables

Nous présentons dans ce chapitre des techniques de *rang inverse* pour des ensembles de structures décomposables décrits par des constructeurs union, produit, ensemble et cycle. Une des applications de ces techniques est la construction une à une des structures appartenant à un ensemble de structures, étant donnée une spécification de ce dernier. Ce problème de rang inverse est relativement simple dans le cas de spécifications n'utilisant que des constructeurs union et produit, et conduit à des techniques similaires à celles présentées dans le cas de la génération aléatoire au chapitre 1. Par contre, l'utilisation des constructeurs ensemble et cycle obligent à concevoir entièrement de nouveaux algorithmes.

Le principal résultat de ce chapitre est la description d'un algorithme incrémental systématique pour toutes les spécifications de Ω et Ω . Nous introduisons pour cela la notion de *forme* des structures, qui permet une approche du problème identique quels que soient les constructeurs utilisés. Ceci permet de décomposer le problème de rang inverse général, portant sur les ensembles de structures, en problèmes de rang inverse plus simples portant sur des suites d'entiers aux propriétés particulières, pour lesquels il existe déjà des algorithmes satisfaisants. Cette démarche permet de plus de décrire de façon plus fine les ensembles de structures que l'on manipule. L'algorithme que nous proposons est effectivement implanté en **Maple** dans le cas des spécifications de Ω et nous donnons quelques exemples d'application.

2.1 Le problème du rang inverse

2.1.1 Définition

Pour un ensemble de structures combinatoires contenant $(a)_n$ structures de taille n , il est possible de définir une bijection entre les structures de l'ensemble et les entiers de l'intervalle $[1, (a)_n]$. Le problème du *rang inverse* consiste à construire la structure a associée à un entier $k \in [1, (a)_n]$, selon la bijection choisie. k est appelé le *rang* de la structure a .

Définition 2.1.1 (Fonction de rang, fonction de rang inverse)

Soit \mathcal{E} un ensemble fini de structures de cardinalité n , ordonné par une relation d'ordre total \prec . Une *fonction de rang* de l'ensemble \mathcal{E} est une bijection β de \mathcal{E} dans l'intervalle $[1, n]$ qui préserve l'ordre \prec :

$$\forall a \forall b \in \mathcal{E}, a \prec b \implies \beta(a) < \beta(b).$$

La fonction inverse, β^{-1} , est appelée *fonction de rang inverse*. .

Remarque. Il n'est pas nécessaire de connaître explicitement l'ordre.

2.1.2 Arbres binaires planaires

Considérons le problème du rang inverse dans le cas des arbres binaires, en univers non étiqueté. Ce problème a fait l'objet de nombreux travaux [42, 43, 56]. Soit T un arbre binaire. Nous considérons que la

taille d'un arbre binaire, notée $|T|$, est donnée par son nombre de nœuds internes (on ne compte pas les feuilles). Nous notons $T.g$ le sous-arbre gauche de T , et $T.d$ le sous-arbre droit. Nous considérons l'ordre suivant sur les arbres :

Définition 2.1.2 (ordre $A \prec_A$)

On a $T \prec_A T'$ suivant l'ordre A si :

$$\left| \begin{array}{l} |T| < |T'| \\ \text{ou} \\ |T| = |T'| \wedge \end{array} \right| \text{ou} \begin{array}{l} T.g \prec_A T'.g \\ T.g = T'.g \wedge T.d \prec_A T'.d \end{array}$$

On note $(b)_n$ le nombre d'arbres binaires de taille n . En univers non étiqueté nous avons, d'après la formule de Catalan, $(b)_n = \frac{1}{n+1} \binom{2n}{n}$. Le nombre d'arbres binaires peut s'écrire également :

$$\left\{ \begin{array}{l} (b)_1 = 1, \\ (b)_n = \sum_{i=0}^{n-1} (b)_i \times (b)_{n-1-i}. \end{array} \right.$$

On note $(b)_{n,j}$ le terme $\sum_{i=0}^j (b)_i \times (b)_{n-1-i}$. Par convention on prend $(b)_{n,-1} = 0$. Un algorithme de rang inverse des arbres binaires, suivant l'ordre A , consiste à chercher tout d'abord la taille que doit avoir le sous-arbre gauche de l'arbre à construire, sachant que, pour deux arbres de même taille, l'arbre dont la taille du sous-arbre gauche est la plus petite aura le plus petit rang. Ceci détermine le plus petit rang k' des arbres dont le sous-arbre gauche est de taille j . On cherche ensuite, en fonction de k , le rang $k.g$ parmi les arbres de taille j , qui correspond au rang du sous-arbre gauche à construire, et le rang $k.d$ parmi les arbres de taille $n-j$ pour le sous-arbre droit, qui respectent l'ordre A .

Algorithme

- *Données* : un entier n et un entier $k \in [1, (b)_n]$
- *Résultat* : l'arbre binaire de taille n et de rang k , suivant l'ordre A . Les feuilles de l'arbre sont notées \circ , les nœuds internes \blacksquare .

```

FUNCTION rang_inverse(k: INTEGER, n: INTEGER)
  IF n = 0 THEN
    RETURN  $\circ$ ;
  ELSE
    LET j SUCH THAT  $(b)_{n,j-1} < k \leq (b)_{n,j}$ ,  $0 \leq j \leq n-1$ ;
    k' :=  $k - (b)_{n,j-1} - 1$ ;
    k.g :=  $1 + \lfloor k' / (b)_{n-j} \rfloor$ ;
    k.d :=  $1 + k' \bmod (b)_{n-j}$ ;
    T.g := rang_inverse(k.g, j);
    T.d := rang_inverse(k.d, n-j);
    RETURN  $\blacksquare$ 
       $\swarrow$   $\searrow$ 
      T.g T.d ;
  END_IF;
END_FUNCTION

```

Remarque. Il existe d'autres algorithmes de rang inverse pour les arbres binaires, qui respectent d'autres ordres. On peut citer par exemple l'ordre B , qui est défini comme suit. On a $T \prec_B T'$ suivant le B-ordre si :

$$\left| \begin{array}{l} |T| = 0 \wedge |T'| \neq 0 \\ \text{ou} \\ |T| \neq 0 \vee |T'| = 0 \wedge \end{array} \right| \text{ou} \begin{array}{l} T.g \prec_B T'.g \\ T.g = T'.g \wedge T.d \prec_B T'.d \end{array}$$

2.1.3 Structures décomposables

Pour résoudre le problème de rang inverse dans le cas des structures décomposables, on peut être tenté, dans un premier temps, d'utiliser les mêmes techniques que celles présentées au chapitre précédent. On associe tout d'abord à la spécification sa spécification standard. Ensuite, au lieu de choisir un nombre aléatoire pour déterminer la taille des composantes de la structure, comme on le fait le cas de la génération aléatoire uniforme, on détermine le rang des composantes en fonction du rang de la structure à construire. Cette méthode est d'ailleurs implantée dans `Ccombstruct`, dans le cas de spécifications n'utilisant que des constructeurs *Prod* et *Union*. Malheureusement, dans le cas d'une spécification standard utilisant le constructeur Θ^{-1} , on ne peut pas calculer le rang de la composante, étant donné que l'on ne sait pas quel terminal il convient de marquer. Résoudre ce problème revient alors à construire au préalable l'ensemble de toutes les structures de taille n de la composante, et résoudre le problème de rang inverse sur cet ensemble. Les constructeurs *Set* et *Cycle* conduisant à l'utilisation du constructeur Θ^{-1} , ceci nous conduit à employer d'autres techniques.

2.2 Formes

Nous introduisons maintenant la notion de *forme*. Dans l'algorithme de rang inverse sur les arbres binaires présenté au début de ce chapitre, la première étape consiste à déterminer la taille du sous-arbre gauche en fonction du rang de l'arbre. L'introduction des *formes* est destinée à rendre systématique cette première étape de l'algorithme, pour chacun des constructeurs d'une spécification.

2.2.1 Définitions

Définition 2.2.1 (Forme d'une structure)

La *forme* d'une structure est un couple (Φ, L) , où Φ est un constructeur et L une séquence de couples $(N_1, n_1), \dots, (N_s, n_s)$, où les N_1, \dots, N_s sont des non terminaux, et les $n_i, i = 1, \dots, s$ des entiers. N_j est appelé l'ensemble de structures de la j^e composante de la forme, et n_j la taille de la j^e composante. .

Définition 2.2.2 (Appartenance d'une structure à une forme)

Une structure o appartient à une forme $\Phi, (N_1, n_1), \dots, (N_s, n_s)$, si et seulement s'il existe une écriture de la structure o sous la forme $\phi(o'_1, \dots, o'_s)$, telle que :

$$\forall j \in [1, s], |o'_j| = n_j \text{ et } o'_j \in N_j.$$

Exemple. Si $a \in A$, avec $|a| = 1$, et $b \in B$, avec $|b| = 2$, la structure $o = \text{cycle}(a, a, b)$ appartient à la forme $(\text{Cycle}, (B, 2), (A, 1), (A, 1))$. On peut en effet écrire la structure o sous la forme $o = \text{cycle}(b, a, a)$. .

Notation. Dans le cas du constructeur *Set*, nous utiliserons deux notations différentes selon les cas. Nous notons :

$$S = (\text{Set}, (A, i_1, m_1), \dots, (A, i_s, m_s)),$$

qui correspond à la forme $S = (\text{Set}, \underbrace{(A, i_1), \dots, (A, i_1)}_{m_1}, \dots, \underbrace{(A, i_s), \dots, (A, i_s)}_{m_s})$. i_j est appelé le j^e opérande de la forme de multiplicité m_j . .

2.2.2 Énumération de formes

Nous employons le terme *comptage* pour désigner l'action qui consiste à compter le nombre de structures ou des formes, et le terme *énumération* pour l'action qui consiste à exhiber une à une des structures ou des formes.

Définition 2.2.3 (Énumération des formes de taille n)

Soit A un non terminal d'une spécification vérifiant $A \rightarrow \Phi(B_1, \dots, B_s)$. Une *énumération de formes* est une séquence de I formes $\mathcal{F} = (f(i))_{i=1}^I$. Une énumération de forme est *valide*, si pour toute structure a

de taille n appartenant à A , il existe un unique $i \in [1, I]$ tel que a appartient à la forme $f(i)$. i est appelé le rang de la forme dans l'énumération \mathcal{F} . .

Les fonctions permettant de construire une énumération valide $\mathcal{F} = (f(i))_{i=1}^I$ sont :

- la fonction `premier_forme`, qui retourne la première forme $f(1)$ de l'énumération \mathcal{F} . Si aucune forme n'existe, la fonction retourne `nil` :

```
FUNCTION premier_forme(A: nonterminal, n: INTEGER) : forme
```

- la fonction `suitant_forme` qui retourne, étant donnée une forme $f(i)$ de \mathcal{F} , la forme de rang suivant $f(i+1)$. S'il n'existe pas de forme de rang suivant, `nil` est retourné :

```
FUNCTION suivant_forme(f: forme) : forme
```

Deux formes f_1 et f_2 d'une énumération de formes sont ordonnées suivant leur rang dans l'énumération. Nous noterons $f_1 < f_2$ si le rang de la forme f_1 est inférieur à celui de f_2 dans l'énumération de formes.

Par la suite, nous supposons qu'il est possible d'associer à chaque forme des *informations* supplémentaires, spécifiques au type de forme. Par exemple, nous pouvons ajouter l'information qui correspond au rang de la forme dans l'énumération. Si f est la i^e forme de l'énumération, nous noterons $f[\text{rang_forme}=i]$. Un appel à la fonction `suitant_forme(f)` retournera une forme $f'[\text{rang_forme}=i+1]$. Nous noterons alors `rang_forme(f)`, la fonction de coût unitaire donnant la valeur du champ `rang_forme` d'une forme f . Pour alléger les notations, les informations des formes ne seront explicitées que quand cela est nécessaire.

Nous décrivons maintenant les fonctions d'énumération de formes pour chaque constructeur. Dans la suite, un appel à la fonction `premier_forme(A)`, où A est un non terminal vérifiant $A \rightarrow \Phi(B_1, \dots, B_s)$, correspond à un appel à la fonction `premier_forme_Φ(A)`. De même, nous faisons correspondre un appel à la fonction `premier_forme(f)`, où f est une forme vérifiant $f = (\Phi, \dots)$, à celui de la fonction `suitant_forme_Φ(f)`.

2.2.2.1 Énumération des formes *Union*

Proposition 2.2.4 (Énumération de formes *Union*)

Les fonctions suivantes permettent de construire une énumération de formes valide pour les règles de production de la forme $A \rightarrow \text{Union}(B_1, \dots, B_s)$:

```
FUNCTION premier_forme_union(A: nonterminal, n: INTEGER) : forme
  RETURN (Union, (B1, n)) [rang_forme=1, operandes=(B1, ..., Bs)];
END_FUNCTION;
```

```
FUNCTION suivant_forme_union(f: forme) : forme
  (Union, (B', n)) := f;
  (B1, ..., Bs) := operandes(f);
  i := rang_forme(f);
  IF i ≥ s THEN RETURN nil; END_IF;
  RETURN (Union, (Bi+1, n)) [rang_forme=i+1, operandes=(B1, ..., Bs)];
END_FUNCTION;
```

Proposition 2.2.5

Le coût en temps d'un appel à la fonction `suitant_forme_union` pour une forme d'arité s est unitaire. .

Démonstration. Nous supposons ici que la détermination de s est unitaire, et l'algorithme est sans boucle. □

2.2.2.2 Énumération des formes *Prod*

Le principe de l'algorithme d'énumération des formes *Prod* de taille n et d'arité s consiste à construire les compositions de l'entier n en s parties. Une composition de n en s parties est une représentation de l'entier n sous forme de somme de s entiers positifs. Deux compositions sont différentes si l'ordre des sommants diffère :

$$\begin{aligned} n &= r_1 + r_2 + \dots + r_s, \\ r_i &\geq 0, i = 1, \dots, s. \end{aligned}$$

Il y a $\binom{n+s-1}{n}$ compositions de n en s parties. À partir d'une composition $n = r_1 + \dots + r_s$, pour une règle de production de la forme $A \rightarrow Prod(B_1, \dots, B_s)$, on construit la forme $Prod, (B_1, r_1), \dots, (B_s, r_s)$. L'entier r_i dans la composition $n = r_1 + r_2 + \dots + r_s$ indique la taille de la i^e composante des structures de la forme. De même, deux structures construites avec le constructeur *Prod* d'arité s diffèrent s'il existe un entier $i \in [1, s]$, tel que les i^e composantes des deux structures n'ont pas la même taille. Une structure est donc dans une seule des formes produites. De plus, comme nous construisons toutes les compositions de n en s parties, chaque structure de taille n construite avec un constructeur *Prod* d'arité s est dans une des formes, et l'énumération est valide.

Nous ramenons le problème de l'énumération des formes dans le cas du constructeur *Prod* d'arité s à celui consistant à trouver toutes les compositions d'un entier n en s parties. Nous appliquons l'algorithme **NEXCOM** proposé par A. Nijenhuis and H. S. Wilf ([41], chapitre 5).

Proposition 2.2.6 (Énumération de *Prod*)

Les fonctions suivantes permettent de construire une énumération de forme valide pour les règles de production de la forme $A \rightarrow Prod(B_1, \dots, B_s)$.

```

FUNCTION premier_forme_prod(A: nonterminal, n: INTEGER) : forme
  RETURN (Prod, (B1, n), (B2, 0), ..., (Bs, 0));
END_FUNCTION;

FUNCTION suivant_forme_prod(f: forme) : forme
  (Prod, (B1, n1), ..., (Bs, ns)) := f;
  IF ns = n THEN RETURN nil; END_IF;
  h := 1;
  WHILE nh = 0 DO
    h := h + 1;
  END_WHILE;
  t := nh; nh := 0; n1 := t - 1; nh+1 := nh+1 + 1;
  RETURN (Prod, (B1, n1), ..., (Bs, ns));
END_FUNCTION;

```

L'algorithme proposé consiste à construire les compositions de n en s parties, suivant l'ordre lexicographique inverse (de la droite vers la gauche, sur l'alphabet des entiers de 1 à n). Soient deux formes f et g de la forme :

$$\begin{aligned} f &= (Prod, (B_1, q_1), \dots, (B_s, q_s)) \\ g &= (Prod, (B_1, r_1), \dots, (B_s, r_s)) \end{aligned}$$

On a $f \prec g$ si et seulement si :

$$\begin{aligned} q_{t-1} &< r_{t-1} \\ r_i &= q_i, i = t, \dots, s \end{aligned}$$

Exemple. Pour $n = 2$ et $s=4$, on a :

$$\begin{aligned} (2, 0, 0, 0) &\prec (1, 1, 0, 0) \prec (0, 2, 0, 0) \prec (1, 0, 1, 0) \prec (0, 1, 1, 0) \prec \\ (0, 0, 2, 0) &\prec (1, 0, 0, 1) \prec (0, 1, 0, 1) \prec (0, 0, 1, 1) \prec (0, 0, 0, 2) \end{aligned}$$

Proposition 2.2.7

Le coût total des appels pour énumérer les $N(n, s) = \binom{n+s-1}{n}$ formes *Prod* de taille n et d'arité s est en $O(N(n, s))$. Le coût moyen asymptotique d'un appel à la fonction `suitant_forme_prod` est constant. .

Démonstration. On note n_k le nombre de compositions de n en s parties commençant par au moins k zéros :

$$n_k = \text{Card}\{\underbrace{(0, \dots, 0)}_k, r_1, \dots, r_{s-k}\} = \binom{n + (s - k) - 1}{n}.$$

Le nombre de compositions de n en s parties commençant par exactement k zéros est donné par $n_k - n_{k+1}$. Le test de la boucle `WHILE` est effectuée $k + 1$ fois pour des compositions se terminant par exactement k zéros. Le coût total de l'algorithme est donc :

$$\sum_{k=0}^{s-1} (k+1) \cdot (n_k - n_{k+1}) = \sum_{k=0}^{s-1} (k+1)n_k - \sum_{k=1}^s kn_k = \sum_{k=0}^{s-1} n_k = \binom{n+s-1}{n} \frac{n+s}{n+1} = N(n, s) \frac{n+s}{n+1}.$$

□

2.2.2.3 Énumération des formes *Set*

Le principe de l'algorithme proposé pour l'énumération des formes *Set* de taille n consiste à construire les partitions de l'entier n . Une partition de n est une représentation de l'entier n sous forme de somme de s entiers positifs :

$$\begin{aligned} s &\in \mathbb{N}, \\ n &= r_1 + r_2 + \dots + r_s, \\ r_1 &\geq r_2 \geq \dots \geq r_s \geq 1, i = 1, \dots, s. \end{aligned}$$

Le problème de l'énumération des formes *Set* est ramené à celui de l'énumération des partitions de l'entier n . Les fonctions proposées correspondent à l'algorithme `NEXPART` proposé par A. Nijenhuis et H. S. Wilf ([41], chapitre 9).

Proposition 2.2.8 (Énumération de *Set*)

Les fonctions suivantes permettent de construire une énumération de forme valide pour les règles de production de la forme $A \rightarrow \text{Set}(B)$:

```
FUNCTION premier_forme_set(A: nonterminal, n: INTEGER) : forme
  RETURN (Set, (B, n, 1)) [taille_forme=n];
END_FUNCTION;
```

```
FUNCTION suivant_forme_set(f: forme) : forme
  (Set, (B, n1, m1), ..., (Bs, ns, ms)) := f;
  n := taille_forme(f);
  IF ms = n THEN RETURN nil; END_IF;
  d := s;
  IF nd = 1 THEN
    sigma := md + 1; d := d - 1;
  ELSE
    sigma := 1;
  END_IF;
  g := nd - 1;
  IF md ≠ 1 THEN
    md := md - 1; d := d + 1;
  END_IF;
  nd := g; md := 1 + ⌊ $\frac{\sigma}{g}$ ⌋; s := sigma mod g;
  IF s ≠ 0 THEN
```

```

    d:=d+1; n_d:=s; m_d:=1;
  END_IF;
  RETURN (Set, (B, n_1, m_1), ..., (B, n_d, m_d)) [taille_forme=n];;
END_FUNCTION;

```

Soient deux formes f et g de la forme :

$$\begin{aligned}
 f &= (Set, (B, q_1), \dots, (B, q_s)), q_1 \geq \dots \geq q_t \\
 g &= (Set, (B, r_1), \dots, (B, r_t)), r_1 \geq \dots \geq r_t
 \end{aligned}$$

On a $f \prec g$ si et seulement si, pour un entier $k \geq 0$:

$$\begin{aligned}
 q_{k+1} &> r_{k+1} \\
 q_i &= r_i, i = 1, \dots, k
 \end{aligned}$$

Exemple. Pour $n = 6$, on a :

$$\begin{aligned}
 (6) &\prec (5, 1) \prec (4, 2) \prec (4, 1, 1) \prec (3, 3) \prec (3, 2, 1) \prec (3, 1, 1, 1) \\
 &\prec (2, 2, 2) \prec (2, 2, 1, 1) \prec (2, 1, 1, 1, 1) \prec (1, 1, 1, 1, 1, 1)
 \end{aligned}$$

Proposition 2.2.9

Le coût d'un appel à la fonction `suitant_forme_set` pour une forme quelconque est constant.

Démonstration. L'algorithme est sans boucle. □

Remarque. Nous avons supposé que les composantes des structures construites à l'aide du constructeur `Set` ne pouvaient pas être de taille nulle. En effet, une composante de taille nulle permettrait de construire un nombre infini de structures dans l'ensemble. Par contre, quand le nombre de composantes est borné, il faut prendre en compte les composantes de taille nulle comme nous le verrons au point 2.4.1.

2.2.2.4 Énumération des formes *Cycle*

Le principe de l'algorithme proposé pour l'énumération des formes *Cycle* s'appuie sur le principe de cycle représentant. L'idée consiste à choisir une unique représentation pour les cycles. Nous choisissons pour cela l'ordre lexicographique inverse pour un alphabet formé de couples d'entier :

Définition 2.2.10 (*Cycle* représentant)

Soit une suite de s couples d'entiers $c_1 = (n_1, k_1), \dots, c_s = (n_s, k_s)$. Nous dirons que la suite est un *cycle* représentant si la propriété suivante est vérifiée pour tout $t \in [1, s - 1]$:

$$\exists u_t \text{ tel que } \begin{cases} c_i = c_{(1+(i+t) \bmod s)}, i = 1, \dots, u_t \\ c_{(u_t+1)} > c_{(1+(u_t+1+t) \bmod s)} \end{cases} \quad (2.1)$$

où $(a, b) = (c, d)$ si et seulement si $a = c \wedge b = d$, et $(a, b) > (c, d)$ si et seulement si $a > c \vee (a = c \wedge b > d)$.

Une structure $o = \text{cycle}(o_1, \dots, o_s)$ est le représentant de toutes les structures cycles égales à o (par décalage cyclique des composantes), si la propriété (2.1) est vérifiée pour les couples formés par les tailles et les rangs des composantes de la structure.

Ceci revient à choisir une écriture particulière unique pour toutes les structures *cycle* identiques. L'écriture choisie pour un cycle est celle suivant l'ordre anti-lexicographique sur la taille des composantes.

Pour l'énumération des formes *Cycle*, nous ne considérons que les formes contenant des cycles représentants, ce qui permet d'assurer que les structures construites à l'aide d'un constructeur *Cycle* n'appartiennent pas à plusieurs formes. La condition pour un cycle d'être représentant ou non dépend de la taille de ses composantes. On peut donc en déduire qu'une forme *Cycle* contient potentiellement des cycles représentants, ou uniquement des cycles non représentants. Nous supposons que nous disposons d'une fonction permettant de tester si la propriété 2.1 est vérifiée :

```

FUNCTION est_representant_cycle((c_1, ..., c_s)) : BOOLEAN

```

Nous ne présentons ici qu'une méthode naïve et très coûteuse pour l'énumération des formes *Cycle*. Il en existe certainement de meilleures. Elle consiste, pour une forme de taille n , à énumérer toutes les compositions bornées de l'entier n en s parties (comme dans le cas des formes *Prod*), et à ne retenir que les compositions qui correspondent à des cycles représentants, et ce pour tout s de 1 à n . L'algorithme se déduit directement des formes produit, et nous ne le détaillons pas.

Exemple. Pour $n = 6$ on a :

$$(6) \prec (4, 1) \prec (5, 1) \prec (4, 2) \prec (3, 3) \prec (4, 1, 1) \prec (3, 2, 1) \prec (3, 1, 2) \prec (2, 2, 2) \\ \prec (3, 1, 1, 1) \prec (2, 2, 1, 1) \prec (2, 1, 2, 1) \prec (2, 1, 1, 1, 1) \prec (1, 1, 1, 1, 1, 1)$$

2.2.2.5 Ordre sur les structures

Remarque. L'ordre sur les structures construites dépend de l'ordre choisi lors de l'énumération des formes. Il dépend non seulement des structures, mais aussi de la spécification utilisée pour les décrire. Deux spécifications équivalentes, au sens où chacune construit le même ensemble de structures, ne déterminent pas forcément le même ordre sur les structures. Cela tient au fait de l'utilisation du constructeur *Union* pour lequel l'ordre des opérandes est indifférent. Pour obtenir un ordre indépendant de la spécification, on peut définir un ordre arbitraire sur les non terminaux dans le cas du constructeur *Union*.

2.2.3 Comptage des structures dans une forme

Nous nous intéressons maintenant au problème qui consiste à savoir combien il existe de structures dans chaque forme. La fonction calculant le nombre de structures dans une forme f , selon les formules ci-dessus, est notée :

`FUNCTION` `compte_forme(f: forme) : INTEGER`

Pour les formes autres que les formes *Cycle*, nous avons des formules explicites de calcul du nombre de structures dans une forme, différentes selon l'univers dans lequel on se place.

2.2.3.1 Univers non étiqueté

Proposition 2.2.11 (Nombre de structures dans une forme)

Le nombre de structures qui appartiennent à une forme en univers non étiqueté est donné par les relations suivantes :

$$\frac{f = (Union, (A, n))}{(f)_n = (a)_n}$$

$$\frac{f = (Prod, (A_1, n_1), \dots, (A_s, n_s))}{(f)_n = \prod_{k=1}^s (a_k)_{n_k}}$$

$$\frac{f = (Set, (A, n_1, m_1), \dots, (A, n_s, m_s))}{(f)_n = \prod_{k=1}^s \binom{m_k + (a)_{n_k} - 1}{m_k}}$$

En guise d'exemple, nous pouvons considérer la spécification suivante :

$$\left\{ \begin{array}{l} A \rightarrow Prod(\mathbf{a}, B) \\ B \rightarrow Set(A) \\ \mathbf{a} \rightsquigarrow Atom \end{array} \right.$$

On a alors :

$$(a)_1 = 1, (a)_2 = 1, (a)_3 = 2, (a)_4 = 4.$$

Les formes des structures de taille 4 associées au non terminal B et les structures qu'elles contiennent sont résumées dans le tableau suivant :

Forme	Structures dans la forme
$(Set, (A, 4, 1))$	$\left. \begin{array}{l} Set(Prod(a, Set(Prod(a, Set(Prod(a, Set(Prod(a, Epsilon))))))) \\ Set(Prod(a, Set(Prod(a, Set(Prod(a, Epsilon), Prod(a, Epsilon)))))) \\ Set(Prod(a, Set(Prod(a, Set(Prod(a, Epsilon))), Prod(a, Epsilon)))) \\ Set(Prod(a, Set(Prod(a, Epsilon), Prod(a, Epsilon), Prod(a, Epsilon)))) \end{array} \right\} \binom{1+(a)_4-1}{1}$
$(Set, (A, 3, 1), (A, 1, 1))$	$\left. \begin{array}{l} Set(Prod(a, Set(Prod(a, Set(Prod(a, Epsilon))))), Prod(a, Epsilon)) \\ Set(Prod(a, Set(Prod(a, Epsilon), Prod(a, Epsilon))), Prod(a, Epsilon)) \end{array} \right\} \binom{1+(a)_3-1}{1} \binom{1+(a)_1-1}{1}$
$(Set, (A, 2, 2))$	$Set(Prod(a, Set(Prod(a, Epsilon))), Prod(a, Set(Prod(a, Epsilon)))) \} \binom{2+(a)_2-1}{2}$
$(Set, (A, 2, 1)(A, 1, 2))$	$Set(Prod(a, Set(Prod(a, Epsilon))), Prod(a, Epsilon), Prod(a, Epsilon)) \} \binom{1+(a)_2-1}{1} \binom{2+(a)_1-1}{2}$
$(Set, (A, 1, 4))$	$Set(Prod(a, Epsilon), Prod(a, Epsilon), Prod(a, Epsilon), Prod(a, Epsilon)) \} \binom{4+(a)_4-1}{4}$

2.2.3.2 Univers étiqueté

Définition 2.2.12 (Coefficient multinomial)

Le *coefficient multinomial* correspond au nombre de réétiquetages compatibles avec l'ordre initial entre k objets de tailles respectives n_1, \dots, n_k . Il est noté :

$$\binom{n_1 + \dots + n_k}{n_1, \dots, n_k} = \frac{(n_1 + \dots + n_k)!}{n_1! \dots n_k!}$$

Considérons l'exemple suivant d'une structure de taille deux étiquetée par $(1, 2)$, et d'une structure de taille 1 étiquetée par (1) . Le nombre de réétiquetages compatibles entre les deux structures est trois :

$$\begin{array}{l} (1, 2)|(3) \\ (2, 3)|(1) \\ (1, 3)|(2) \end{array}$$

Proposition 2.2.13 (Nombre de structures dans une forme)

Le nombre de structures qui appartient à une forme en univers étiqueté est donné par les relations suivantes :

$$\begin{aligned} f &= (Union, (A, n)) \\ (f)_n &= (a)_n \\ f &= (Prod, (A_1, n_1), \dots, (A_s, n_s)) \\ (f)_n &= \binom{n_1 + \dots + n_s}{n_1, \dots, n_s} \prod_{k=1}^s (a_k)_{n_k} \\ f &= (Set, (A, n_1, m_1), \dots, (A, n_s, m_s)) \\ (f)_n &= \binom{n_1 m_1 + \dots + n_s m_s}{n_1 m_1, \dots, n_s m_s} \prod_{k=1}^s ((a)_{n_k})^{m_k} \end{aligned}$$

2.2.3.3 Formes Cycle

Dans le cas des formes cycles, nous utilisons là encore un algorithme coûteux mais simple, utilisable en univers étiqueté ou non. Son principe revient en fait à résoudre le problème de rang inverse dans le cas du produit, en utilisant la fonction `calcul_parties` qui sera présentée et expliquée plus loin. Pour cela, on calcule le rang k_i de chaque composante, comme si la forme était une forme produit. On ne retient ensuite que les structures qui correspondent à des cycles représentants, c'est-à-dire les structures telles que la suite $((k_1, n_1), \dots, (k_s, n_s))$ vérifie la propriété (2.1).

```

FUNCTION compte_forme_cycle(f: forme): INTEGER
  Cycle, (B, n1), ..., (B, ns):=f;
  n:=taille_forme(f);
  if n=0 then
    RETURN 0;
  else
    p:= $\prod_{k=1}^s (b)_{n_k}$ ;
    c:=0;
    FOR k FROM 1 TO p DO
      (k1, ..., ks):=calcul_parties(k, (b1)n1, ..., (bs)ns);
      if est_representant_cycle(((k1, n1), ..., (ks, ns))) then
        c:=c+1;
      END_IF
    END_FOR;
    RETURN c;
  END_IF;
end;

```

Remarque. Il est possible de faire mieux :

- en univers étiqueté : compter les cycles revient à considérer le cycle comme étant une séquence,
- en univers non étiqueté : on peut décomposer les cycles en cycles primitifs [25].

2.3 Algorithme de comptage de structures

Nous pouvons maintenant écrire une fonction permettant de compter le nombre $(a)_n$ de structures de taille n d'un ensemble de structures associé à un non terminal A d'une spécification. Cette fonction utilise les fonctions `premier_forme` et `suiwant_forme` présentées page 44 :

```

FUNCTION compte_structures(A: nonterminal, n: INTEGER) : INTEGER
  IF A → a, a↔Epsilon THEN
    IF n = 0 THEN RETURN 1; ELSE RETURN 0; END_IF;
  IF A → a, a↔Atom THEN
    IF n = 1 THEN RETURN 1; ELSE RETURN 0; END_IF;
  ELSE_IF
    f:=premier_forme(A,n);
    compte_f:=0;
    WHILE f ≠ nil DO
      compte_f:=compte_f+compte_forme(f);
      f:=suiwant_forme(f);
    END_WHILE;
    RETURN compte_f;
  END_IF
END_FUNCTION;

```

2.4 Conditions supplémentaires sur les structures

La fonction de calcul du nombre de structures utilisant les formes fait double emploi avec celle présentée au chapitre précédent, et est en pratique de coût en temps légèrement supérieur. Elle permet cependant

de décomposer davantage le calcul, ce qui permet d'introduire de nouvelles conditions plus précises pour décrire les ensembles de structures.

2.4.1 Arité des multi-constructeurs

Il est possible d'ajouter des conditions d'arité des structures des ensembles décrits par des multi-constructeurs, comme cela a déjà été fait au chapitre précédent. On écrira par exemple $A \rightarrow \text{Set}(B, \text{card} \leq 5)$ pour indiquer que les structures de A doivent être de la forme $\text{set}(o_1, \dots, o_s)$, avec $s \leq 5$. Les fonctions d'énumération des formes s'étendent simplement pour tenir compte de ces contraintes. Par exemple, dans le cas des formes Set , si l'arité des structures est bornée par un entier k , on ajoute les formes dont les composantes sont de taille nulle, et on « saute » les formes dont la condition d'arité n'est pas vérifiée. Par rapport au chapitre précédent, les contraintes peuvent être plus précises : on peut par exemple demander à ce que le nombre de composantes soit un nombre pair ou un nombre premier. Ces extensions sont naturelles une fois les formes introduites, mais techniques. Nous ne les détaillons pas.

2.4.2 Taille des composantes

Il est possible d'ajouter des conditions supplémentaires aux constructeurs utilisés dans les règles de production des spécifications :

$$A \rightarrow \Phi[g](B_1, \dots, B_s)$$

où g est une fonction de paramètre une forme du constructeur Φ et à valeur booléenne.

Les fonctions d'énumération des formes du constructeur Φ sont modifiées, en ajoutant un champ d'information $\text{fcond}=g$ à chaque forme. La fonction de comptage des structures est également modifiée pour ne prendre en compte, pendant la phase de comptage, que les formes f du constructeur Φ pour lesquelles $g(f)$ est vrai :

```

FUNCTION compte_structures(A: nonterminal, n: INTEGER) : INTEGER
  IF A → a THEN
    ERROR("");
  ELSE
    f:=premier_forme(A,n);
    compte_f:=0;
    WHILE f ≠ nil DO
      g:=fcond(f);
      IF g(f) THEN
        compte_f:=compte_f+compte_forme(f);
      END_IF;
      f:=suivant_forme(f);
    END_WHILE;
    RETURN compte_f;
  END_IF;
END_FUNCTION;

```

Exemple. Nous pouvons définir une fonction `ma_condition` qui retourne vrai si et seulement si les composantes d'une forme sont de taille un nombre premier ou de taille inférieure à un :

```

FUNCTION ma_condition(f: forme) : BOOLEAN
  (constructeur, (B1, n1), ..., (Bs, ns)):=f;
  bo:=TRUE;
  i:=1;
  WHILE bo AND i ≤ s DO
    bo:=est_premier(ni) OR ni ≤ 1;
    i:=i+1;
  END_WHILE;
  RETURN bo;
END_FUNCTION;

```

Nous pouvons maintenant considérer l'ensemble des arbres binaires dont les sous-arbres sont tous des feuilles ou de taille un nombre premier :

$$\left\{ \begin{array}{l} A \rightarrow \text{Union}(F, B) \\ B \rightarrow \text{Prod}[\text{ma_condition}](N, A, A) \\ N \rightarrow \mathbf{n} \\ F \rightarrow \mathbf{f} \\ \mathbf{n} \rightsquigarrow \text{Atom} \\ \mathbf{f} \rightsquigarrow \text{Atom} \end{array} \right.$$

2.5 Rang inverse d'une forme

Le problème du rang inverse général d'un ensemble de structures est ramené à un problème plus simple, celui du rang inverse d'une forme ; par analogie avec le cas général, il est possible de définir une bijection entre les structures de taille n appartenant à une forme f et les entiers de l'intervalle $[1 \dots (f)_n]$. L'entier k_f associé à une structure s de la forme f est appelé le *sous-rang* de la structure s dans la forme f .

Pour déterminer à quelle forme une structure s de rang k appartient, et quel est son sous-rang dans cette forme, nous pouvons appliquer l'algorithme suivant, qui utilise les fonctions `premier_forme` et `suitant_forme` présentées page 44 :

Algorithme

- *Données* : un non terminal A , un entier k et un entier n ,
- *Résultat* : la forme f qui contient la structure de taille n et de rang k dans l'ensemble de structures associé au non terminal A , le sous-rang k_f de la structure dans cette forme, et le nombre compte_f de structures dans cette forme.

```

FUNCTION cherche_forme(k: INTEGER, A: nonterminal, n: INTEGER) : (INTEGER, forme)
  f:=premier_forme(A,n);
  compte_f:=compte_forme(f);
  k_f:=k;
  WHILE compte_f < k_f DO
    k_f:=k_f-compte_f;
    f:=suitant_forme(f);
    compte_f:=compte_forme(f);
  END_WHILE;
  RETURN (k_f, f, compte_f);
FUNCTION;

```

Proposition 2.5.1

Dans le cas d'une distribution uniforme des structures dans les formes, le coût moyen en temps d'un appel à la fonction `cherche_forme` est $O(N_f(A, n)s)$, où $N_f(A, n)$ est le nombre de formes de taille n du non terminal A , s leur arité. Le coût d'un appel à la fonction `cherche_forme`, pour $k = 1$ et une distribution uniforme des structures dans les formes est en $O(s)$.

Démonstration. Soit $T(n)$ le coût moyen d'un appel à `cherche_forme` pour des structures de taille n . On a :

$$(a)_n T(n) = \sum_{k=1}^{N_f(n)} k \underbrace{O(s)}_{\text{compte_forme}} \underbrace{\frac{(a)_n}{N_f(n)}}_{\text{nombre de structures dans la } k^{\text{e}} \text{ forme}}$$

$$\begin{aligned} T(n) &= O(s) \frac{1}{N_f(n)} \sum_{k=1}^{N_f(n)} k \\ &= O(s) \frac{N_f(n)}{2} \end{aligned}$$

□

Nous nous proposons de résoudre récursivement le problème du rang inverse d'une forme. Pour construire la structure de sous-rang k_f dans la forme f , nous déterminons les rangs de ses composantes directes. Nous cherchons donc à définir une fonction de la forme :

```
FUNCTION calcul_rangs_operandes( $k_f$ : INTEGER,  $f$ : forme) : (INTEGER,...,INTEGER)
```

Le calcul des rangs des composantes diffère selon le type de constructeur considéré et l'univers dans lequel nous nous plaçons. Nous définissons donc des fonctions `calcul_rangs_operandes_prod`, `calcul_rangs_operandes_set` etc., spécifiques à chaque constructeur, qui seront appelées par la fonction `calcul_rangs_operandes` en fonction du constructeur de la forme f manipulée.

2.5.1 Univers non étiqueté

2.5.1.1 Formes *Union*

Regardons tout d'abord le cas le plus simple, celui des formes du constructeur *Union*, pour lesquelles il n'y a qu'une seule composante. Le rang de la composante est le même que le sous-rang dans la forme.

```
FUNCTION calcul_rangs_operandes_union( $k_f$ : INTEGER,  $f$ : forme) : (INTEGER)
  RETURN ( $k_f$ );
END_FUNCTION;
```

2.5.1.2 Formes *Prod*

Il est possible de calculer directement les rangs des composantes des formes *Prod*. Trouver les rangs des composantes d'une forme produit ($Prod, (B_1, n_1), \dots, (B_s, n_s)$) revient à résoudre un nouveau problème de rang inverse, mais portant sur toutes les suites d'entiers de la forme :

$$\begin{aligned} &(r_1, \dots, r_s) \\ &\text{avec } 1 \leq r_i \leq (b_i)_{n_i}, i = 1, \dots, s \end{aligned}$$

Il y a $\prod_{i=1}^s (b_i)_{n_i}$ suites ainsi définies, et nous définissons l'ordre suivant sur ces suites :

$$(r_1, \dots, r_m) \prec (q_1, \dots, q_m) \text{ si } \begin{cases} r_i = q_i, i \leq u \\ r_{u+1} < q_{u+1} \end{cases}$$

Le problème consiste, étant donné un entier $k \in [1, \prod_{i=1}^s (b_i)_{n_i}]$, à trouver la suite d'entiers (r_1, \dots, r_s) correspondant à la k^e suite suivant l'ordre défini. La solution (r_1, r_2) du problème pour $s = 2$ et deux entiers quelconques n_1 et n_2 est :

$$\begin{aligned} r_2 &= 1 + \lfloor \frac{k-1}{(b_1)_{n_1}} \rfloor \\ r_1 &= 1 + (k-1) \bmod (b_1)_{n_1} \end{aligned} \tag{2.2}$$

En généralisant au cas $s \geq 2$, on obtient, en appliquant le même principe, la fonction de calcul des rangs des composantes `calcul_parties`, définie par :

```
FUNCTION calcul_parties( $k_f$ : INTEGER,  $t_1, \dots, t_s$  : INTEGER):(INTEGER,...,INTEGER)
   $r := k_f$ ;
  FOR  $i$  FROM 1 TO  $s-1$  DO
```



```

     $k_i := 1 + (r - 1) \bmod t_i$ ;
     $r := 1 + \lfloor \frac{(r-1)}{t_i} \rfloor$ ;
  END_FOR;
   $k_s := r$ ;
  RETURN ( $k_1, \dots, k_s$ );
END_FUNCTION;

```

Cette fonction sera utilisée également dans le calcul des rangs des composantes pour les autres types de forme. La fonction pour le cas du produit est simplement :

```

FUNCTION calcul_rangs_operandes_prod( $k_f$ : INTEGER,  $f$ : forme) : (INTEGER, ..., INTEGER)
  ( $Prod, (A_1, n_1), \dots, (A_s, n_s)$ ):= $f$ ;
  RETURN calcul_parties( $k_f, (a_1)_{n_1}, \dots, (a_s)_{n_s}$ );
END_FUNCTION;

```

2.5.1.3 Formes *Set*

Avant de résoudre le problème complet pour les formes *Set*, regardons tout d'abord le cas des formes *Set* dont toutes les composantes ont la même taille. Trouver les rangs des composantes d'une telle forme *Set*, (A, n, m) revient à s'intéresser au problème de rang inverse sur les suites d'entiers, de longueur m vérifiant la propriété suivante :

$$(k_1, \dots, k_m) \\ 1 \leq k_1 \leq k_2 \leq \dots \leq k_m \leq (a)_n$$

Il y a $N(m, (a)_n) = \binom{(a)_n + m - 1}{m}$ suites ainsi définies, et nous définissons l'ordre lexicographique sur celles-ci :

$$(r_1, \dots, r_m) < (q_1, \dots, q_m) \\ r_i = q_i, i \leq u \\ r_{u+1} < q_{u+1} \quad (2.3)$$

Le problème consiste, étant donné un entier $k \in [1, N(m, (a)_n)]$, à trouver (k_1, \dots, k_m) correspondant à la k^e suite d'entiers suivant l'ordre défini. Regardons le cas où $t = (a)_n$ et m valent 3, on obtient :

$$\left. \begin{array}{l} 1 \ 1 \ 1 \\ 1 \ 1 \ 2 \\ 1 \ 1 \ 3 \\ 1 \ 2 \ 2 \\ 1 \ 2 \ 3 \\ 1 \ 3 \ 3 \end{array} \right\} N(m-1, t) = \binom{t+m-2}{m-1}$$

$$\left. \begin{array}{l} 2 \ 2 \ 2 \\ 2 \ 2 \ 3 \\ 2 \ 3 \ 3 \end{array} \right\} N(m-1, t-1) = \binom{t+m-3}{m-1}$$

$$3 \ 3 \ 3 \left. \right\} N(m-1, t-2) = \binom{t+m-4}{m-1}$$

Soit $t = (a)_n$. L'algorithme proposé consiste à chercher l'entier i minimal tel que $k \leq \sum_{j=0}^i N(m-1, t-j)$, qui correspond alors au terme k_1 de la suite cherchée. Ensuite, on appelle récursivement l'algorithme pour déterminer la suite de longueur $m-1$ dont les composantes sont de taille inférieure à $t-i$, de rang $k - \sum_{j=0}^{i-1} N(m-1, t-j)$. On ajoute ensuite l'entier i aux éléments de la suite obtenue après l'appel récursif, pour assurer que les termes k_2, \dots, k_m soient bien supérieurs à i . Cela conduit à l'algorithme suivant pour le calcul des rangs des composantes des formes *Set* ayant m composantes de taille inférieure à t :

```

FUNCTION calcul_rangs_operandes_Δ( $k$ : INTEGER,  $t$ : INTEGER,  $m$ : INTEGER) : (INTEGER, ..., INTEGER)
  IF  $m = 1$  THEN
    RETURN ( $k$ );

```

```

ELSE
  i:=0;
  b:=( $\binom{t-i+m-2}{m-1}$ );
  kp:=k;
  WHILE kp > b DO
    i:=i+1;
    kp:=kp-b;
    b:=( $\binom{t-i+m-2}{m-1}$ );
  END_WHILE
  k1:=i+1;
  (k2,...,km):=calcul_rangs_operandes_Δ(kp,t-i,m-1);
  RETURN (k1,i+k2,...,i+km);
END_IF;
END_FUNCTION

```

Nous pouvons vérifier que nous obtenons le résultat cherché pour $t = 3$ et $m = 3$:

```

> seq(calcul_rangs_operandes_delta(i,3,3),i=1..10);
[1, 1, 1], [1, 1, 2], [1, 1, 3], [1, 2, 2], [1, 2, 3], [1, 3, 3], [2, 2, 2],
[2, 2, 3], [2, 3, 3], [3, 3, 3]

```

Nous pouvons maintenant traiter le cas des formes *Set* quelconques. Nous avons la relation suivante dans le cas d'une forme $f = (Set, (A, n_1, m_1), \dots, (A, n_s, m_s))$:

$$(f)_n = \prod_{i=1}^s \Delta(A, n_i, m_i)$$

avec $\Delta(A, n, m) = \binom{m + \binom{a}{n} - 1}{m}$

Le terme $\Delta(A, n_i, m_i)$ correspond à la contribution des m_i structures de taille n_i . Nous pouvons considérer les m_i structures de taille n_i comme un seul et même ensemble de structures de taille $n_i m_i$, contenant $\Delta(A, n_i, m_i)$ structures. On cherche alors les rangs (k_1, \dots, k_s) des composantes d'un ensemble dont toutes les composantes sont différentes. Ceci revient à traiter le même problème que dans le cas des formes *Prod*. La valeur k_i obtenue correspond ensuite au rang dans la forme *Set* ayant m_i composantes de même taille n_i , et on applique alors l'algorithme `calcul_rangs_operandes_Δ`. On en déduit l'algorithme de calcul suivant pour le rang des composantes des formes *Set* :

```

FUNCTION calcul_rangs_operandes_set(kf: INTEGER, f: forme) : (INTEGER,...,INTEGER)
  (Set, (A, n1, m1), ..., (A, ns, ms)):=f;
  (k1,...,ks):=calcul_parties(kf,Δ(A, n1, m1), ..., Δ(A, ns, ms));
  FOR i FROM 1 TO s DO
    (k(i,1), ..., k(i,mi)):=calcul_rangs_operandes_Δ(ki, (a)ni, mi);
  END_FOR;
  RETURN (k(1,1), ..., k(1,m1), ..., k(s,1), ..., k(s,ms));
END_FUNCTION;

```

2.5.1.4 Formes *Cycle*

Nous appliquons le même principe que celui présenté lors du comptage et du calcul des sous-rangs des formes *Cycle*, à savoir que nous ramenons le problème à celui des formes produit, en éliminant les structures qui ne sont pas représentant. Pour une forme donnée, nous calculons les rangs des composantes comme s'il s'agissait d'une forme produit, puis nous testons si les rangs obtenus correspondent bien à des structures cycles représentant :

```

FUNCTION calcul_rangs_operandes_cycle(kf: INTEGER, f: forme)
  Cycle, (B, n1), ..., (B, ns):=f;
  kc:=0;
  kp:=0;

```

```

WHILE  $kc \neq k_f$  DO
   $k_p := k_p + 1$ ;
   $(k_1, \dots, k_s) := \text{calcul\_parties}(k_p, ((b_1)_{n_1}, \dots, (b_s)_{n_s}))$ ;
  IF  $\text{est\_representant\_cycle}(((k_1, n_1), \dots, (k_s, n_s)))$  THEN
     $kc := kc + 1$ ;
  END_IF
END_WHILE;
RETURN  $(k_1, \dots, k_s)$ ;
END_FUNCTION

```

2.5.2 Univers étiqueté

Le principe des algorithmes de calcul des rangs est similaire en univers étiqueté, si l'on ne prend pas en compte les étiquettes. Regardons par exemple le cas des formes produit :

```

FUNCTION calcul_rangs_operandes_prod( $k_f$ : INTEGER,  $f$ : forme) : (INTEGER, ..., INTEGER)
   $(Prod, (A_1, n_1), \dots, (A_s, n_s)) := f$ ;
   $k' := k_f / \binom{n_1 + \dots + n_s}{n_1, \dots, n_s}$ ;
  RETURN calcul_parties( $k'$ ,  $((a_1)_{n_1}, \dots, (a_s)_{n_s})$ );
END_FUNCTION;

```

En univers étiqueté, les autres constructeurs se ramènent au cas du produit.

2.6 Algorithme de rang inverse

Nous savons maintenant trouver la forme correspondant à une structure de rang k , puis calculer les rangs des composantes de la structure. Ceci permet de donner un premier algorithme de rang inverse. Nous introduisons tout d'abord une structure de donnée, l'*arbre de rang inverse* d'une structure s :

Définition 2.6.1 (Arbre de rang inverse)

L'arbre de rang inverse d'une structure s est soit :

- une *feuille* : couple (A, \mathbf{t}) , tel que $A \rightarrow \mathbf{t}$,
- un *nœud* : un n -uplet de la forme $[k_f, f, \text{compte}_f, (k_1, \dots, k_s), (a_1, \dots, a_s)]$, où f est la forme de s , compte_f le nombre de structures dans f , k_f le sous-rang de s dans f avec $1 \leq k_f \leq \text{compte}_f$, k_i le rang de la i^{e} composante de s , et a_i l'arbre de rang inverse de la i^{e} composante de s . •

À partir de la donnée de l'arbre de rang inverse d'une structure s , il est possible de construire la structure s à l'aide de la fonction suivante :

```

FUNCTION structure( $a$ : arbre_rang_inverse) : structure
  IF  $a$  OF KIND  $(A, \mathbf{t})$  THEN
    RETURN  $\mathbf{t}$ ;
  ELSE
     $[k_f, f, \text{compte}_f, (\dots), (a_1, \dots, a_s)] := a$ ;
     $(Constructeur, (\dots), \dots, (\dots)) := f$ ;
    IF  $Constructeur = Union$  THEN
      RETURN structure( $a_1$ );
    ELSE
      RETURN constructeur(structure( $a_1$ ), ..., structure( $a_s$ ));
    END_IF;
  END_IF;
END_FUNCTION;

```

Proposition 2.6.2

Le coût en temps de l'algorithme `structure` est linéaire en le nombre de nœuds internes et de feuilles de l'arbre de rang inverse.

Le principe de l'algorithme permettant de construire l'arbre de rang inverse d'une structure s , à partir de la donnée du rang k de cette structure, est le suivant. On cherche tout d'abord la forme de la structure s à construire, à l'aide de l'algorithme `cherche_forme` présenté en section 2.5. Cette étape utilise la fonction `compte_forme`, qui permet de compter le nombre de structures dans une forme. Ensuite on calcule les rangs des composantes de la forme à l'aide de la fonction `calcul_rangs_operandes`, et on appelle récursivement l'algorithme sur chacune des composantes.

Algorithme (Rang Inverse)

- *Données* : la taille n et le non terminal A d'une spécification de $\hat{\Omega}$ ou Ω , d'une structure de rang k .
- *Résultat* : l'arbre de rang inverse de la structure de rang k de taille n appartenant à l'ensemble de structures de A .

```

FUNCTION rang_inverse( $k$ : INTEGER,  $A$ : nonterminal,  $n$ : INTEGER) : arbre_rang_inverse
  IF  $A \rightarrow t$  THEN
    RETURN ( $A, t$ );
  ELSE
    ( $k_f, f, \text{compte}_f$ ):=cherche_forme( $k, A, n$ );
    RETURN rang_inverse_forme( $k_f, f, \text{compte}_f$ );
  END_IF;
END_FUNCTION;

FUNCTION rang_inverse_forme( $k_f$ : INTEGER,  $f$ : forme,  $\text{compte}_f$ : INTEGER) : arbre_rang_inverse
  (Constructeur, ( $A_1, n_1$ ), ..., ( $A_s, n_s$ )):= $f$ ;
  ( $k_1, \dots, k_s$ ):=calcul_rangs_operandes( $k_f, f$ );
  FOR  $i$  FROM 1 TO  $s$  DO
     $a_i$ :=rang_inverse( $k_i, A_i, n_i$ )
  END_FOR;
  RETURN [ $k_f, f, \text{compte}_f, (k_1, \dots, k_s), (a_1, \dots, a_s)$ ];
END_FUNCTION;

```

Proposition 2.6.3 (Complexité de l'algorithme de rang inverse)

Soit une spécification telle que pour toute taille n et tout non terminal, les structures sont distribuées uniformément dans les formes. Alors le coût moyen d'un appel à la fonction `rang_inverse` pour un non terminal A et une taille n est :

$$T(n) \leq \underbrace{O(sN_f(n))}_{\text{cherche_forme}} + \underbrace{O(s)}_{\text{calcul_rangs_operandes}} + T(n_1) + \dots + T(n_s)$$

où $N_f(n)$ est le nombre de formes de taille n du non terminal A , n_1, \dots, n_s les tailles des composantes de la structure. Pour la construction de l'arbre de rang inverse d'une spécification ne contenant que des constructeurs *Union* ou *Prod*, on obtient :

$$T(n) \leq O(ns \max(\underbrace{s}_{\text{nombre de formes Union}}, \underbrace{\binom{n+s-1}{n}}_{\text{nombre de formes Prod}})) = O(sn^s)$$

Nous aurions très bien pu nous passer, pour ce premier algorithme, de la structure de donnée intermédiaire d'arbre de rang inverse. Son introduction est destinée à faciliter la description de l'algorithme de rang inverse incrémental présenté maintenant.

2.7 Algorithme de rang inverse incrémental

Une des applications du rang inverse est l'énumération de toutes les structures d'un ensemble donné. Il est possible dans cette optique de réaliser un algorithme, qui à partir d'une structure, construit celle de rang suivant, en effectuant uniquement les calculs sur les parties différentes des deux structures. L'algorithme suivant met en œuvre ce principe en ne calculant que les parties modifiées de l'arbre de rang inverse des structures :

```

FUNCTION suivant_rang_inverse(a: arbre_rang_inverse) : arbre_rang_inverse
  IF a OF KIND (A,t) THEN
    ERROR("(")"la structure de rang suivant n'existe pas");
  ELSE
    [kf,f,comptef,(...),(...)]:=a;
    kf:=kf+1;
    WHILE comptef < kf DO
      f:=suivant_forme(f);
      kf:=kf-comptef;
      comptef:=compte_forme(f);
    END_WHILE;
    IF kf≠1 THEN
      RETURN suivant_rang_inverse_forme(a);
    ELSE
      RETURN rang_inverse_forme(kf,f,comptef);
    END_IF;
  END_IF;
END_FUNCTION;

```

Dans cet algorithme, on n'appelle l'algorithme de rang inverse général que lorsque l'on « change de forme », c'est-à-dire uniquement quand on calcule l'arbre de rang inverse de la première structure d'une forme. Dans les autres cas, on appelle l'algorithme de rang inverse incrémental spécifique à chaque forme.

Remarque. Il est possible d'appliquer l'algorithme suivant si l'on ne peut pas trouver d'algorithme spécifique pour un constructeur. On perd cependant en temps de calcul, du fait de la présence de la boucle **for** qui parcourt chaque composante.

```

FUNCTION suivant_rang_inverse_forme(a: arbre_rang_inverse) : arbre_rang_inverse
  IF a OF KIND (A,t) THEN
    ERROR("(")"la structure de rang suivant n'existe pas");
  ELSE
    [kf,f,comptef,(prec_k1,...,prec_ks),(prec_a1,...,prec_as)]:=a;
    (Constructeur,(A1,n1),..., (As,ns)):=f;
    kf:=kf+1;
    (k1,...,ks) :=calcul_rangs_operandes(kf,f);
    FOR i FROM 1 TO s DO
      IF ki=prec_ki THEN
        ai:=prec_ai;
      ELSE_IF ki=prec_ki+1 THEN
        ai:=suivant_rang_inverse(prec_ai)
      ELSE
        ai:=rang_inverse(ki,Ai,ni);
      END_IF
    END_FOR;
    RETURN [kf,f,comptef,(k1,...,ks),(a1,...,as)];
  END_IF;
END_FUNCTION

```

2.7.1 Formes Union

Dans le cas des formes union, il n'y a qu'une seule composante, et on obtient directement :

```

FUNCTION suivant_rang_inverse_union(a: arbre_rang_inverse) : arbre_rang_inverse
  [kf,f,comptef,(k1),(a1)]:=a;

```

```

(Union, (A1, n1)) := f;
kf := kf + 1;
k1 := k1 + 1;
a1 := suivant_rang_inverse(a1);
RETURN [kf, f, comptef, (k1), (a1)];
END_FUNCTION

```

2.7.2 Formes Prod

L'algorithme pour les formes *Prod* s'appuie sur la relation (2.2), qui donne l'ordre des structures dans la forme.

```

FUNCTION suivant_rang_inverse_prod(a: arbre_rang_inverse) : arbre_rang_inverse
[kf, f, comptef, (k1, ..., ks), (a1, ..., as)] := a;
(Prod, (A1, n1), ..., (As, ns)) := f;
kf := kf + 1;
h := 1;
WHILE kh = (a)nh DO
  kh := 1;
  ah := rang_inverse(1, Ah, nh);
  h := h + 1;
END_WHILE;
kh := kh + 1;
ah := suivant_rang_inverse(ah);
RETURN [kf, f, comptef, (k1, ..., ks), (a1, ..., as)];
END_FUNCTION

```

2.7.3 Formes Set

Nous utilisons le même principe que celui utilisé pour les formes, pour joindre à l'arbre de rang inverse des informations supplémentaires, qui permettent de conserver les rangs intermédiaires k_1, \dots, k_s (dans un champs d'information **rangs_Δ**), ainsi que les tailles $N_i = \Delta(A, n_i, m_i), i = 1, \dots, s$ (dans un champs d'information **tailles_Δ**). La première partie de la fonction, dans laquelle on cherche les rangs intermédiaires, correspond exactement au cas du produit. Ensuite, on cherche les rangs suivant l'ordre défini en (2.3).

```

FUNCTION suivant_rang_inverse_set(a: arbre_rang_inverse) : arbre_rang_inverse
[kf, f, comptef, (k(1,1), ..., k(1,m1), ..., k(s,1), ..., k(s,ms)), (a(1,1), ..., a(1,m1), ..., a(s,1), ..., a(s,ms))] := a;
(Set, (A, n1), ..., (A, ns)) := f;
(k1, ..., ks) := rangs_Δ(a);
(N1, ..., Ns) := tailles_Δ(a);
kf := kf + 1;
h := 1;
WHILE kh = Nh DO
  kh := 1;
  k(h,1) := 1;
  a(h,1) := rang_inverse(1, Ah, nh);
  FOR i FROM 2 TO mh DO
    k(h,i) := 1;
    a(h,i) := a(h,1);
  END_FOR;
  h := h + 1;
END_WHILE;
kh := kh + 1;
i := mh;
WHILE k(h,i) = (a)nh DO
  i := i - 1;
END_WHILE;

```

```

 $k_{(h,i)} := k_{(h,i)} + 1;$ 
 $a_{(h,i)} := \text{suivant\_rang\_inverse}(a_{(h,i)});$ 
FOR  $j$  FROM  $i + 1$  TO  $m_h$  DO
   $k_{(h,j)} := k_{(h,i)};$ 
   $a_{(h,j)} := a_{(h,i)};$ 
END_FOR;
RETURN [ $k_f, f, \text{compte}_f, (k_{(1,1)}, \dots, k_{(1,m_1)}, \dots, k_{(s,1)}, \dots, k_{(s,m_s)}),$ 
  ( $a_{(1,1)}, \dots, a_{(1,m_1)}, \dots, a_{(s,1)}, \dots, a_{(s,m_s)}$ )];
END_FUNCTION

```

2.7.4 Formes Cycle

Le cas des formes cycle reprend l'algorithme présenté pour le calcul général des rangs. On introduit tout d'abord la fonction `suivant_calcul_parties` :

```

FUNCTION suivant_calcul_parties (( $k_1, \dots, k_s$ ): (INTEGER  $\times \dots \times$  INTEGER),
  ( $n_1, \dots, n_s$ ): (INTEGER  $\times \dots \times$  INTEGER)) : (INTEGER  $\times \dots \times$  INTEGER)
   $h := 1;$ 
  WHILE  $k_h = n_h$  DO
     $k_h := 1;$ 
     $h := h + 1;$ 
  END_WHILE;
   $k_h := k_h + 1;$ 
  RETURN ( $k_1, \dots, k_s$ );
END_FUNCTION

```

On modifie ensuite la fonction décrite précédemment dans le cas général, pour utiliser la fonction `suivant_calcul_parties` à la place de `calcul_parties` :

```

FUNCTION suivant_rang_inverse_cycle ( $a$ : arbre_rang_inverse) : arbre_rang_inverse
  [ $k_f, f, \text{compte}_f, (k_1, \dots, k_s), (a_1, \dots, a_s)$ ] :=  $a$ ;
  Cycle, ( $A, n_1$ ), ..., ( $A, n_s$ ) :=  $f$ ;
   $k_f := k_f + 1;$ 
   $kc := k_f - 1;$ 
   $bo := \text{TRUE};$ 
  WHILE  $bo$  DO
    ( $k_1, \dots, k_s$ ) := suivant_calcul_parties (( $k_1, \dots, k_s$ ), ( $n_1, \dots, n_s$ ));
    IF est_representant_cycle (( $k_1, n_1$ ), ..., ( $k_s, n_s$ )) THEN
       $h := 1;$ 
      WHILE  $k_h = 1$  DO
         $a_h := \text{rang\_inverse}(1, A_h, n_h);$ 
         $h := h + 1;$ 
      END_WHILE;
       $a_h := \text{rang\_inverse\_suivant}(a_h);$ 
       $bo := \text{FALSE}$ 
    END_IF
  END_WHILE;
  RETURN
END_FUNCTION

```

2.8 Mise en œuvre

Les algorithmes proposés dans ce chapitre sont implantés en **Maple**. Nous pouvons illustrer leur utilisation dans le cas des arbres généraux non planaires, dont nous rappelons la définition :

$$\left\{ \begin{array}{l} A \rightarrow \text{Prod}(N, B) \\ B \rightarrow \text{Set}(A) \\ N \rightarrow \mathbf{n} \\ \mathbf{n} \rightsquigarrow \text{Atom} \end{array} \right.$$

Pour générer le troisième arbre de taille 5, on écrit :

```
> read('rang_inverse.mp'):
> spec:={A=Prod(N,B), B=Set(A), N=n, n=Atom}:

> specification_courante(spec):
> rang_inverse(3,A,5);

    Prod(n,Set(Prod(n,Set(Prod(n,Set(Prod(n,Epsilon))),Prod(n,Epsilon))))))
```

Pour générer tous les arbres de taille 5, on utilise la fonction prédéfinie `rang_inverse_tous` comme suit :

```
> rang_inverse_tous(A,5);
[
Prod(n,Set(Prod(n,Set(Prod(n,Set(Prod(n,Set(Prod(n,Epsilon))))))))) ,
Prod(n,Set(Prod(n,Set(Prod(n,Set(Prod(n,Epsilon),Prod(n,Epsilon)))))) ,
Prod(n,Set(Prod(n,Set(Prod(n,Set(Prod(n,Epsilon))),Prod(n,Epsilon)))) ,
Prod(n,Set(Prod(n,Set(Prod(n,Epsilon),Prod(n,Epsilon),Prod(n,Epsilon)))) ,
Prod(n,Set(Prod(n,Set(Prod(n,Set(Prod(n,Epsilon))))),Prod(n,Epsilon))),
Prod(n,Set(Prod(n,Set(Prod(n,Epsilon),Prod(n,Epsilon))),Prod(n,Epsilon))),
Prod(n,Set(Prod(n,Set(Prod(n,Epsilon))),Prod(n,Set(Prod(n,Epsilon))))),
Prod(n,Set(Prod(n,Set(Prod(n,Epsilon))),Prod(n,Epsilon),Prod(n,Epsilon))),
Prod(n,Set(Prod(n,Epsilon),Prod(n,Epsilon),Prod(n,Epsilon),Prod(n,Epsilon)))]
```

Nous obtenons 9 structures de taille 5, qui sont représentées graphiquement sur la figure 2.1. Les feuilles des arbres correspondent au terme `Prod(n,Epsilon)` dans l'expression du résultat, les nœuds internes au terme `n`. Les 48 arbres de taille 7 sont représentés sur la figure 2.2.

Remarque. On peut vérifier expérimentalement, sur de petites tailles, que l'on obtient bien les mêmes structures que celles générées de façon aléatoire avec `combstruct`. La procédure `test` ci-dessous permet de choisir une représentation normale pour les structures spécifiées avec le constructeur `Set`. Il reste ensuite à générer les structures avec `combstruct` jusqu'à obtenir les 9 structures différentes.

```
> with(combstruct):
> spec:={A=Prod(N,B), B=Set(A), N=n, n=Atom}:
> test:=proc(expr)
  if op(0,expr)=Set then
    op(0,expr)(op(map(test,convert(expr,set))));
  elif op(0,expr)<>string then
    map(test,expr);
  else
    expr;
  fi; end;
> nb:=count([A,spec,unlabelled],size=5):
> res:={}: while nops(res)<>nb do res:=res union {test(draw([A,spec,unlabelled],size=5))}: od:
  lprint(res);
```

En changeant de spécification, on peut énumérer tous les arbres généraux, en considérant que deux arbres sont égaux à une permutation cyclique de leurs sous-arbres près :

```
> spec:={A=Union(F,C), C=Prod(N,B), B=Cycle(A), N=n, n=Atom, F=f, f=Atom}:
> specification_courante(spec):
> rang_inverse_tous(A,7);
```

On obtient tous les arbres de la figure 2.2, avec en plus trois arbres supplémentaires, représentés sur la figure 2.3.

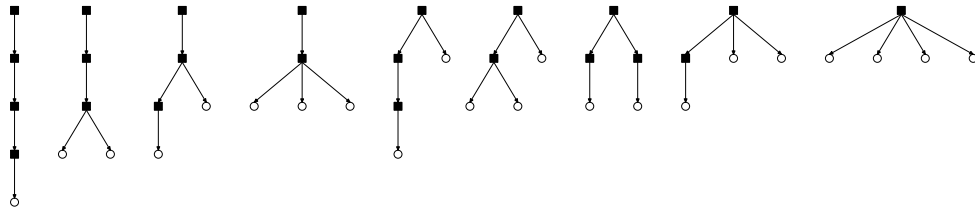


Fig. 2.1: Représentation graphique du résultat de l'algorithme de rang inverse, pour les arbres généraux non planaires de taille 5

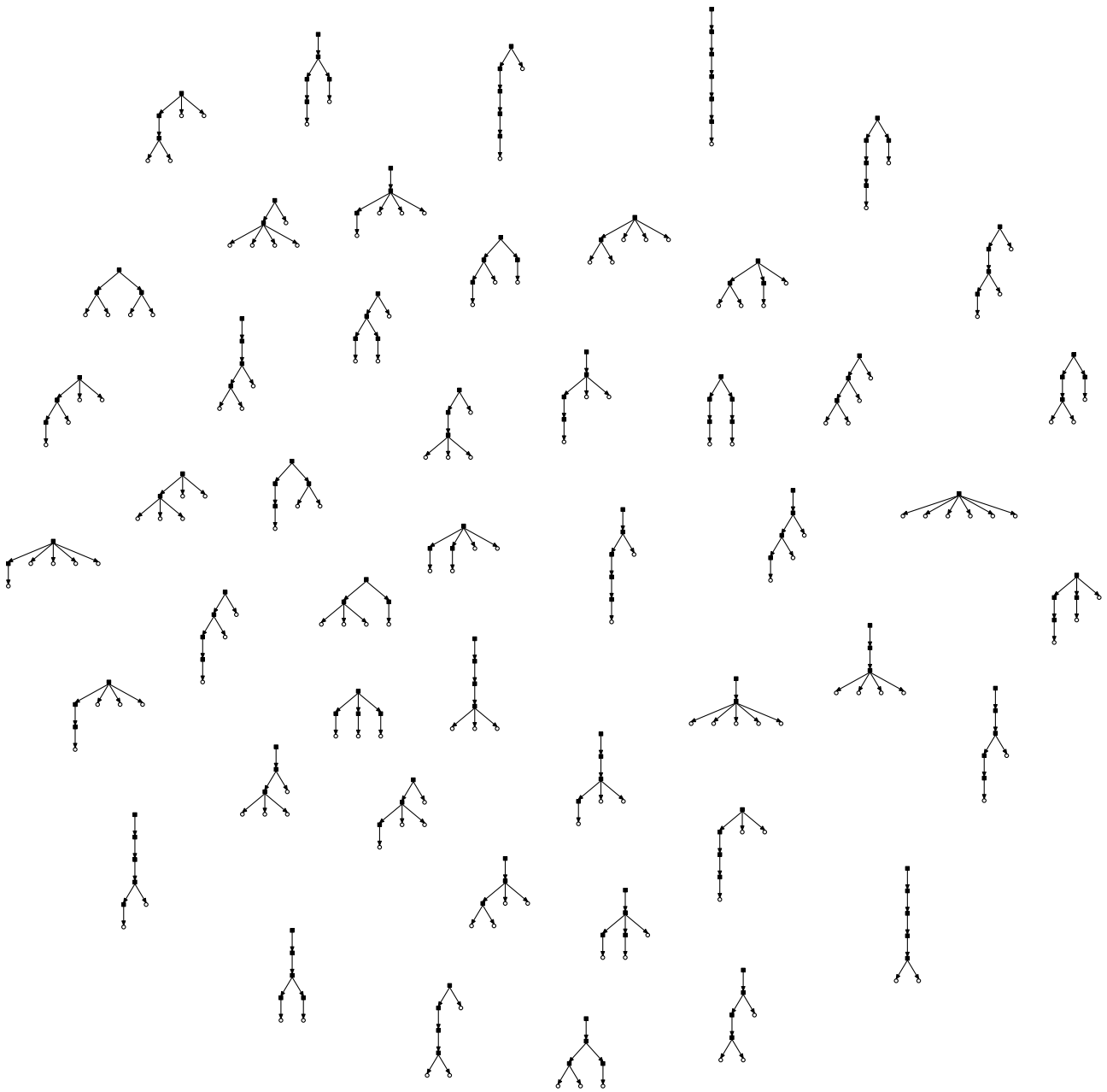


Fig. 2.2: Représentation graphique des 48 arbres généraux non planaires de taille 7

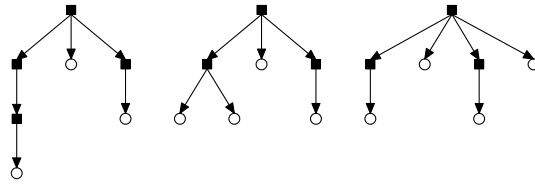


Fig. 2.3: Arbres généraux avec permutation cyclique des fils.

Pour générer les arbres généraux dont les sous-arbres sont soit des feuilles, soit de taille un nombre premier, on peut écrire :

```

ma_condition:=proc(f)
  N:=convert(tailles_operandes_forme(f),table);
  s:=arite_forme(f);
  bo:=true;
  i:=1;
  while bo and i<=s do
    bo:=N[i]<=1 or isprime(N[i]);
    i:=i+1;
  od;
  bo;
end;

> spec:={A=Prod(N,B), B=Set[ma_condition](A), N=n, n=Atom};
> specification_courante(spec);
> rang_inverse_tous(A,7);

```

On obtient 16 arbres, représentés sur la figure 2.4.

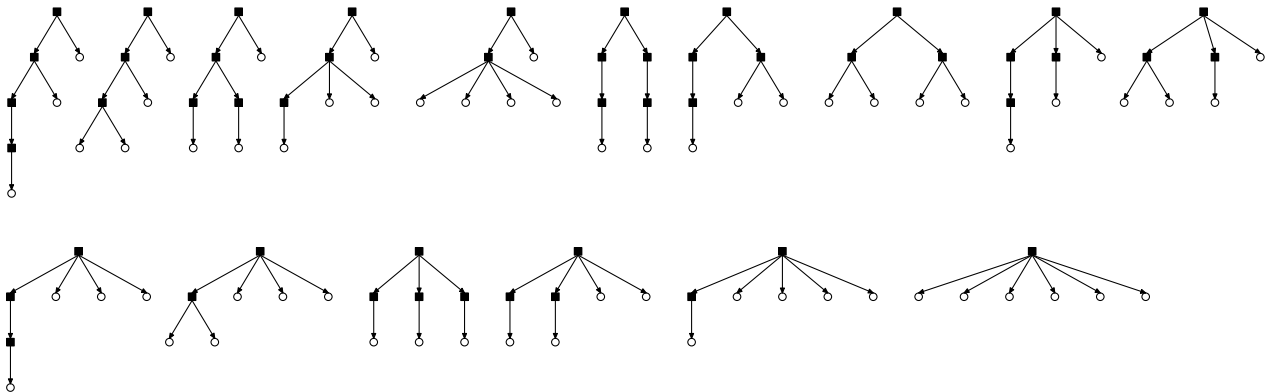


Fig. 2.4: Les 16 arbres généraux non planaires de taille 7 dont les sous-arbres sont des feuilles ou de taille un nombre premier

Dans l'exemple suivant, nous considérons les ensembles de colliers de perles bicolores de longueur au moins 3. Ceci nous permet d'illustrer l'utilisation des conditions de cardinalité ajoutées aux multiconstructeurs.

```

> spec:={A=Set(F), F=Cycle(P,card>=3), P=Union(B,C), C=c, c=Atom, B=b, b=Atom};
> specification_courante(spec);
> rang_inverse_tous(A,6);

```

L'ensemble de structures obtenu est représenté sur la figure 2.5.

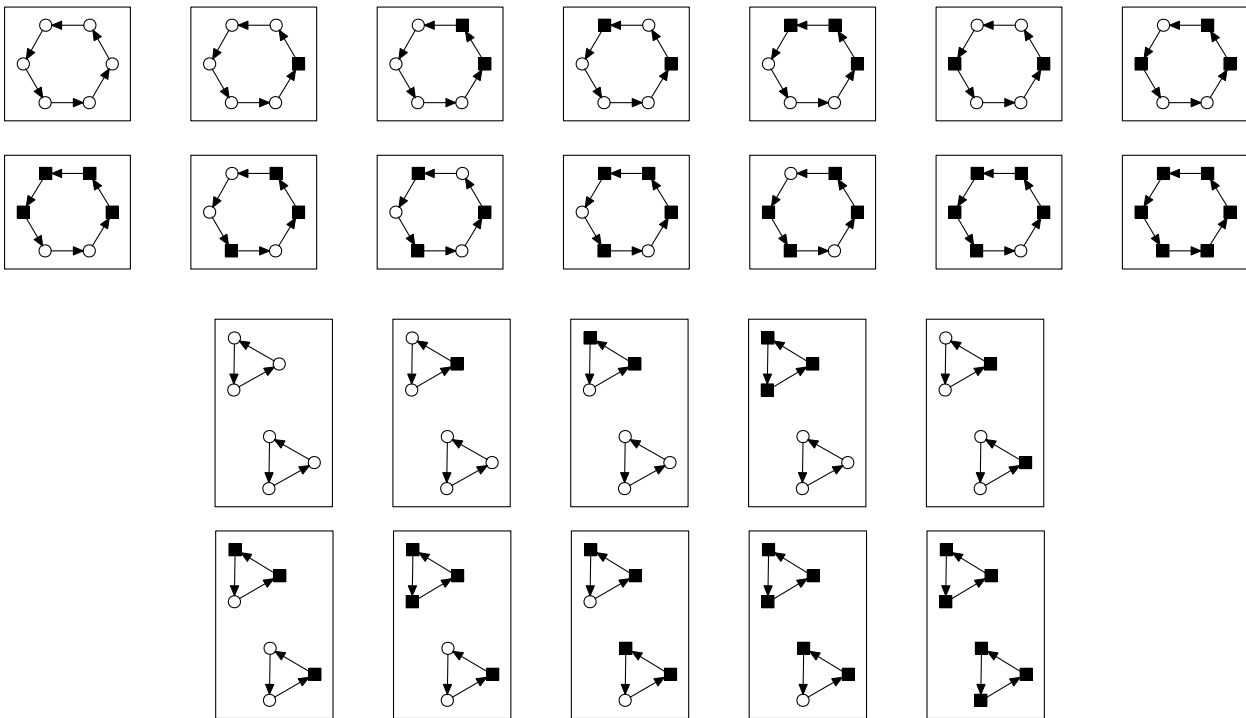


Fig. 2.5: Les 24 ensembles de taille 6 formés de colliers bicolores de longueur au moins 3.

2.9 Conclusion

Nous avons abordé dans ce chapitre le problème du rang inverse sur des ensembles de structures décomposables. Nous avons pour cela introduit la notion de forme d'une structure, qui moyennant la définition d'opérations spécifiques à chaque constructeur utilisé dans les spécifications, nous a permis de proposer un algorithme incrémental systématique, quels que soient les constructeurs utilisés. Les opérations à définir pour chaque constructeur doivent permettre de compter le nombre de structures dans une forme, ainsi que le calcul des rangs des composantes d'une forme, et nous avons décrit ces opérations dans le cas des constructeurs union, produit, multi-ensemble et cycle. Pour ce dernier constructeur, il est possible d'envisager d'améliorer la solution proposée. Nous avons également montré que la notion de forme permettait de définir des ensembles de structures ayant des propriétés autres qu'une taille donnée. L'algorithme de rang inverse proposé dans ce chapitre est implanté en pratique dans le cas de structures non étiquetées, et nous en avons donné quelques exemples d'utilisation.

Chapitre 3

Représentation graphique de graphes orientés et non orientés

CE CHAPITRE CONTIENT une description d’algorithmes de tracé de graphes pour des classes particulières : les arbres enracinés, les graphes orientés acycliques et les graphes non orientés.

Les résultats de ce chapitre sont d’ordres différents. Les premiers concernent la présentation d’algorithmes classiques, étendus pour permettre de tracer des graphes dont les nœuds de taille arbitrairement grande. C’est le cas de l’algorithme de tracé des graphes orientés présenté en section 3.6.2, celui de tracé des graphes non orientés présenté en section 3.5 ainsi que celui de tracé d’arbres enracinés proposé au point 3.3. Pour ces deux premiers algorithmes, le travail présenté se rapproche d’un état de l’art à propos des algorithmes de tracé pour ces classes de graphes, la difficulté étant simplement de l’ordre de la mise en œuvre de ces techniques. L’algorithme de tracé d’arbres enracinés est lui plus original, dans la mesure où il utilise une méthode qui semble de premier abord inutilisable dans le cas de nœuds de tailles différentes. A. Bloesh dans [5], parlant de l’algorithme qu’il propose, écrit ainsi : « *[...], it does not (and indeed cannot) take advantage of the threading techniques of Reingold and Tilford.* ». Notre algorithme est précisément une adaptation de ces techniques, présentées initialement dans le cas d’arbres binaires ayant des nœuds de taille ponctuelle dans [44], au cas d’arbres généraux ayant des nœuds de taille quelconque. Au final, il est très proche de l’algorithme proposé par Reingold et Tilford, et peut être considéré comme une simplification de ce dernier tout en permettant de traiter une classe d’arbres plus étendue.

Les autres algorithmes présentés, bien qu’ils s’inspirent également d’algorithmes existants, sont également originaux et offrent de nouvelles possibilités de représentation graphique. L’algorithme présenté en section 3.6.1 met ainsi en œuvre les principes de l’algorithme de tracé de graphes non orientés présenté en section 3.5, mais s’applique au cas de graphes orientés. Notre apport principal dans cet algorithme est que nous proposons des techniques permettant le tracé incrémental de ces graphes. Cette approche n’est pas utilisée à notre connaissance, étant donné qu’elle est moins performante, dans le cas d’un tracé dans le plan, que l’algorithme classique présenté en section 3.6.2. Elle donne néanmoins de bons résultats pour des graphes de taille modeste dans le cas d’un tracé en trois dimensions. Enfin l’algorithme **FRCC** que nous proposons au point 3.5.3 permet, partant d’une configuration initiale, de conserver les propriétés de partitionnement des nœuds dans le graphe. À notre connaissance, cette possibilité n’est proposée par aucun autre algorithme existant. Elle permet par exemple de faire apparaître des propriétés de symétrie dans les graphes planaires comme nous l’illustrons sur des exemples en fin de chapitre.

Les algorithmes de ce chapitre sont présentés en détail et sont effectivement implantés dans les logiciels **CGraph** ou **Padnon** qui seront présentés au chapitre 5. Nous illustrons leur utilisation sur plusieurs exemples.

3.1 Le problème de la représentation graphique des graphes

Les cours de théorie des graphes commencent en général avec l’exemple suivant. Dans la cité prussienne de Königsberg, située sur la rivière Pregel, représentée graphiquement sur la figure 3.1, est-il possible,

partant d'un point, de traverser exactement une fois chaque pont et de revenir au point de départ? Euler a résolu ce problème en le ramenant à celui de la recherche d'un chemin eulérien dans le graphe représenté sur la figure 3.2. Les arêtes correspondent aux ponts et les nœuds aux îles et aux berges.

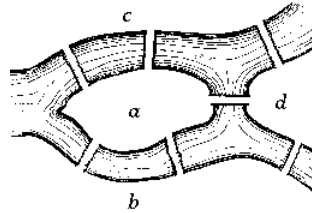


Fig. 3.1: *Le problème des ponts de la ville de Königsberg*

Dans ce problème, nous avons tout d'abord converti les données de la figure de départ en un graphe, puis nous l'avons représenté schématiquement sous la forme d'un dessin pour aider au raisonnement. Une grande variété de problèmes peuvent être posés de façon similaire, en termes d'opérations sur les graphes. On peut citer par exemple les problèmes de routages dans les réseaux, les problèmes de planning ou encore d'allocation de registres par un compilateur. Obtenir une représentation du graphe s'avère être un outil précieux d'aide à l'étude de ces graphes. S'il est relativement simple d'obtenir à la main un dessin satisfaisant pour des graphes de petite taille, il devient par contre indispensable de disposer de techniques automatiques dès que le nombre de nœuds ou d'arêtes devient important.

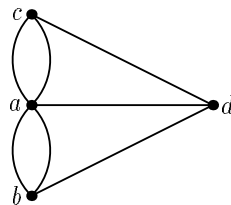


Fig. 3.2: *Le graphe correspondant à la figure 3.1.*

Il n'existe malheureusement pas de représentation idéale pour l'ensemble des graphes, les représentations les plus appropriées dépendant des propriétés que l'on souhaite valoriser. Ceci a conduit à proposer de nombreux algorithmes, dont on peut voir l'étendue dans [13], dédiés à des classes particulières de graphes, ou dédiés à des critères esthétiques particuliers dont certains se sont imposés comme standards. Parmi les premiers résultats, on peut citer le tracé des graphes planaires, pour lesquels il existe une représentation planaire dans laquelle les nœuds sont des points et les arêtes sont rectilignes [22, 54, 59]. Les autres résultats significatifs, en dehors des méthodes de transformation d'un graphe en un graphe planaire, concernant les arbres [44, 61], les graphes orientés [52] ou les graphes non orientés [15], sont plus récents.

Une des difficultés supplémentaires du problème, est que les critères esthétiques que l'on cherche à vérifier sont souvent contradictoires, l'amélioration d'un des critères se faisant au détriment des autres. Même pris individuellement, ils conduisent parfois à des problèmes difficiles.

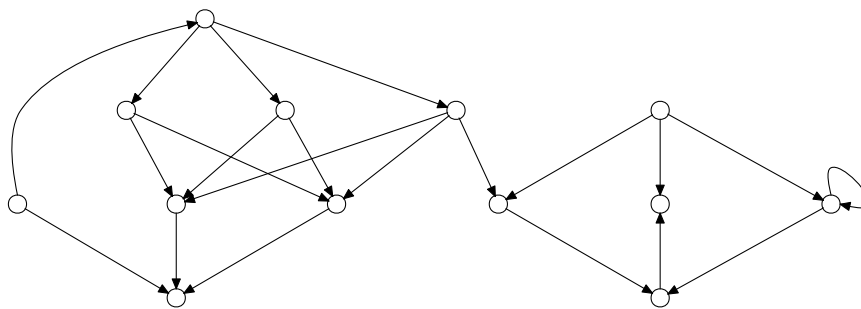


Fig. 3.3: Graphe orienté dessiné par niveau.

Prenons par exemple le cas de la représentation dans le plan des graphes orientés. Une représentation possible consiste à placer les nœuds par niveaux, comme sur la figure 3.3, en cherchant à placer les arêtes dirigées vers le bas. Dans le cas d'un graphe contenant des cycles, on essaye alors de trouver un placement de sorte que le nombre d'arêtes dirigées dans le mauvais sens soit minimal. Ce premier problème est NP-complet [31]. Le problème consistant, partant d'un graphe sans cycle, à placer les nœuds par niveaux et à minimiser le nombre de croisements entre les arêtes l'est également [31]. Des résultats de NP-complétude apparaissent également pour d'autres classes de graphes, comme par exemple les arbres enracinés si l'on cherche, en plus de certains critères particuliers, à minimiser la largeur du dessin obtenu [53].

3.2 Définitions

Nous commençons par introduire les définitions et notations utilisées au cours de ce chapitre. Pour une introduction à la théorie des graphes, on peut se référer au livre de F. Harary [34].

3.2.1 Graphes non-orientés

Définition 3.2.1 (Graphe non orienté)

Un *graphe non orienté* est un couple (N, E) , où N est un ensemble fini de *nœuds* et E un multi-ensemble fini (avec répétitions éventuelles) d'*arêtes non orientées*. Les arêtes non orientées de E sont des paires de nœuds de N et seront notées $a \circ \circ b$. a et b sont les *extrémités* de l'arête. .

Définition 3.2.2 (Sous-graphe)

Un *sous-graphe* d'un graphe non orienté (N, E) , est un couple (N', E') , avec $N' \subseteq N$ et $E' \subseteq E$, dont les extrémités des arêtes de E' sont dans N' . .

Définition 3.2.3 (Voisinage et degré d'un nœud d'un graphe non orienté)

Le *voisinage non orienté* d'un nœud $a \in N$ est l'ensemble (sans répétitions) $V_{\circ \circ}(a) = \{a' \in N, a \circ \circ a'\}$. Le *degré non orienté* d'un nœud $a \in N$, noté $\partial_{\circ \circ}(a)$, est le nombre d'arête qui ont le nœud a comme extrémité. .

Définition 3.2.4 (Chaîne)

Une *chaîne de longueur k* est une suite d'arêtes non orientées e_1, \dots, e_k , telle que l'on peut construire une suite de nœuds $(a_1, b_1, a_2, b_2, \dots, a_k, b_k)$ vérifiant :

- pour tout $i \in [1, k]$, a_i et b_i sont les extrémités de l'arête e_i ,
- pour tout $i \in [1, k - 1]$, $b_i = a_{i+1}$.

a_1 et b_k sont appelés les extrémités de la chaîne. .

Définition 3.2.5 (Graphe connexe)

Un graphe non orienté $G = (N, E)$ est *connexe*, si pour tout couple de nœuds $(a, b) \in N \times N$, il existe une chaîne d'extrémités a et b . .

3.2.2 Graphes orientés

Définition 3.2.6 (Graphe orienté)

Un *graphe orienté* est un couple (N, E) , où N est un ensemble fini de *nœuds* et E un multi-ensemble fini d'*arêtes orientées*. Les arêtes orientées de E sont des couples de nœuds de N et seront notées $a \circ\rightarrow b$. Les nœuds a et b , éléments de N , sont les *extrémités* de l'arête, a est appelé l'*origine*, b la *destination* de l'arête. .

Définition 3.2.7 (Sous-graphe)

Un *sous-graphe* d'un graphe orienté (N, E) , est un couple (N', E') , avec $N' \subseteq N$ et $E' \subseteq E$, dont les extrémités des arêtes de E' sont dans N' . .

Définition 3.2.8 (Voisinage et degré d'un nœud d'un graphe orienté)

On définit, pour un graphe orienté $G = (N, E)$, les notions de voisinage et de degré d'un nœud suivantes :

- le *voisinage entrant orienté* d'un nœud $a \in N$ est l'ensemble $V_{\circ\rightarrow}^-(a) = \{f \in N, f \circ\rightarrow a\}$. Le *degré entrant orienté* d'un nœud $a \in N$, noté $\partial_{\circ\rightarrow}^-(a)$ est le nombre d'arêtes qui ont le nœud a comme destination,
- le *voisinage sortant orienté* d'un nœud $a \in N$ est l'ensemble $V_{\circ\rightarrow}^+(a) = \{s \in N, a \circ\rightarrow s\}$. Le *degré sortant orienté* d'un nœud $a \in N$, noté $\partial_{\circ\rightarrow}^+(a)$ est le nombre d'arêtes qui ont le nœud a comme origine. .

Définition 3.2.9 (Chemin et cycle orienté)

Un *chemin orienté* de *longueur* k est une suite d'arêtes orientées e_1, \dots, e_k , telle que pour tout $i \in [1, k - 1]$, la destination de e_i est égale à l'origine de e_{i+1} . L'origine de l'arête e_1 est l'*origine du chemin*, la destination de l'arête e_k est la *destination du chemin*. Si l'origine et la destination du chemin sont égales, le chemin est un *cycle orienté*. .

Définition 3.2.10 (Graphe connexe)

Un graphe orienté $G = (N, E)$ est *connexe* si, pour tout couple de nœuds $(a, b) \in N \times N$, il existe un chemin orienté d'origine a et de destination b dans le graphe $G' = (N, E \cup E')$, avec si $a \circ\rightarrow b \in E$, alors $b \circ\rightarrow a \in E'$. .

Définition 3.2.11 (Arbre enraciné)

Un *arbre enraciné* est un graphe orienté connexe $A = (N, E)$, dont les nœuds sont de degré rentrant égal à un, excepté pour un nœud unique de degré rentrant nul, que l'on nomme *racine* de l'arbre. Les nœuds de $V_{\circ\rightarrow}^+(a)$ sont les *filles* du nœud a . a est appelé le *père* des nœuds de $V_{\circ\rightarrow}^+(a)$. Un nœud dont le degré sortant orienté est nul est appelé une *feuille*. .

Définition 3.2.12 (Hauteur d'un nœud, hauteur d'un arbre)

La *hauteur d'un nœud* dans un arbre est donnée par la longueur du chemin orienté ayant pour origine la racine de l'arbre et comme destination le nœud considéré. La racine de l'arbre a pour hauteur zéro. La *hauteur de l'arbre* est le maximum des hauteurs des feuilles de l'arbre. .

Définition 3.2.13 (Sous-arbre)

Un *sous-arbre* d'un arbre enraciné A est un sous-graphe de A qui est un arbre enraciné. .

Définition 3.2.14 (Arbre enraciné planaire)

Un arbre enraciné A est *planaire* s'il existe, pour tout nœud a de A , une relation d'ordre totale \prec sur les nœuds de $V_{\circ\rightarrow}^+(a)$. En d'autres termes :

$$\forall a \in A, \forall b, c \in V_{\circ\rightarrow}^+(a), \left| \begin{array}{l} b \prec c \\ b \not\prec c \wedge c \prec b \end{array} \right. \text{ ou } \left| \begin{array}{l} b \prec c \\ b \not\prec c \wedge c \prec b \end{array} \right.$$

Dans le cas où $\partial_{\circ\rightarrow}^+(a) = 2$, si $b, c \in V_{\circ\rightarrow}^+(a)$, et $b \prec c$, alors b est appelé le *fil gauche* de a et c son *fil droit*. .

Définition 3.2.15 (Symétrie d'un arbre enraciné)

Le *symétrique* d'un arbre enraciné planaire $A = (N, E)$ est l'arbre enraciné planaire $\overline{A} = (N, E)$, tel que si $a \prec b$ dans l'arbre A , alors $b \prec a$ dans l'arbre \overline{A} . Deux arbres sont dits *symétriques* s'ils sont le symétrique l'un de l'autre. .

3.3 Tracé d'arbres enracinés planaires

Nous présentons un algorithme pour la représentation des arbres enracinés, basé sur l'algorithme de E. Reingold et J. S. Tilford [44] de tracé d'arbres binaires. Cet algorithme a été proposé initialement pour répondre aux problèmes de dissymétrie qui apparaissent avec les algorithmes proposés antérieurement par J. Vaucher [57] ou C. Wetherell et A. Shannon [61]. Nous proposons une extension de cet algorithme, en nous intéressant au cas général des arbres k -aires dont les nœuds peuvent être de taille quelconque. De plus, nous choisissons de ne plus placer obligatoirement les nœuds verticalement selon le seul critère de leur hauteur dans l'arbre.

3.3.1 Arbres binaires

Avant de décrire l'algorithme de tracé que nous proposons dans le cas des arbres k -aires, voici brièvement le principe de l'algorithme RT proposé par Reingold et Tilford [44] dans le cas des arbres binaires. Les critères esthétiques que l'algorithme cherche à vérifier sont :

- i. les nœuds de même hauteur dans l'arbre doivent être placés sur une même ligne horizontale,
- ii. un fils gauche doit être placé à gauche de son père, un fils droit à sa droite,
- iii. un père doit être centré horizontalement vis-à-vis de ses fils,
- iv. deux sous-arbres identiques doivent être dessinés de la même façon, quelle que soit leur position dans l'arbre. Deux sous-arbres symétriques doivent être dessinés de façon symétrique.

Les critères i, ii et iii sont ceux énoncés par C. Whetherell et A. Shannon [61]. Nous dirons que deux nœuds de même hauteur dans l'arbre, et donc placés sur une même ligne horizontale, sont placés sur un même niveau. Bien que cela ne soit pas dit explicitement, le critère i sous-entend que l'ordre des niveaux est celui sur les hauteurs des nœuds. Le critère iv a été introduit par E. Reingold et J. S. Tilford [44].

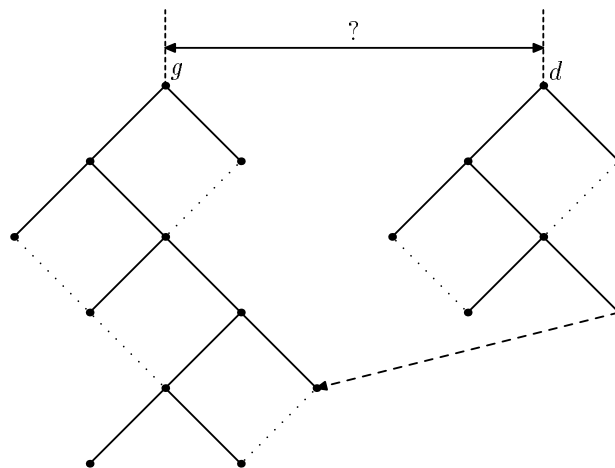


Fig. 3.4: Principe de l'algorithme de tracé d'arbres binaires enracinés

Pour déterminer la position de chaque nœud de l'arbre relativement à la position de son père, l'algorithme RT procède de la façon suivante. Pour dessiner un arbre de racine f , g et d étant les fils de f , avec $g \prec d$, nous appliquons tout d'abord l'algorithme aux deux sous-arbres de racine g et d , ce qui permet de connaître les positions relatives de chaque nœud des deux sous-arbres. Nous cherchons alors à quelle distance minimale doivent se situer les nœuds g et d , d devant être situé à droite de g , pour que les dessins des deux sous-arbres ne se chevauchent pas. Pour déterminer rapidement cette distance optimale, l'idée consiste à ne mesurer la distance qu'entre les nœuds de même niveau dans les deux sous-arbres, en se

limitant aux nœuds situés le plus à droite dans le sous-arbre de racine g , et ceux situés le plus à gauche dans le sous-arbre de racine d . Ceci revient à parcourir simultanément les nœuds situés sur les *contours* des deux sous-arbres, en commençant par les racines des sous-arbres. Nous dirons que le contour droit du sous-arbre de racine g et le contour gauche du sous-arbre de racine d sont les *contours intérieurs*, les autres étant les *contours extérieurs*. Pour permettre le parcours des nœuds situés sur les contours, nous ajoutons, au cours de l'exécution de l'algorithme, des *liens* (en pointillés courts sur la figure), qui permettent d'atteindre directement le nœud suivant du contour. Pour permettre l'ajout de ces liens, une fois déterminée la distance entre g et d , nous suivons également le contour extérieur de chaque sous-arbre, et nous ajoutons un nouveau lien (en pointillé long sur la figure) entre le dernier nœud parcouru sur le contour extérieur du sous-arbre le moins haut, et le nœud de niveau suivant sur le contour intérieur de l'autre sous-arbre. Si les deux sous-arbres ont même hauteur, aucun lien n'est ajouté. Il reste ensuite à placer les nœuds g et d de sorte que leur père f soit centré.

Proposition 3.3.1 (Reingold Tilford [44])

L'algorithme RT de tracé d'arbres binaires est de coût linéaire en temps.

Démonstration. Dans l'algorithme RT, on teste la distance entre les nœuds placés sur les contours des deux sous-arbres, en s'arrêtant dès que la fin d'un des deux contours est atteinte. Si A est un arbre de sous-arbre gauche G , et de sous-arbre droit D , et si $h(B)$ représente la hauteur d'un arbre B , et $n(B)$ son nombre de nœuds on a :

$$\begin{cases} T(\circ) = 0, \\ T(A) = T(G) + T(D) + \min(h(G), h(D)), \end{cases}$$

où \circ représente les arbres réduits à un nœud et T correspond au nombre de fois où l'on teste la distance entre deux points du contour. La solution de cette équation est $T(A) = n(A) - h(A)$, ce qui se montre par induction sur la taille des arbres :

$$\begin{aligned} T(A) &= (n(G) - h(G)) + (n(A) - n(G) - 1 - h(D)) + \min(h(G), h(D)) \\ &= n(A) - 1 - h(G) - h(D) + \min(h(G), h(D)) \\ &= n(A) - (\max(h(G), h(D)) + 1) \\ &= n(A) - h(A). \end{aligned}$$

□

3.3.2 Cas des arbres k -aires avec des nœuds de taille quelconque

L'algorithme RT s'étend simplement au cas k -aire si l'on considère les nœuds comme ponctuels. Par contre, dès que les nœuds ont des tailles arbitraires, et que l'on ne souhaite plus placer les nœuds par niveaux, la méthode proposée ne peut plus être utilisée telle quelle.

Commençons tout d'abord par définir les notations qui seront utilisées pour la description de l'algorithme.

Notation. Nous nous plaçons dans un repère de coordonnées usuel, l'axe des x dirigé vers la droite, l'axe des y dirigé vers le haut. Nous supposons qu'il est possible d'associer les informations suivantes à un nœud n du graphe :

- $x(n), y(n)$: coordonnées du nœud n , absolue ou relative à celle du père de n selon le contexte,
- $hauteur_boite_noeud(n)$ et $largeur_boite_noeud(n)$: hauteur et largeur du plus petit rectangle contenant le tracé du nœud n ,
- $gauche(x, y, n)$, $droit(x, y, n)$, $bas(x, y, n)$ et $haut(x, y, n)$ correspondent aux segments que forme le plus petit rectangle qui contient le tracé du nœud n , lorsque n est placé au point de coordonnées (x, y) ,
- $arete(xa, ya, a, xb, yb, b)$ correspond au segment formé par une arête virtuelle joignant le nœud a et le nœud b , lorsque a et b sont placés aux points de coordonnées (xa, ya) et (xb, yb) ,

- $lien(n)$: nœud lié à n . $lien(n) = \mathbf{nil}$ si aucun nœud n'est relié à n ,
- $x_lien(n)$ et $y_lien(n)$: coordonnées relatives à n du nœud lié à n ,
- $hauteur_boite_sous_arbre(n)$: hauteur du tracé du sous-arbre de racine n .

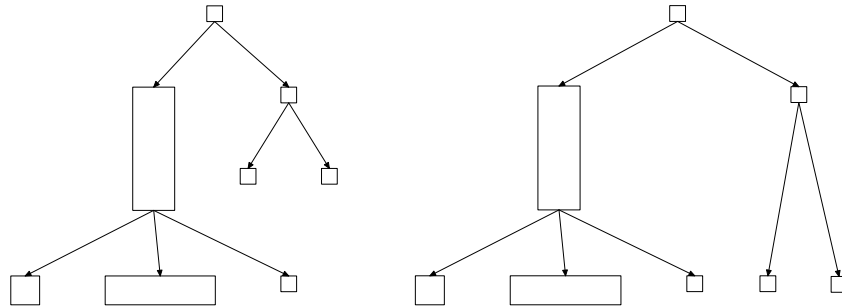


Fig. 3.5: Arbre k -aire avec nœuds de différentes tailles : nœuds placés « au plus près » vs nœuds placés par niveaux.

Comme dans le cas binaire, nous pouvons maintenant donner une liste de critères esthétiques communément admis, qu'il est souhaitable de préserver :

- i. les fils d'un nœud doivent être placés de sorte que le bord haut de chacun des fils soit situé sur une même ligne horizontale,
- ii. les fils d'un nœud doivent être placés de gauche à droite suivant l'ordre \prec ,
- iii. un père doit être centré vis-à-vis de ses fils,
- iv. deux sous-arbres identiques doivent être dessinés de la même façon, quelle que soit leur position dans l'arbre. Deux sous-arbres symétriques doivent être dessinés de façon symétrique,
- v. le bord haut d'un nœud et le bord bas de son père doivent être séparés verticalement par une distance exacte δ_v .

Les critères i à iv sont les mêmes que dans le cas de l'algorithme **RT**. Le critère v correspond à un placement des nœuds *au plus près*. Un critère implicite supplémentaire est qu'il ne doit pas y avoir de chevauchements entre nœuds et arêtes.

Plusieurs algorithmes linéaires ont été proposés pour dessiner suivant ces critères des arbres k -aires dont les nœuds sont de taille quelconque, comme par exemple dans [5], mais aucun à notre connaissance ne tire avantage de la technique des liens de l'algorithme **RT** que nous avons présentée au point 3.3.1. Le principe général de l'algorithme que nous proposons dans le cas k -aire avec des nœuds de taille arbitraire est similaire au cas binaire présenté ci-dessus : pour dessiner un arbre de racine r , on trace tout d'abord les sous-arbres ayant pour racine les fils de r . Ensuite, on place ces sous-arbres relativement les uns aux autres pour éviter les chevauchements de nœuds et d'arêtes, puis on place la racine au milieu des positions de ses fils. L'idée de l'algorithme est également basée sur l'utilisation de liens entre les nœuds pour suivre simplement le contour de l'arbre. Par contre la gestion des liens est différente, et ne nécessite pas de suivre à la fois les contours gauche et droit de chaque sous-arbre comme dans l'algorithme **RT**. Ceci permet de tracer des arbres dont les nœuds ont des tailles différentes, sans placer obligatoirement les nœuds de même hauteur dans l'arbre au même niveau dans le dessin final et assurer ainsi le placement au plus près suivant le critère v.

3.3.2.1 Initialisation

La première étape de l'algorithme consiste à initialiser les liens des nœuds à **nil** :

Algorithme (Initialisation des liens)

```
FOR ALL nœud a DO
  lien(a) := nil;
END_FOR;
```

3.3.2.2 Parcours du contour d'un sous-arbre

Nous supposons connue la fonction **est_feuille** qui retourne **vrai** si un nœud est de degré sortant nul. Ensuite, pour les nœuds de l'arbre qui ne sont pas des feuilles, nous définissons la fonction **fils_gauche**(*n*:nœud), qui retourne le nœud *s* tel que :

$$s \in V_{O \rightarrow O}^+(n) \\ \forall s' \neq s \in V_{O \rightarrow O}^+(n), s \prec s'.$$

De la même façon, la fonction **fils_droit**(*n*:nœud) retourne le nœud *s* vérifiant :

$$s \in V_{O \rightarrow O}^+(n) \\ \forall s' \neq s \in V_{O \rightarrow O}^+(n), s' \prec s.$$

Nous supposons maintenant que les liens ont été placés correctement au cours des étapes précédentes de l'algorithme. Le parcours du contour d'un sous-arbre se fait de la racine vers les feuilles : étant donné un nœud du contour, on détermine le nœud suivant du contour soit en suivant les liens, soit en choisissant le nœud le plus à gauche ou le plus à droite selon le contour choisi. On suppose que les coordonnées d'un nœud du sous-arbre sont données relativement à la position du père de ce nœud. Pour suivre le contour gauche d'un sous-arbre et déterminer les positions absolues des nœuds sur ce contour, on utilisera la fonction décrite ci-dessous. Les paramètres soulignés sont passés par adresse lors de l'appel de la fonction, et correspondent aux coordonnées absolues des nœuds, calculées lors du parcours du contour :

Algorithme (Parcours du contour d'un arbre)

- *Données* : un nœud *n* de position absolue (*x*, *y*) du contour d'un sous-arbre,
- *Résultat* : retourne le nœud du contour dont la hauteur est la hauteur du nœud *n* plus un, ainsi que sa position absolue (*x*, *y*).

```
FUNCTION chemin_gauche_suivant(n:nœud, x:INTEGER, y:INTEGER):nœud
  IF lien(n) ≠ nil THEN
    s := lien(n);
    x := x + x_lien(n);
    y := y + y_lien(n);
  ELSE
    s := fils_gauche(n);
    x := x + x(s);
    y := y + y(s);
  END_IF;
  RETURN s;
END_FUNCTION;
```

On procède de la même façon pour le contour droit en définissant la fonction **chemin_droit_suivant**. Le parcours du contour s'arrête lorsqu'il n'y a plus de lien et qu'il n'y a plus de fils, ou lorsqu'il y a un cycle de longueur deux sur les liens :

```
FUNCTION fin_de_chemin(n:nœud):BOOLEAN
  RETURN (est_feuille(n) AND (lien(n) = nil OR lien(lien(n)) = n));
END_FUNCTION
```

3.3.2.3 Distance entre deux sous-arbres

La fonction `distance_sous_arbres` permet de calculer la *distance horizontale* entre deux sous-arbres dont les positions relatives des nœuds sont connues. La distance horizontale entre deux points est définie comme étant égale à la différence de leurs abscisses si ces deux points ont même ordonnée, et à $+\infty$ sinon. Nous dirons qu'un point est situé sur le tracé d'un sous-arbre s'il est situé sur une arête ou sur le contour d'un nœud. On détermine alors la distance horizontale entre deux sous-arbres en cherchant la distance horizontale minimale entre deux points, l'un situé sur le tracé de a , l'autre sur le tracé de b . Le principe de l'algorithme consiste à suivre le contour des deux sous-arbres à l'aide des fonctions `chemin_gauche_suivant` ou `chemin_droit_suivant` définies ci-dessus, en suivant en priorité le contour dont le bas du dernier nœud atteint est situé le plus haut. Le suivi des contours est interrompu dès que l'extrémité d'un des deux contours est atteinte (fonction `fin_de_chemin`). Lors de ce parcours, on mémorise les derniers nœuds atteints sur chacun des contours, avec leurs positions absolues, ce qui permettra ultérieurement de mettre à jour les liens entre les deux sous-arbres.

On suppose connue une fonction `distance_segment` qui calcule la distance horizontale entre deux segments, en retournant $+\infty$ si aucun couple de points, situés chacun sur les deux segments, n'a même ordonnée.

Algorithme (Distance entre deux sous-arbres)

- *Données* : les racines g et d de deux sous-arbres dont on connaît les positions relatives des nœuds les composant,
- *Résultat* : la valeur de la distance horizontale entre les deux sous-arbres. Le paramètre gt correspond au dernier nœud parcouru sur le contour droit du sous-arbre de racine g , tel qu'il existe un point du sous-arbre de racine d de même ordonnée. (xgt, ygt) correspond aux coordonnées absolues du nœud gt calculées lors de ce parcours. On procède de même pour le sous-arbre droit pour lequel on détermine le point dt et ses coordonnées (xdt, ydt) .

```

FUNCTION distance_sous_arbres (g:nœud, d:nœud, gt:nœud, xgt:INTEGER, ygt:INTEGER,
                               dt:nœud, xdt:INTEGER, ydt:INTEGER) : INTEGER
  g1:=g; xg1:=x(g); yg1:=y(g);
  d1:=d; xd1:=x(d); yd1:=y(d);
  dga:=distance_segment(droit(xg1,yg1,g1),gauche(xd1,yd1,d1));
  WHILE (NOT fin_de_chemin(g1)) AND (NOT fin_de_chemin(d1)) DO
    xg2:=xg1; yg2:=yg1;
    xd2:=xd1; yd2:=yd1;
    g2:=chemin_droit_suivant(g1,xg2,yg2);
    d2:=chemin_gauche_suivant(d1,xd2,yd2);
    dga:=min(dga,distance_segment(arête(xg1,yg1,g1,xg2,yg2,g2),gauche(xd1,yd1,d1)));
    dga:=min(dga,distance_segment(droit(xg1,yg1,g1),arête(xd1,yd1,d1,xd2,yd2,d2)));
    cond_g:=bas(xg1,yg1,g1)≥bas(xd1,yd1,d1);
    cond_d:=bas(xg1,yg1,g1)≤bas(xd1,yd1,d1);
    IF cond_g THEN
      g1:=g2; xg1:=xg2; yg1:=yg2;
    END_IF;
    IF cond_d THEN
      d1:=d2;
      xd1:=xd2;
      yd1:=yd2;
    END_IF;
  END_WHILE;
  gt:=g1; xgt:=xg1; ygt:=yg1;
  dt:=d1; xdt:=xd1; ydt:=yd1;
  WHILE NOT fin_de_chemin(g1) AND haut(droit(xg1,yg1,g1)) > bas(arête(xd1,yd1,d1,xd2,yd2,d2)) DO
    g1:=chemin_droit_suivant(g1,xg1,yg1);
    dga:=min(dga,distance_segment(droit(xg1,yg1,g1),arête(xd1,yd1,d1,xd2,yd2,d2)));
  END_WHILE;

```

```

WHILE NOT fin_de_chemin(d1) AND haut(gauche(xd1, yd1, d1)) > bas(arete(xg1, yg1, g1, xg2, yg2, g2)) DO
  d1:=chemin_gauche_suisvant(d1, xd1, yd1);
  dgd:=min(dgd, distance_segment(arete(xg1, yg1, g1, xg2, yg2, g2), gauche(xd1, yd1, d1)));
END_WHILE;
RETURN dgd;
END_FUNCTION;

```

3.3.2.4 Mise à jour des liens

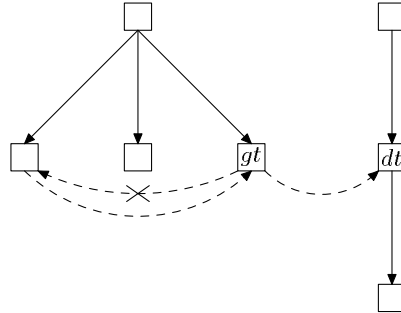


Fig. 3.6: Exemple de mise à jour des liens. *gt* correspond au dernier nœud parcouru sur le contour droit du sous-arbre gauche lors de l'exécution de la fonction *distance_sous_arbres*, *dt* au dernier nœud atteint sur le contour gauche du sous-arbre droit.

Une fois que l'on a déterminé la distance horizontale entre deux sous-arbres, nous pouvons placer ces sous-arbres de sorte qu'aucun nœud ne se chevauche. Il reste ensuite à mettre à jour, dans certains cas, les liens de chacun des sous-arbres pour permettre de parcourir le contour formé par la réunion des deux sous-arbres. Le fait de décider de mettre à jour les liens entre deux sous-arbres ou non sera déterminé ultérieurement.

Algorithme (Mise à jour des liens)

```

PROCEDURE mise_a_jour_des_liens(d:INTEGER, lt:noeud, xlt:INTEGER, ylt:INTEGER,
                                rt:noeud, xrt:INTEGER, yrt:INTEGER)
  cond_l:=fin_de_chemin(lt);
  cond_r:=fin_de_chemin(rt);
  IF cond_r AND (NOT cond_l OR bas(xlt, ylt, lt) ≤ bas(xrt, yrt, rt)) THEN
    lien(rt):=lt;
    x_lien(rt):=xlt - xrt - d;
    y_lien(rt):=ylt - yrt;
  END_IF;
  IF cond_l AND (NOT cond_r OR bas(xrt, yrt, rt) ≤ bas(xlt, ylt, lt)) THEN
    lien(lt):=rt;
    x_lien(lt):=xrt - xlt + d;
    y_lien(lt):=yrt - ylt;
  END_IF;
END_PROCEDURE;

```

3.3.2.5 Position relative des sous-arbres

Nous cherchons maintenant à placer les nœuds relativement à leur père. Le placement se fait en deux étapes. Nous cherchons tout d'abord le placement vertical relatif des nœuds, puis leur placement horizontal.

3.3.2.5.1 Placement vertical. Nous plaçons verticalement les nœuds suivant le critère i , en calculant pour chaque nœud n la position verticale relative de ses fils. Si δ_v représente la distance de séparation verticale entre un nœud et ses fils :

Algorithme (Placement vertical des sous-arbres)

```

FUNCTION placement_vertical( $n$ : nœud,  $f_1, \dots, f_s$  : nœud  $\times \dots \times$  nœud)
  FOR  $i$  FROM 1 TO  $s$  DO
     $y(f_i) := -\delta_v - \text{hauteur\_boite\_noeud}(f_i)/2$ ;
  END\_FOR;
END\_FUNCTION;

```

Remarque. Il est également possible de proposer plusieurs types de placements des sous-arbres à la place du critère i , sans changer le reste de l'algorithme de tracé. À chaque nœud n de l'arbre, nous associons la donnée $\text{placement}_v(n)$, qui peut prendre trois valeurs, *haut*, *bas* ou *centre*, qui correspondent aux placements de la figure 3.7. Pour chaque fils f_i de n , on détermine alors la hauteur h_i du tracé du sous-arbre de racine f_i . On note H la hauteur maximale des tracés des sous-arbres. On place ensuite chaque nœud f_i de la façon suivante :

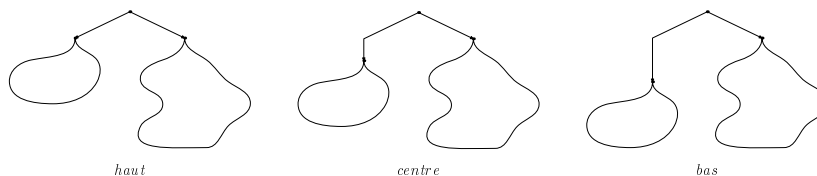


Fig. 3.7: Placement vertical des sous-arbres

```

FUNCTION placement_vertical( $n$ : nœud,  $f_1, \dots, f_s$  : nœud  $\times \dots \times$  nœud)
   $H := \max_{i=1, \dots, s} (\text{hauteur\_boite\_sous\_arbre}(f_i))$ ;
  FOR  $i$  FROM 1 TO  $s$  DO
    IF  $\text{placement}(f_i) = \text{haut}$  THEN
       $y(f_i) := -\delta_v - \text{hauteur\_boite\_noeud}(f_i)/2$ ;
    ELSE\_IF  $\text{placement}(f_i) = \text{centre}$  THEN
       $y(f_i) := -\delta_v - \text{hauteur\_boite\_noeud}(f_i)/2 - (H - \text{hauteur\_boite\_sous\_arbre}(f_i))/2$ ;
    ELSE
       $y(f_i) := -\delta_v - \text{hauteur\_boite\_noeud}(f_i)/2 - (H - \text{hauteur\_boite\_sous\_arbre}(f_i))$ ;
    END\_IF;
  END\_FOR;
END\_FUNCTION;

```

En pratique, dans le cas des placements *centre* et *bas*, nous ajoutons simplement un nœud temporaire comme racine des sous-arbres pour éviter les chevauchements entre nœuds et arêtes.

3.3.2.5.2 Placement horizontal. Nous cherchons maintenant à placer k sous-arbres, de racines s_1, \dots, s_k , à une distance δ_h les uns des autres, en respectant l'ordre sur les racines. Pour simplifier les notations, nous supposons que les racines des sous-arbres vérifient $s_1 \prec s_2 \prec \dots \prec s_k$. Pour respecter les propriétés de symétrie des sous-arbres, l'idée consiste à placer en premier les sous-arbres du « milieu », puis à ajouter deux à deux, à gauche et à droite, les sous-arbres restants en respectant l'ordre \prec . Ceci revient, dans le cas où k est pair, à placer d'abord les deux sous-arbres $s_{\frac{k}{2}}$ et $s_{\frac{k}{2}+1}$. Dans le cas où k est impair, nous plaçons simplement le sous-arbre $s_{\frac{k+1}{2}}$ au point d'abscisse nulle. Ensuite, si l'on suppose que les sous-arbres de racine s_i , pour $i \in [g, d]$, sont déjà placés, nous calculons la distance entre les sous-arbres s_{g-1} et s_g , notée $d_{g-1,g}$, puis celle entre s_d et s_{d+1} , notée $d_{d,d+1}$, et enfin celle entre s_{g-1} et s_{d+1} , notée $d_{g-1,d+1}$. Nous avons alors deux possibilités selon les distances trouvées, résumées sur les figures 3.8 et 3.9. Il reste ensuite à mettre en place les liens entre les sous-arbres, comme cela est illustré sur les figures 3.10, 3.11 et 3.12, en fonction des derniers nœuds atteints sur les contours lors du calcul de la distance horizontale entre les sous-arbres (et donc en fonction des tailles respectives des sous-arbres).

Algorithme (Placement horizontal des sous-arbres)

```

PROCEDURE placement_horizontal( $n$ : nœud,  $(s_1, \dots, s_k)$  : nœud  $\times \dots \times$  nœud)

```

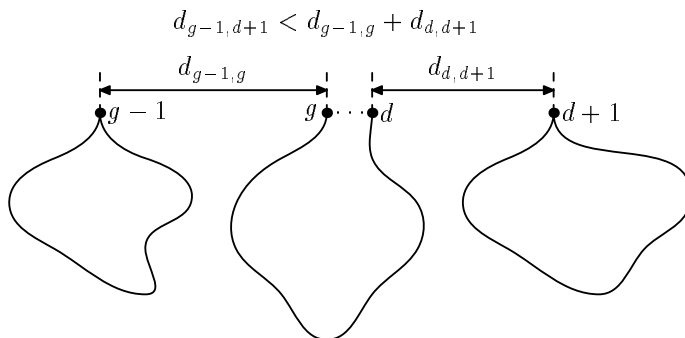


Fig. 3.8: Placement horizontal des sous-arbres : cas (1).

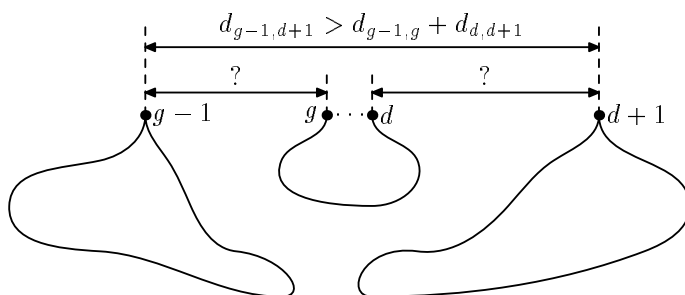


Fig. 3.9: Placement horizontal des sous-arbres : cas (2).

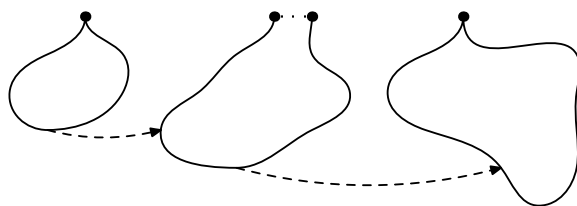


Fig. 3.10: Placement horizontal des sous-arbres : mise à jour des liens (1)

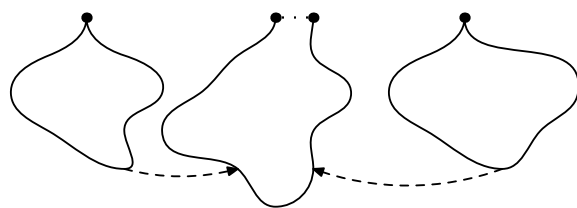


Fig. 3.11: Placement horizontal des sous-arbres : mise à jour des liens (2)

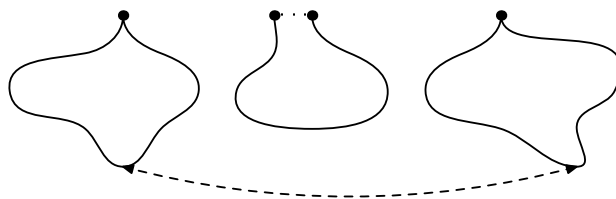


Fig. 3.12: Placement horizontal des sous-arbres : mise à jour des liens (3)

```

IF k = 1 THEN
  x(s1):=0;
ELSE
  IF k mod 2 = 0 THEN
    g:=k/2;
    d:=k/2+1;
    dgd:=distance_sous_arbres(sg,sd,ltgd,xltgd,yldgd,rtgd,xrtgd,yrtgd);
    mise_a_jour_des_liens(dgd,ltgd,xltgd,yldgd,rtgd,xrtgd,yrtgd);
    x(sd):=x(sd) + dgd;
  ELSE
    g:=(k+1)/2;
    d:=g;
    x(sg):=0;
  END_IF
  WHILE g ≥ 1 DO
    dg-1,d+1:=distance_sous_arbres(sg-1,sd+1,ltg-1,d+1,xltg-1,d+1,yldg-1,d+1,
      rtg-1,d+1,xrtg-1,d+1,yrtg-1,d+1);
    dg-1,g:=distance_sous_arbres(sg-1,sg,ltg-1,g,xltg-1,g,yldg-1,g,rtg-1,g,xrtg-1,g,yrtg-1,g);
    dd,d+1:=distance_sous_arbres(sd,sd+1,ltd,d+1,xltd,d+1,yldd,d+1,rtd,d+1,xrtd,d+1,yrtd,d+1);
    IF dg-1,d+1 > dg-1,g + dd,d+1 THEN
      mise_a_jour_des_liens(dg-1,d+1,ltg-1,d+1,xltg-1,d+1,yldg-1,d+1,
        rtg-1,d+1,xrtg-1,d+1,yrtg-1,d+1);
      r:=(dg-1,d+1 - dg-1,g - dd,d+1)/2;
      x(sg-1):=x(sg-1) - dg-1,g - r;
      x(sd+1):=x(sd+1) + dd,d+1 + r;
    ELSE
      x(sg-1):=x(sg-1) - dg-1,g;
      x(sd+1):=x(sd+1) + dd,d+1;
      yg-1:=bas(ltg-1,g,xltg-1,g,yldg-1,g);
      yg:=bas(rtg-1,g,xrtg-1,g,yrtg-1,g);
      yd:=bas(ltd,d+1,xltd,d+1,yldd,d+1);
      yd+1:=bas(rtd,d+1,xrtd,d+1,yrtd,d+1);
      IF yg-1 ≤ yg AND yg-1 ≤ yd THEN
        mise_a_jour_des_liens(dg-1,g + dd,d+1,ltg-1,d+1,xltg-1,d+1,yldg-1,d+1,
          rtg-1,d+1,xrtg-1,d+1,yrtg-1,d+1);
      ELSE
        mise_a_jour_des_liens(dg-1,g,ltg-1,g,xltg-1,g,yldg-1,g,rtg-1,g,xrtg-1,g,yrtg-1,g);
        mise_a_jour_des_liens(dd,d+1,ltd,d+1,xltd,d+1,yldd,d+1,rtd,d+1,xrtd,d+1,yrtd,d+1);
      END_IF
    END_IF
    g:=g - 1;
    d:=d + 1;
  END_WHILE
END_IF;
END_PROCEDURE;

```

3.3.2.6 Placement de la racine

Nous savons maintenant placer les nœuds des sous-arbres verticalement et horizontalement. Il ne reste plus qu'à placer la racine de l'arbre. Nous proposons un nouveau critère esthétique pour le placement de la racine :

- vi. la racine n d'un arbre doit être placée comme indiqué sur la figure 3.13. Le nœud racine n est centré vis-à-vis de ses fils s_1 et s_k , et séparé verticalement par au moins une distance δ_v . On demande de plus à ce que l'angle α entre les arêtes et l'horizontale soit supérieur à un angle minimum α_{min} .

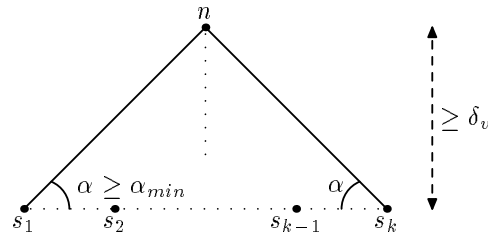
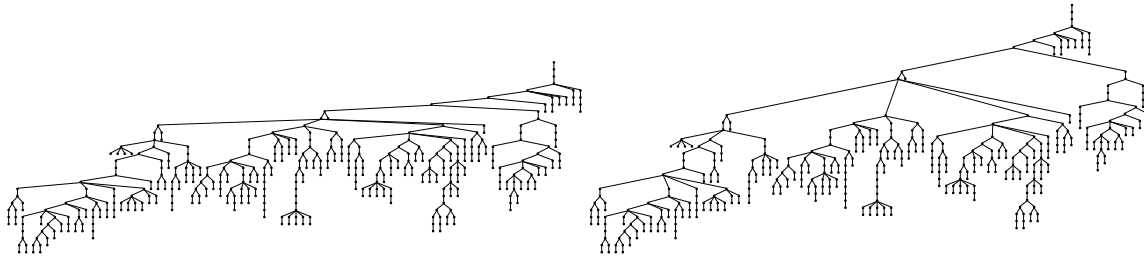


Fig. 3.13: Placement de la racine des sous-arbres.

Ce critère est introduit pour limiter les effets de confusions entre nœuds et arêtes qui apparaissent lors de l'affichage de grand graphes. La figure 3.14 montre le même arbre dessiné sans et avec le critère δ_v . Pour dessiner un arbre sans tenir compte du critère δ_v , il suffit de choisir $\alpha_{min} = 0$.

Fig. 3.14: Arbre général de taille 500 dessiné avec et sans le critère δ_v .

L'algorithme général permettant de calculer les positions relatives des nœuds d'un sous-arbre s'écrit alors :

Algorithme (Positions relatives des nœuds d'un sous-arbre)

```

PROCEDURE positions_relatives_sous_arbres (n:nœud)
  LET (s1, ..., sk) SUCH THAT n O→ si;
  FOR i FROM 1 TO s DO
    positions_relatives_sous_arbres (si);
  END_FOR;
  placement_vertical (n, (s1, ..., sk));
  placement_horizontal (n, (s1, ..., sk));
  mx := (x(s1) + x(sk))/2;
  my := max(0, tan(alpha_min) * (x(sk) - x(s1))/2 - delta_v);
  FOR i FROM 1 TO s DO
    x(si) := x(si) - mx;
    y(si) := y(si) - my;
  END_FOR
  x(n) := 0;
  y(n) := 0;
END_PROCEDURE;

```

3.3.2.7 Position absolue

Une fois connue la position des nœuds des sous-arbres relativement à leur père, on peut facilement en déduire leur position absolue :

Algorithme (Positions absolues des nœuds d'un sous-arbre)

- Données : un arbre de racine n dont les positions relatives des nœuds sont connues,

– *Résultat* : un arbre dont les positions absolues sont connues.

```

PROCEDURE positions_absolues_sous_arbres (n : nœud)
  LET (s1, ..., sk) SUCH THAT n O→ si;
  FOR i FROM 1 TO k DO
    x(si) := x(si) + x(n);
    y(si) := y(si) + y(n);
    positions_absolues_sous_arbres (si);
  END_FOR;
END_PROCEDURE;

```

3.3.2.8 Complexité

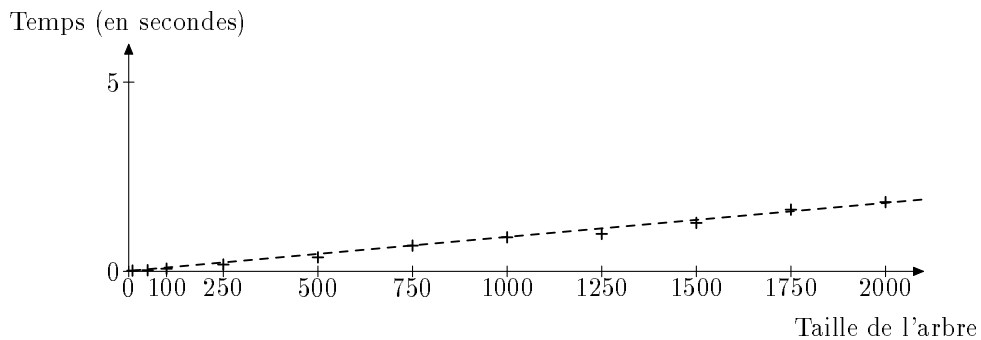


Fig. 3.15: Temps de calcul des positions absolues des nœuds d'arbres généraux.

Dans le cas d'arbres dont tous les nœuds ont la même taille et pour lesquels l'angle minimal α_{min} entre les arêtes et l'horizontale est nul, la complexité de l'algorithme reste linéaire. En effet, les nœuds sont alors placés par niveaux, et à chaque fusion de sous-arbres, les liens ajoutés garantissent que lors du parcours du contour de l'arbre formé, on change de niveau après au plus deux appels à la fonction `chemin_suivant_gauche` ou `chemin_suivant_droit`. Le résultat dans le cas k -aire s'obtient ensuite de la même façon que dans le cas binaire [44].

On peut vérifier expérimentalement que ce résultat reste vrai en pratique pour des arbres ayant des nœuds de taille différente et un placement au plus près. Nous indiquons sur les figures 3.15 et 3.16 les temps de calcul des positions absolues des nœuds pour des arbres généraux de différentes tailles, sans prendre en compte les temps de lecture et d'affichage. Ces arbres sont générés aléatoirement, et on affecte aux nœuds de l'arbre des largeurs comprises entre $\delta_h/10$ et $10 \times \delta_h$ et des hauteurs comprises entre $\delta_v/10$ et $10\delta_v$.

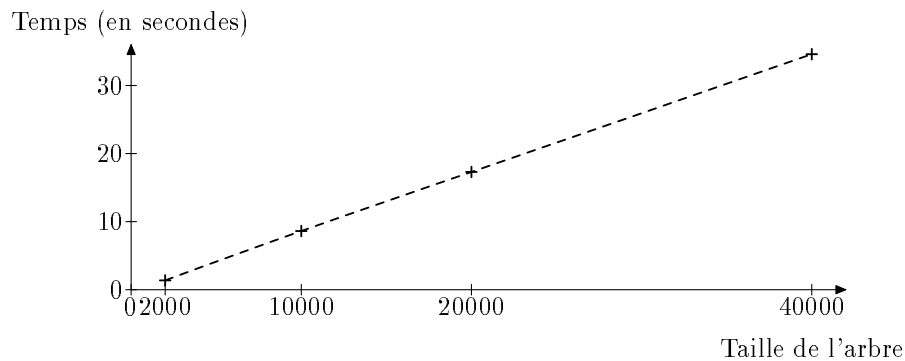


Fig. 3.16: Temps de calcul des positions absolues des nœuds d'arbres binaires.

3.3.3 Exemples

Nous illustrons l'algorithme de tracé d'arbres enracinés par quelques exemples. Le premier concerne la représentation de *familles aliquotes* sous la forme d'arbres k -aires ayant des nœuds ponctuels. Le second exemple montre l'utilisation de l'algorithme dans le cas d'arbres ayant des nœuds de tailles différentes. Enfin, le dernier exemple est une représentation d'une structure combinatoire particulière, les arbres binaires récursifs.

3.3.3.1 Suites aliquotes

Définition 3.3.2 (Suite aliquote [6])

La *suite aliquote* d'un entier n est la suite définie par :

$$\begin{cases} u_0 = n \\ u_{j+1} = f(u_j) = \sigma(u_j) - u_j \end{cases}$$

où $\sigma(n)$ est la somme des diviseurs de n . Un nombre n pour lequel $f(n) = n$ est un *nombre parfait*. .

Exemple. La suite aliquote de 12 est 12, 16, 15, 9, 4, 3, 1, 0, 0, Le nombre 6 est parfait. .

Définition 3.3.3 (Famille aliquote [10])

La *famille aliquote* d'un entier n est l'ensemble des nombres dont la suite aliquote contient l'élément n . .

Nous représentons graphiquement la famille aliquote d'un nombre n à l'aide de notre algorithme de tracé d'arbres enracinés, en se limitant aux suites aliquotes des nombres inférieurs à 300 qui ne possèdent pas de terme supérieur à 100000. Pour cela nous construisons tout d'abord un graphe fonctionnel, dont les nœuds sont les entiers de 1 à 100000. Deux nœuds du graphe, représentant les entiers a et b sont reliés par une arête orientée $b \rightarrow a$ si $a = f(b)$. Ensuite, nous sélectionnons la composante connexe du graphe contenant l'entier n dont on cherche la famille. La famille de l'entier 37, représentée sous forme d'arbre enraciné, est représentée sur la figure 3.17. On remarquera sur cette figure que les nœuds 37 et 196 sont placés suivant le critère vi, ce qui conduit à un graphe moins large que dans le cas d'un placement par niveaux.

3.3.3.2 Nœuds de tailles différentes

Pour illustrer l'algorithme de tracé avec des nœuds de différentes tailles, nous nous intéressons dans cet exemple au tracé d'arbres généraux générés aléatoirement, en affectant des largeurs comprises entre $\delta_h/10$ et $3\delta_h$ et des hauteurs comprises entre $\delta_v/10$ et $3\delta_v$. Cet exemple permet en outre de tester l'algorithme. On peut voir le résultat d'une génération de taille 150 sur la figure 3.18.

3.3.3.3 Arbres binaires récursifs

Nous considérons dans cet exemple la spécification de structures décomposables suivante :

$$\begin{cases} T \rightarrow Union(A, B) \\ B \rightarrow Prod(T, T, T) \\ A \rightarrow \mathbf{a} \\ \mathbf{a} \rightsquigarrow Atom \end{cases} \quad (3.1)$$

Nous générons ensuite une structure aléatoire de taille 49, que nous représentons sur la figure 3.19. Les composantes de la structure générée, construites à l'aide du constructeur *prod*, sont dessinées comme des arbres binaires dont la première composante du produit est incluse dans un nœud racine, les deux autres formant les sous-arbres de cette racine.

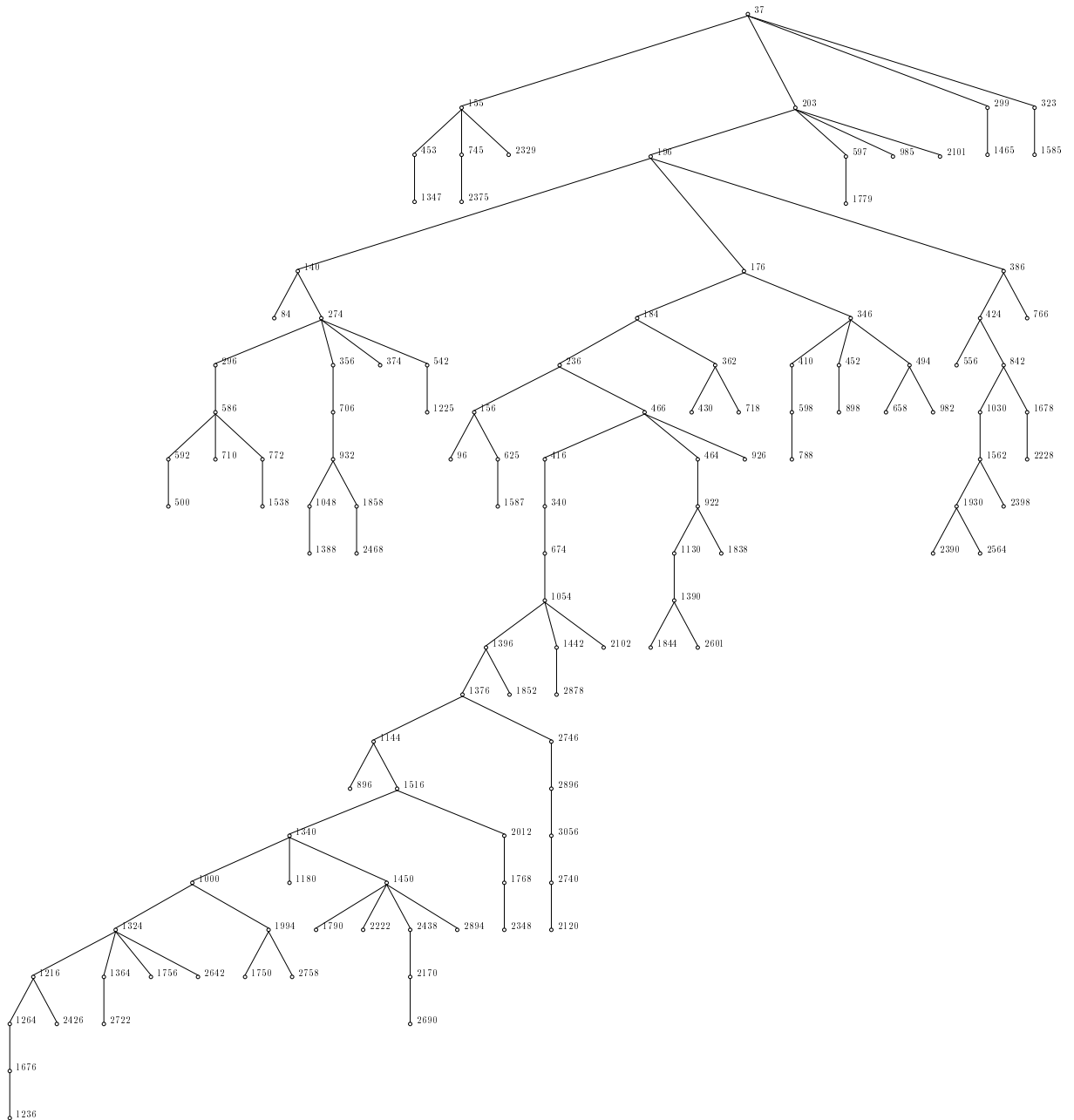


Fig. 3.17: La famille aliquote de l'entier 37.

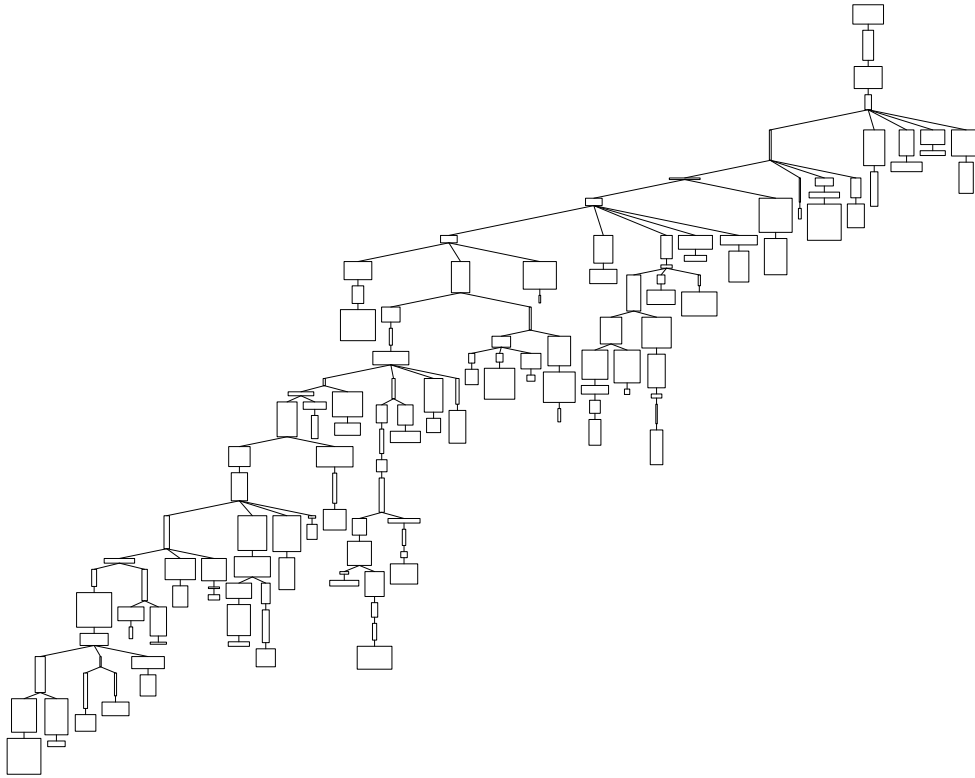


Fig. 3.18: Arbre général de taille 150 dont les nœuds ont des tailles différentes.

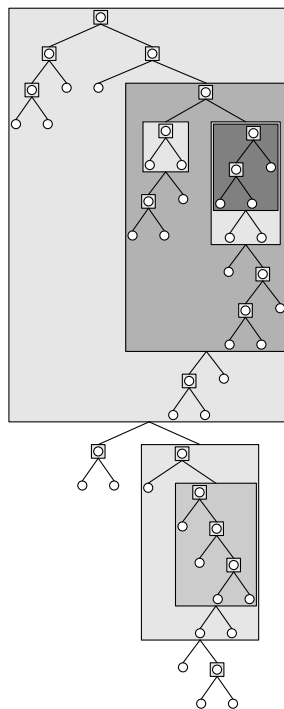


Fig. 3.19: Arbre binaire récursif représentant une structure de taille 49 issue de la spécification (3.1).

3.4 Arbres enracinés en « cascade »

Nous présentons un algorithme basé sur les conventions de tracé définies par P. Eades, T. Lin et X. Lin dans [19]. Un tracé en cascade d'un arbre enraciné consiste à placer les fils d'un nœud de haut en bas, contrairement à l'algorithme précédent où les fils étaient placés de gauche à droite. La figure 3.20 montre le même arbre enraciné dessiné avec les deux conventions de tracé. Comme cela est illustré sur cette figure, les tracés en cascade conduisent à des arbres moins larges mais plus hauts qu'avec la convention classique. Le principe de l'algorithme est exactement le même que dans la convention de tracé classique. La figure 3.21 ci-dessous illustre les différentes étapes, que l'on peut comparer avec la figure 3.4. Les flèches en pointillés courts correspondent aux liens permettant de suivre les contours haut et bas des sous-arbres, la flèche en pointillés longs représente le lien ajouté entre les deux sous-arbres une fois déterminée la distance verticale entre les sous-arbres. L'algorithme est plus simple que dans le cas de la convention classique, et nous ne le détaillons pas.

3.5 Graphes non orientés

Nous considérons maintenant le tracé de graphes non orientés. Une première méthode consiste à associer aux représentations des graphes une *fonction énergie* J d'autant plus basse que la qualité du dessin est grande. Pour un graphe donné, on cherche alors à minimiser la fonction J par des méthodes classiques d'optimisation. Cette méthode est utilisée initialement par R. Davidson et D. Harel [11], en utilisant une méthode d'optimisation par recuit simulé. L'avantage de cette méthode est que l'on peut associer des fonction d'estimation du dessin quelconques, qui peuvent traduire des critères esthétiques très variés. En revanche, le coût de la méthode est élevé et celle-ci nécessite de déterminer de nombreux paramètres, différents pour chaque graphe, pour obtenir de bons résultats. Une méthode proposée antérieurement par P. Eades [15], appelée *spring embedder* peut être vue comme un cas particulier de la méthode décrite ci-dessus, mais pour laquelle il est possible d'obtenir rapidement une solution satisfaisante sans réglage de paramètres. Nous présentons maintenant deux algorithmes issus de cette dernière approche.

3.5.1 Algorithme FR

Le principe de l'algorithme *spring embedder* proposé par P. Eades consiste à ramener le problème de tracé d'un graphe à celui de la simulation d'un système physique constitué d'anneaux et de ressorts. Les nœuds du graphe sont les anneaux, reliés deux à deux par des ressorts. Les longueurs au repos des ressorts sont déterminées par la présence ou non d'une arête entre les nœuds extrémités. Cette méthode est similaire à celle de R. Davidson et D. Harel dans le cas où la fonction d'énergie J ne prend en compte que les critères d'éloignement des nœuds et de longueur uniforme des arêtes, mais a l'avantage de converger plus rapidement sans aucun réglage de paramètres. Un avantage supplémentaire de cet algorithme est qu'il est simple à mettre en œuvre et qu'il semble faire ressortir naturellement les symétries du graphe.

Plusieurs heuristiques pour le tracé d'arbres généraux s'inspirent de l'algorithme *spring embedder*. On peut citer l'algorithme de T. Kamada et S. Kawai [37] qui prend en compte des informations supplémentaires concernant la longueur minimale des chemins entre les nœuds du graphe. Nous choisissons de présenter ici l'algorithme **FR** proposé par T. Fruchterman et E. Reingold [27], qui utilise des principes de répulsion entre tous les couples de nœuds et d'attraction entre les nœuds reliés par des arêtes. Nous étendons l'algorithme initial pour tenir compte des nœuds de taille arbitrairement grande.

3.5.2 Principe

Si $d_{opt}(a, b)$ représente la distance idéale entre les centres des nœuds a et b reliés par une arête, alors on peut définir les forces d'attractions qui portent sur a et b comme suit :

Algorithme (Attraction des nœuds)

```
PROCEDURE attraction(a:nœud, b:nœud)
  d:=distance(a,b);
  f:= $\frac{d^2}{d_{opt}(a,b)}$ ;
```

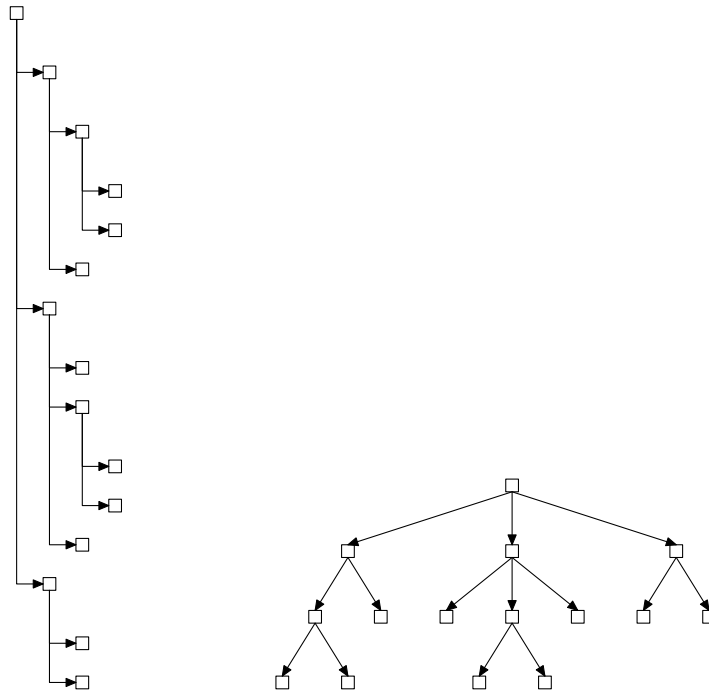


Fig. 3.20: Arbre enraciné suivant les conventions en « cascade » et classique.

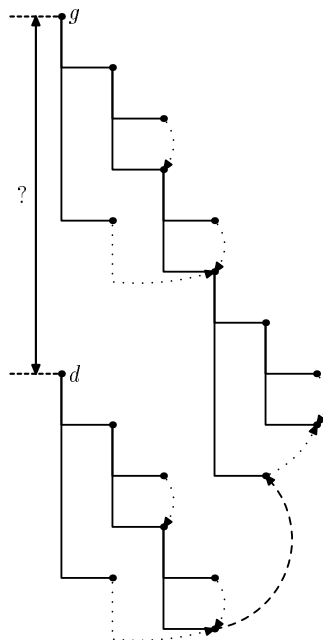


Fig. 3.21: Principe de l'algorithme de tracé en « cascade ».

```

force(a):=force(a)+f  $\frac{\vec{ab}}{d}$ ;
force(b):=force(b)-f  $\frac{\vec{ab}}{d}$ ;
END_PROCEDURE;

```

De même pour chaque couple de nœuds du graphe, on définit les forces de répulsion entre les nœuds. On choisit de ne considérer que les couples de nœuds situés à une distance inférieure à un paramètre d_{max} , ce qui permet entre autres de traiter les graphes non connexes.

Algorithme (Répulsion des nœuds)

```

PROCEDURE repulsion(a:nœud, b:nœud)
d:=distance(a,b);
IF  $d \leq d_{max}$  THEN
f:=- $\frac{d_{opt}(a,b)^2}{d}$ ;
force(a):=force(a)+f  $\frac{\vec{ab}}{d}$ ;
force(b):=force(b)-f  $\frac{\vec{ab}}{d}$ ;
END_IF;
END_PROCEDURE;

```

La valeur de d_{opt} peut être calculée en fonction des tracés des nœuds et de la longueur souhaitée pour les arêtes. Si l'on note δ_e la longueur souhaitée pour une arête, on détermine la valeur de d_{opt} en fonction des positions et des formes des nœuds comme indiqué sur la figure 3.22.

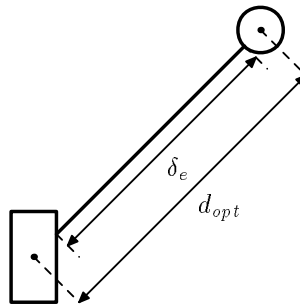


Fig. 3.22: Calcul de la distance souhaitable d_{opt} entre les centres de deux nœuds en fonction de la longueur souhaitée δ_e de l'arête les reliant.

Une étape de l'algorithme consiste à calculer les forces de répulsion pour chaque couple de nœuds et les forces d'attraction pour chaque couple de nœuds reliés par une arête. Ensuite, on déplace les nœuds en fonction des forces qui leur sont appliquées, en limitant les déplacements à l'intérieur d'une boule centrée sur le nœud et de rayon *borne_deplacement*.

Algorithme (Étape du calcul)

```

PROCEDURE etape(borne_deplacement:INTEGER)
FOR ALL nœud a DO
force(a):= $\vec{0}$ ;
END_FOR;
FOR ALL nœud a DO
FOR ALL nœud b DO
repulsion(a,b);
END_FOR;
END_FOR;
FOR ALL arete e OF KIND a-o-b DO
attraction(a,b);
END_FOR;
FOR ALL nœud a DO

```



```

IF norme(force(a)) > borne_deplacement THEN
  force(a) :=  $\frac{\text{force}(a) \cdot \text{borne\_deplacement}}{\text{norme}(\text{force}(a))}$ ;
END_IF;
position(a) := position(a) + force(a);
END_FOR;
END_PROCEDURE;

```

On appelle ensuite un certain nombre de fois la procédure `etape` en faisant décroître la valeur de `borne_deplacement`.

Remarque. Il est également possible de choisir d'arrêter les itérations dès qu'une position satisfaisante est atteinte. Pour cela, on peut par exemple tester si les nœuds oscillent autour d'une position d'équilibre. .

La figure 3.23 est un exemple d'application de l'algorithme pour des nœuds de tailles différentes.

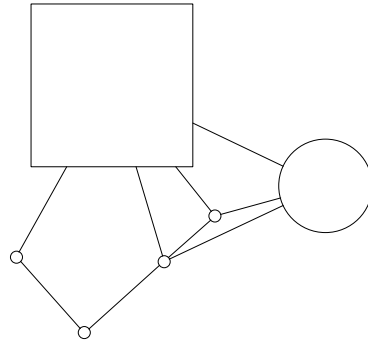


Fig. 3.23: Exemple d'application de l'algorithme `FR` pour un graphe ayant des nœuds de tailles différentes.

Remarque. L'algorithme `FR` s'applique dans des espaces euclidiens de dimension quelconque. .

3.5.2.1 Complexité

Si l'on considère que le coût de calcul est déterminé par le coût de calcul des forces d'attraction et de répulsion, ainsi que par les coûts d'initialisation et de déplacement des nœuds, le coût de l'algorithme est :

$$O(km) + O(kn^2),$$

où k correspond au nombre d'étapes de l'algorithme, n au nombre de nœuds et m au nombre d'arêtes du graphe.

Pour améliorer l'ordre de complexité, T. Fruchterman et E. Reingold proposent d'utiliser des techniques inspirées de celles utilisées dans le problème des n corps. Étant donné que l'on ne considère plus les interactions entre les couples de nœuds situés à une distance supérieure à d_{max} , l'idée consiste à séparer l'espace comprenant les nœuds en une grille carrée, et à ne considérer les interactions qu'entre carrés voisins. Pour cela, les carrés sont choisis avec une largeur et une hauteur égales à d_{max} . Quand la répartition des nœuds est uniforme et le nombre de carrés de l'ordre de n , le coût du calcul des forces de répulsion des nœuds est alors en $O(n)$.

3.5.2.2 Exemples

3.5.2.2.1 Appels téléphoniques. Le graphe de la figure 3.24 correspond à la représentation d'un graphe issu d'une base de données collectant des appels téléphoniques. Les nœuds sont étiquetés par les numéros de téléphone, et la couleur des nœuds est différente selon les indicatifs téléphonique correspondants. Les arêtes traduisent l'information selon laquelle deux personnes sont entrées en contact. Le tracé du graphe a été réalisé à l'aide de l'algorithme `FR` que nous avons présenté, suivi de quelques déplacements de nœuds effectués manuellement pour rendre le dessin plus régulier [20].

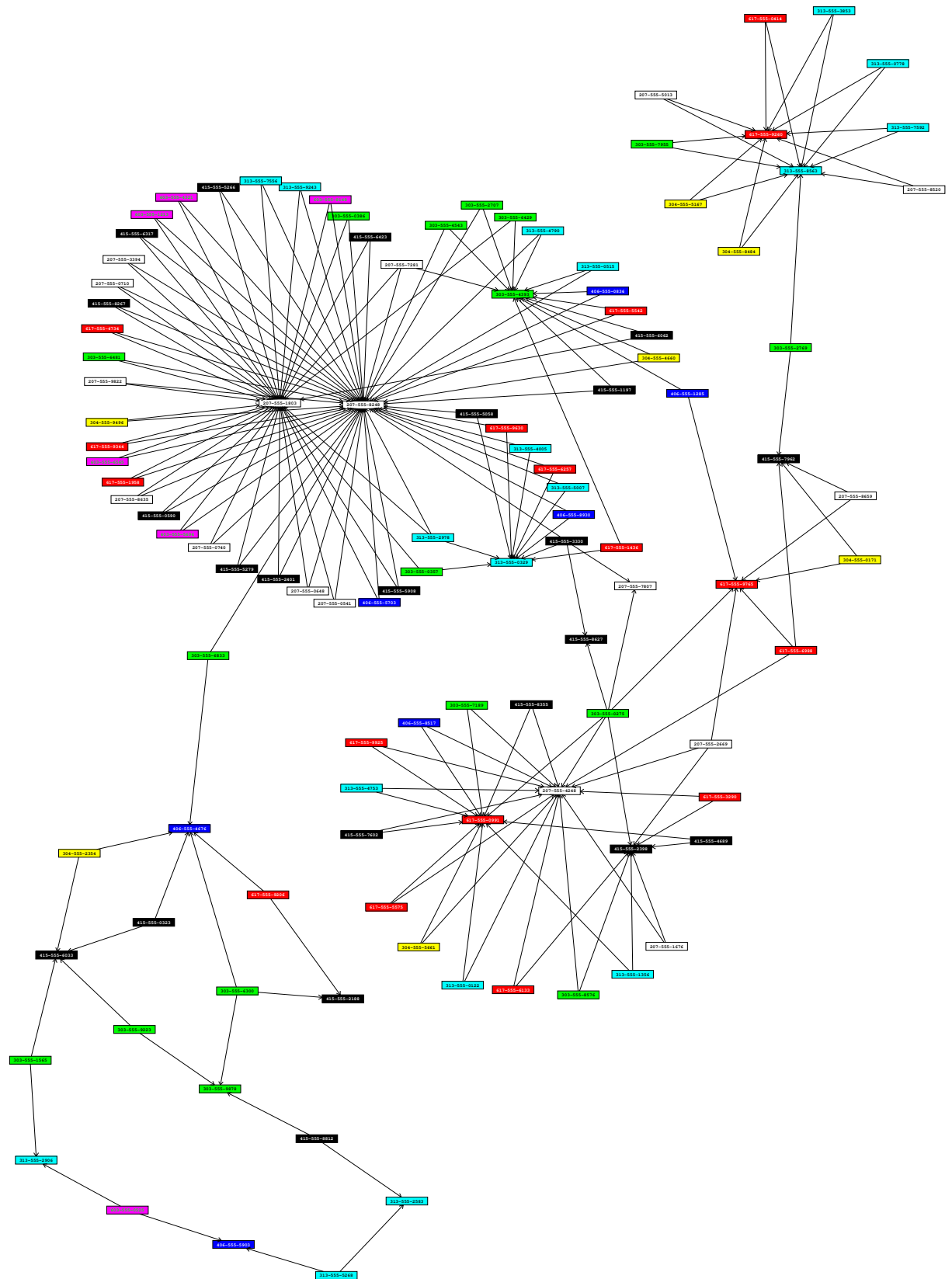


Fig. 3.24: Grphe B.

3.5.2.2.2 Groupes de permutations et graphes de Cayley.

Définition 3.5.1 (Groupe)

Un groupe est un couple (G, \cdot) , où G est un ensemble non vide et \cdot une application de $G \times G$ dans G , associative, possédant un élément neutre et telle que tout élément g du groupe admet un inverse g^{-1} . L'application \cdot est appelée l'*opération* du groupe. Le cardinal $|G|$ de l'ensemble G est appelé l'*ordre* du groupe. .

Définition 3.5.2 (générateurs)

Un ensemble $\mathcal{G} \subseteq G$ engendre le groupe G si chaque élément du groupe g peut être exprimé comme un produit d'éléments de \mathcal{G} ou de leurs inverses. L'ensemble \mathcal{G} est appelé un ensemble de *générateurs*, et ses éléments sont des générateurs du groupe G . .

Définition 3.5.3 (Graphe de Cayley)

Soit (G, \cdot) un groupe et \mathcal{G} un ensemble de générateurs de G . Le *graphe de Cayley* $Cay(G, \mathcal{G})$ est le graphe orienté défini par :

- l'ensemble des nœuds de $Cay(G, \mathcal{G})$ est en bijection avec les éléments du groupe G ,
 - il existe une arête orientée entre les nœuds représentant les éléments a et b du groupe G , étiquetée par un générateur g de \mathcal{G} , si et seulement si $a \cdot g = b$.
- .

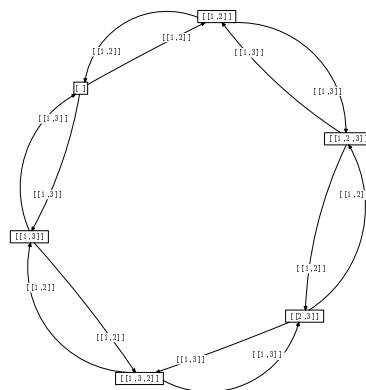


Fig. 3.25: Graphe de Cayley du groupe de permutations de générateurs $((1, 2)$ et $((1, 3))$.

Nous nous intéressons à la représentation des graphes de Cayley pour les groupes de permutations. Dans l'exemple suivant, nous définissons la variable **gen** qui correspond aux générateurs du groupe. Le programme **Maple Cayley** (écrit pour permettre de traiter cet exemple) permet ensuite de construire le graphe de Cayley du groupe engendré, en prenant comme loi de groupe la composition des permutations (fonction **mulperms**). La variable **m** correspond à l'ensemble des éléments du groupe. La description **g** du graphe comprend l'ensemble des générateurs, l'ensemble des éléments du groupe et l'ensemble des relations entre éléments. Ensuite, nous convertissons cette description à l'aide de la fonction **Cayley2cgraph** pour permettre de dessiner le graphe à l'aide du programme **CGraph**. Le tracé obtenu est indiqué sur la figure 3.25 :

```
> read('Cayley.mp');
> gen:=[[1,2]], [[1,3]]:
> m:=permgrouppermembers(gen):
> g:=Cayley(m,
    gen,
    mulperms):

g:=[[[[1, 2]], [[1, 3]]], # generateurs
```

```

[[], [[1, 2]], [[1, 3]], [[1, 2, 3]], [[1, 3, 2]], [[2, 3]]], # noeuds
[[1, 2, 1], [1, 3, 2], [2, 1, 1], [2, 4, 2], [3, 1, 2], [3, 5, 1], # aretes
 [4, 2, 2], [4, 6, 1], [5, 3, 1], [5, 6, 2], [6, 4, 1], [6, 5, 2]]]

```

```
> Cayley2cgraph_label(g, 'Cayley.g'):
```

Les figures 3.26 et 3.27 montrent l'application du même algorithme pour un graphe plus complexe. Cet exemple illustre les limitations de l'algorithme dès que le nombre d'arêtes devient important. On peut comparer les résultats obtenus avec ceux des figures 3.28 et 3.29, qui utilisent un algorithme avec conservation des propriétés de croisements que nous décrivons par la suite. Cependant, dans cette dernière représentation, le placement initial des nœuds doit être guidé par l'utilisateur.

3.5.3 Algorithme FRCC

Nous présentons maintenant un nouvel algorithme, appelé **FRCC**, variante de l'algorithme **FR**, qui permet de conserver les propriétés de croisements entre arêtes. Cet algorithme construit, à partir d'une représentation d'un graphe, une nouvelle représentation, telle que si deux arêtes du graphe se croisent sur la représentation initiale (respectivement ne se croisent pas), elle se croisent également sur la représentation finale (respectivement ne se croisent pas).

Le principe de l'algorithme proposé est simple, et consiste à assurer que les déplacements des nœuds, lors de chaque étape de l'algorithme, conservent les propriétés de croisements des arêtes. À chaque nœud a du graphe, nous associons l'ensemble $E(a)$ qui correspond à l'ensemble des positions que le nœud a est autorisé à prendre lors de son déplacement. Au début de chaque étape, $E(a)$ correspond à la boule centrée en a et de rayon le déplacement maximal autorisé, donné par la valeur *borne_deplacement*. Ensuite, pour chaque nœud a et chaque arête e dont les extrémités sont les nœuds b et c , nous calculons la distance euclidienne d entre a et e , et nous mettons à jour les ensembles $E(a)$, $E(b)$ et $E(c)$ comme indiqué sur la figure 3.30. $E(a)$, figuré en grisé sur la figure, devient l'intersection de $E(a)$ et du demi plan contenant a et de frontière la droite parallèle à e et à distance $d/3$ de a . $E(b)$ et $E(c)$ sont construits de façon similaire, en effectuant leur intersection avec le demi-plan ne contenant pas a dont la frontière est parallèle à e et située à une distance $2/3d$ de a . Ceci garantit que les propriétés de croisement des arêtes sont conservées. En effet les nœuds ne peuvent pas « traverser » les arêtes au cours des différentes étapes.

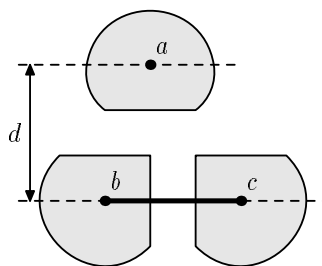


Fig. 3.30: Mise à jour des déplacements autorisés.

Par contre, les nœuds tendent alors à se rapprocher des arêtes. Pour éviter cela, nous introduisons une nouvelle interaction entre les nœuds et les arêtes. Pour un nœud a et une arête e d'extrémités b et c , nous déterminons le point i , s'il existe, qui correspond à la projection du point a sur l'arête. Nous calculons alors la force de répulsion entre le nœud a et un nœud fictif situé au point i , et nous transmettons ensuite les forces calculées aux points extrémités de l'arête (Fig. 3.31). La force de répulsion appliquée entre les nœuds et les arêtes peut être ajustée en fonction d'un paramètre d_e .

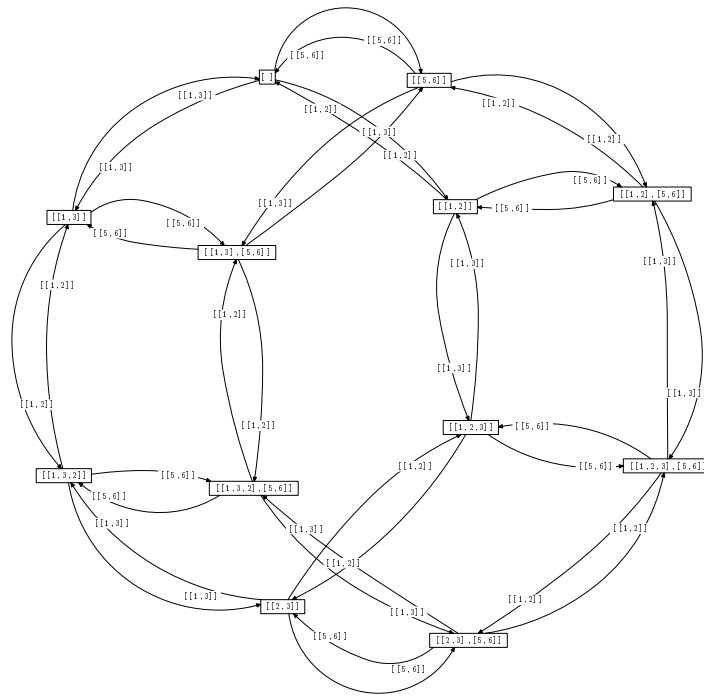


Fig. 3.26: Graphe de Cayley du groupe de permutations de générateurs $((1,2))$, $((1,3))$ et $((5,6))$ (algorithme FR).

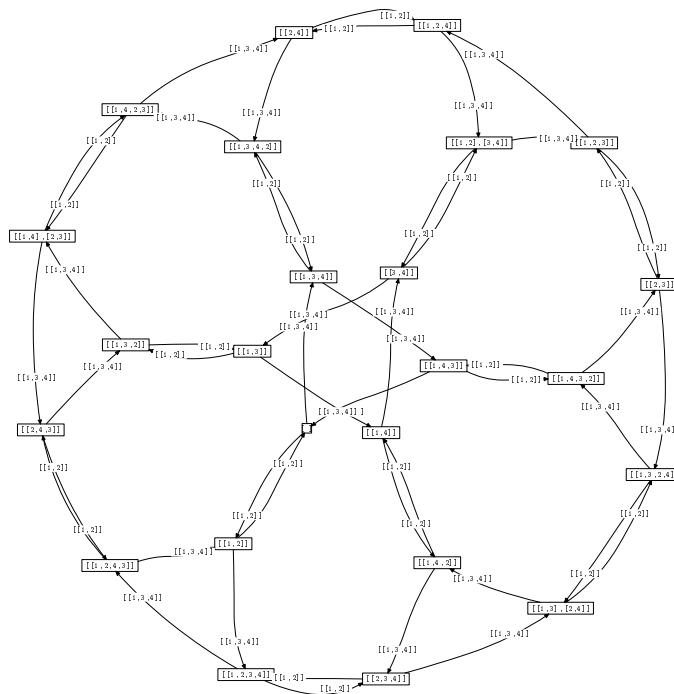


Fig. 3.27: Graphe de Cayley du groupe de permutations de générateurs $((1,2))$ et $((1,3,4))$ (algorithme FR).

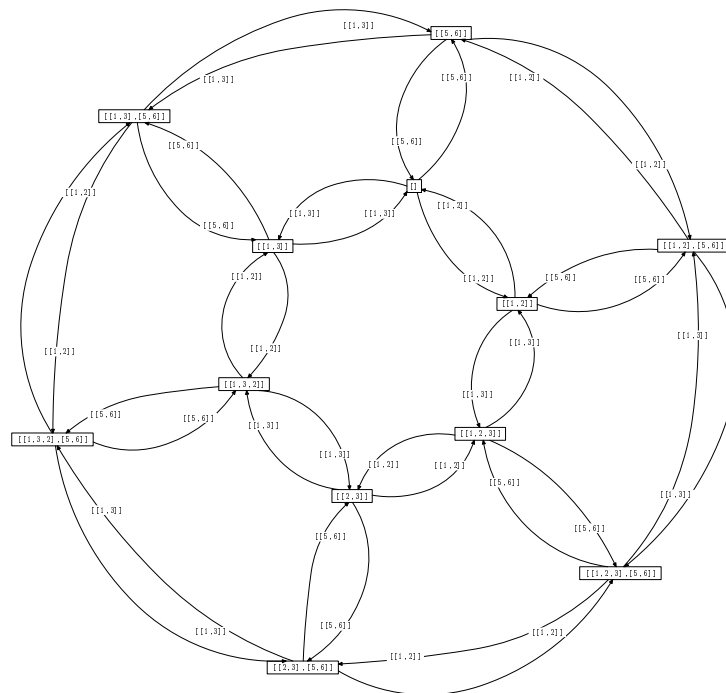


Fig. 3.28: Graphe de Cayley du groupe de permutations de générateurs $((1,2))$, $((1,3))$ et $((5,6))$ (algorithme FRCC).

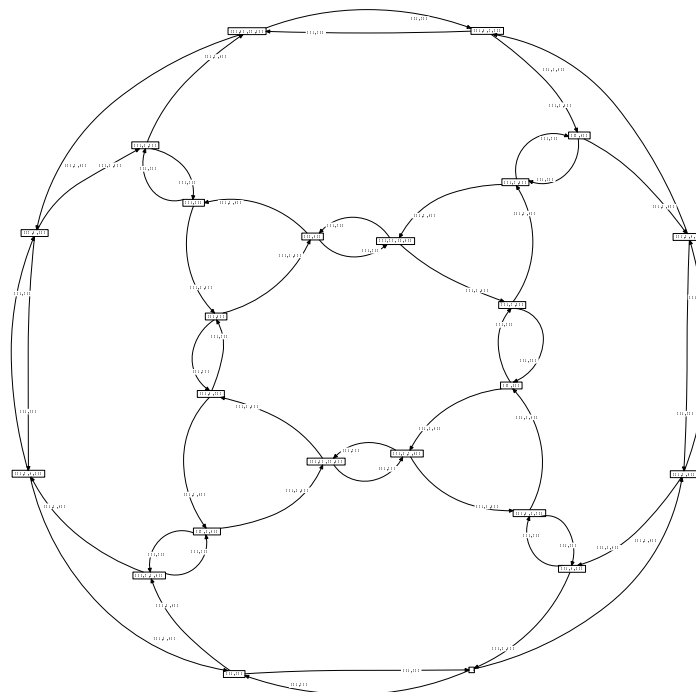


Fig. 3.29: Graphe de Cayley du groupe de permutations de générateurs $((1,2))$ et $((1,3,4))$ (algorithme FRCC).

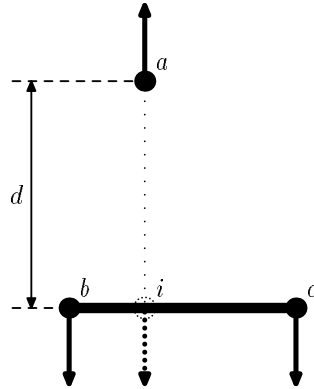


Fig. 3.31: Calcul des interactions entre nœuds et arêtes.

Algorithme (Répulsion entre nœuds et arêtes)

```

PROCEDURE repulsion(a:nœud, e:arête)
  IF  $\exists i \in e$  SUCH THAT  $\vec{ai} \cdot \vec{bc} = 0$  THEN
    d:=distance(a,i);
    IF  $d < d_{max}$  THEN
       $f := -\frac{d_{e^2}}{d}$ ;
      force(a):=force(a)+f  $\frac{\vec{ai}}{d}$ ;
      force(b):=force(b)-f  $\frac{\vec{ai}}{d}$ ;
      force(c):=force(c)-f  $\frac{\vec{ai}}{d}$ ;
    END_IF;
  END_IF;
END_PROCEDURE;

```

La figure 3.32 montre quelques exemples d'application de l'algorithme FRCC. Les exemples (1) et (2) montrent l'utilisation du programme pour améliorer les représentations obtenues à partir des algorithmes classiques de tracé de graphes planaires. L'algorithme proposé permet ainsi de faire apparaître les symétries dans les graphes planaires. Les exemples (3), (4) et (5) illustrent les propriétés de conservation de la partition de l'espace induite par les arêtes, et montrent l'intérêt de l'algorithme dans le cas d'une utilisation interactive : l'utilisateur choisit une position de départ, qui peut provenir éventuellement d'une modification du tracé obtenu après application de l'algorithme FR, puis applique l'algorithme FRCC pour affiner les placements des nœuds.

3.6 Graphes orientés

Nous considérons maintenant le tracé de graphes orientés. Nous présentons deux algorithmes différents. Le premier utilise les principes d'attraction répulsion des algorithmes à ressorts. L'adaptation au cas des graphes orientés diffère cependant assez nettement du cas non orienté, le tracé du graphe obtenu étant différent selon l'ordre dans lequel les arêtes du graphe sont considérées. Le second algorithme utilise les techniques courantes pour dessiner des graphes orientés, selon les principes généraux proposés par K. Sugiyama, S. Tagawa et M. Toda [52].

3.6.1 Algorithme à « ressorts »

Nous proposons d'appliquer les principes mis en œuvre dans le cas des graphes non-orientés au cas des graphes orientés. Nous cherchons à vérifier les critères esthétiques suivants :

- i. les arêtes doivent être dirigées de haut en bas,

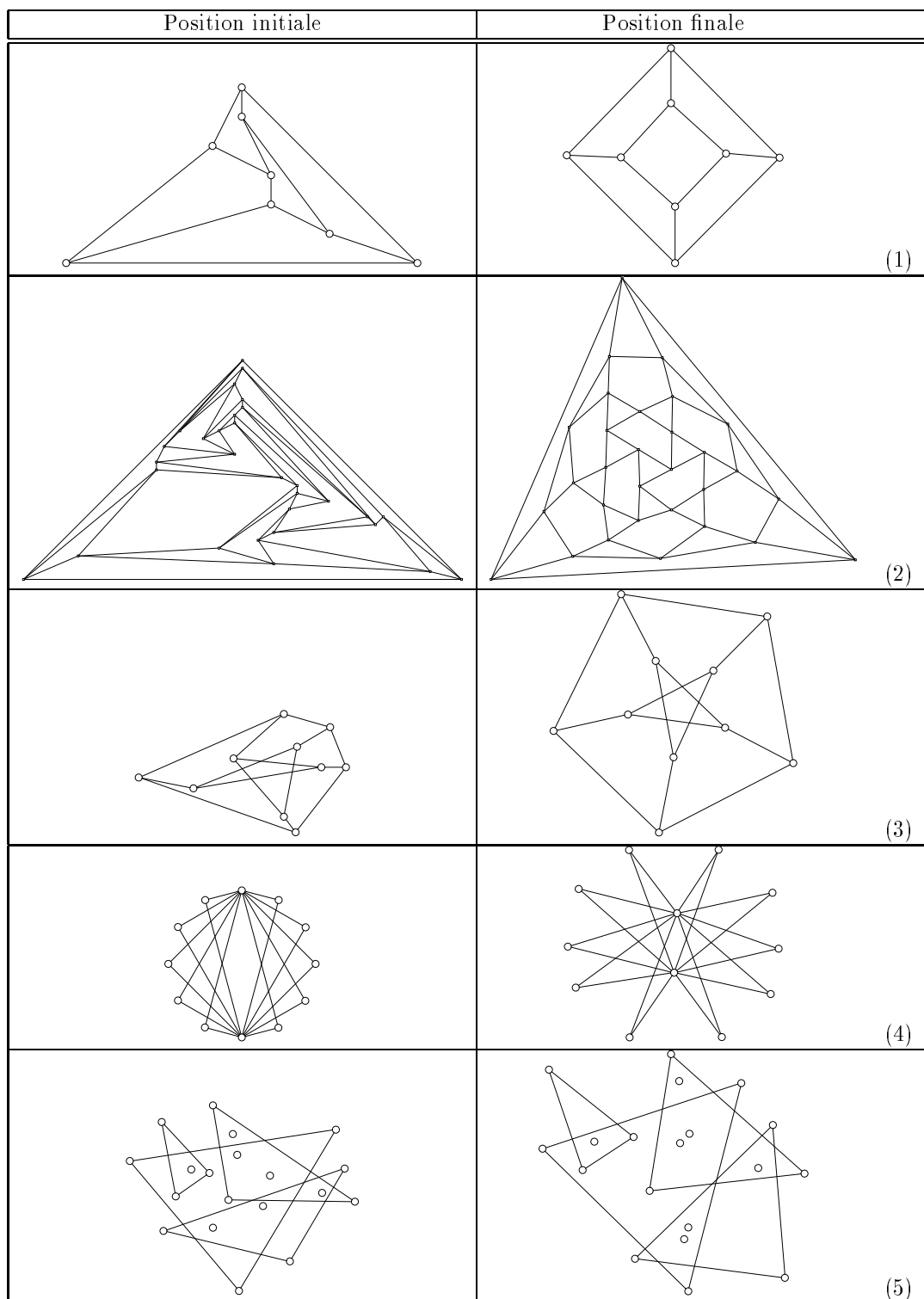


Fig. 3.32: Exemples de tracés obtenus avec l'algorithme FRCC.

- ii. les nœuds doivent être distribués uniformément sur la page,
- iii. les arêtes doivent être de longueur uniforme.

Nous conservons les mêmes notations que dans le cas non-orienté. Le vecteur position d'un nœud a est noté $position(a)$, le vecteur correspondant à la force exercée sur le nœud, dont dépend le déplacement du nœud à chaque étape de l'algorithme, est noté $force(a)$.

Nous présentons maintenant les différences de l'algorithme par rapport au cas non orienté. Nous distinguons l'une des composantes du vecteur position, la composante *verticale*, qui traduit la notion de haut vers le bas utilisée dans le critère i. L'algorithme présenté est incrémental, et le calcul des forces diffère selon l'ordre dans lequel les nœuds et arêtes sont ajoutées au graphe. Nous introduisons de nouvelles données pour chaque nœud a du graphe :

- le *nœud d'attache* de a , noté $attache(a)$ est le nœud à partir duquel nous calculons la composante verticale du vecteur position de a ,
- la *position d'attache* de a , notée $posattache(a)$, est la position relative de a par rapport à son nœud d'attache,
- l'*identificateur de composante*, noté $composante(a)$, est un entier qui est identique entre deux nœuds appartenant à la même composante connexe du graphe.

Les nœuds d'attache sont calculés au fur et à mesure que l'on ajoute des arêtes au graphe. Lors de l'ajout d'un nœud a , $attache(a)$ est initialisé à `nil`, et $composante(a)$ reçoit une valeur non encore utilisée. Lors de l'ajout d'une arête, on regarde tout d'abord si les deux nœuds reliés font partie de la même composante connexe. Si c'est le cas, on ne fait rien. Sinon, si l'une des extrémités de l'arête n'a pas de nœud d'attache, on lui attache l'autre extrémité. La position d'attache est déterminée en fonction de la valeur $\delta_{opt}(a, b)$, la distance verticale optimale entre les nœuds a et b . Les nœuds a et b font maintenant partie de la même composante connexe : on met donc à jour les valeurs des composantes des nœuds.

Algorithme (Ajout d'une arête)

```

PROCEDURE ajoute_arete(a:nœud, b:nœud)
  IF composante(a)≠composante(b) alors
    IF attache(b)=nil THEN
      attache(b)=a;
      pos_attache(b)=δ_opt(a, b);
    ELSE_IF attache(a)=nil THEN
      attache(a)=b;
      pos_attache(a)=-δ_opt(a, b);
    END_IF;
    tmp:=composante(b);
    FOR ALL nœud n DO
      IF composante(n) = tmp THEN
        composante(n) :=composante(a);
      END_IF;
    END_FOR;
  END_IF;
END_PROCEDURE;

```

L'algorithme ci-dessus, en ne permettant de créer des liens *attache* qu'entre des nœuds appartenant, avant l'ajout de l'arête, à des composantes connexes différentes, garantit qu'il n'y a pas de cycles selon les liens *attache*. Le coût de l'algorithme est en $O(n)$, où n est le nombre de nœuds du graphe.

Nous dirons que deux nœuds sont situés à un *même niveau* s'ils sont séparés par une distance verticale inférieure à un nombre δ_v . Pour deux nœuds situés au même niveau, nous appliquons les mêmes procédures que dans le cas non-orienté, mais en ignorant la composante verticale. Si deux nœuds ne sont pas sur un même niveau, nous ignorons les forces de répulsion ; si ces deux nœuds sont reliés par une arête, nous essayons de faire en sorte que l'arête soit la plus verticale possible. Dans les procédures qui suivent, nous

notons \vec{v} le vecteur dont toutes les composantes valent un, excepté pour la composante verticale qui est nulle, et \vec{v}' le vecteur dont toutes les composantes sont nulles, excepté pour la composante verticale qui vaut un :

Algorithme (Attraction des nœuds)

```

PROCEDURE attraction(a:nœud, b:nœud)
  dz:=|b.z - a.z|;
  IF dz > δ THEN
    force(a):=force(a) +  $\frac{\vec{a}\vec{b}\cdot\vec{v}}{2}$ ;
    force(b):=force(a) -  $\frac{\vec{a}\vec{b}\cdot\vec{v}}{2}$ ;
  ELSE
    d:=|| $\vec{a}\vec{b}\cdot\vec{v}$ ||;
    IF d ≤ d_max THEN
      f:= $\frac{d^2}{d_{opt}(a,b)}$ ;
      force(a):=force(a) + f ·  $\frac{\vec{a}\vec{b}\cdot\vec{v}}{d}$ ;
      force(b):=force(b) - f ·  $\frac{\vec{a}\vec{b}\cdot\vec{v}}{d}$ ;
    END_IF;
  END_IF;
END_PROCEDURE;

```

Algorithme (Répulsion des nœuds)

```

PROCEDURE repulsion(a:nœud, b:nœud)
  dz:=||position(b). $\vec{v}'$ ||;
  IF dz ≤ δ THEN
    d:=|| $\vec{a}\vec{b}\cdot\vec{v}'$ ||;
    f:=- $\frac{d_{opt}(a,b)^2}{d}$ ;
    force(a):=force(a) + f ·  $\frac{\vec{a}\vec{b}\cdot\vec{v}}{d}$ ;
    force(b):=force(b) - f ·  $\frac{\vec{a}\vec{b}\cdot\vec{v}}{d}$ ;
  END_IF;
END_PROCEDURE;

```

Nous ajoutons également des interactions visant à faire s'écartier les nœuds des arêtes. Par rapport au cas non orienté présenté dans le cas de l'algorithme FRCC, on ne calcule plus la projection orthogonale sur la droite passant par l'arête, mais l'intersection entre le plan horizontal passant par le nœud et la droite passant par l'arête.

Algorithme (Répulsion des nœuds)

```

PROCEDURE repulsion_noeud_arete(a:nœud, e:arete)
  s O→ t := e;
  α := ( $\vec{s}\vec{a}\cdot\vec{v}$ ) / ( $\vec{a}\vec{t}\cdot\vec{v}$ );
  IF 0 ≤ α ≤ 1 THEN
    LET i SUCH THAT  $\vec{s}\vec{i} = \alpha \cdot \vec{s}\vec{t}$ ;
    d := || $\vec{i}\vec{a}$ ||;
    IF d ≤ d_max THEN
      f := - $\frac{d_{opt}(a,b)^2}{d}$ ;
      force(a) := force(a) + f ·  $\frac{\vec{a}\vec{i}\cdot\vec{v}}{d}$ ;
      force(b) := force(b) - f ·  $\frac{\vec{a}\vec{i}\cdot\vec{v}\vec{e}\vec{c}\vec{v}}{d}$ ;
    END_IF;
  END_IF;
END_PROCEDURE;

```

En plus des attractions et des répulsions entre nœuds, il convient d'essayer, à chaque étape, de placer chaque nœud en fonction de son père d'attache.

Algorithme (Attachement des nœuds)

```

PROCEDURE attachement(a:nœud)

```

```

IF attache(a)≠nil THEN
  b:=attache(a);
  position(a).v→:=position(a).v→ + || $\vec{ab}$ .v→|| + posattache(a);
END_IF;
END_PROCEDURE;

```

Il convient ensuite de prendre en compte chacune de ces interactions dans une procédure **etape** similaire à celle présentée dans le cas non orienté.

Dans le cas d'un dessin dans le plan, cette méthode n'est pas satisfaisante, du fait du choix de la longueur uniforme des arêtes qui conduit à de nombreux croisements. Son utilisation dans un espace en trois dimensions comporte par contre plusieurs avantages: les croisements sont moins fréquents et c'est une méthode incrémentale bien adaptée à la représentation de graphes modifiés dynamiquement.

3.6.2 Algorithme classique

La méthode proposée par K. Sugiyama, S. Tagawa et M. Toda [52] pour la représentation des graphes orientés consiste à vérifier successivement les critères esthétiques suivants :

- i. les arêtes doivent être dirigées de haut en bas,
- ii. les nœuds doivent être distribués uniformément sur la page,
- iii. les croisements d'arêtes doivent être évités.

Les étapes de l'algorithme consistent à transformer le graphe en graphe acyclique en retournant certaines arêtes du graphe (critère i), placer les nœuds par niveaux (critères i et ii), supprimer les arêtes entre des nœuds de niveaux éloignés en ajoutant des nœuds supplémentaires pour permettre de minimiser les croisements d'arêtes (critère iii).

3.6.2.1 Suppression des cycles

La première étape l'algorithme consiste à supprimer les cycles en inversant un minimum d'arêtes pour permettre de suivre le critère i. Cette étape sera décrite dans le chapitre 4 en section 4.3.3.1, et nous supposons ici que nous disposons dès le départ d'un graphe acyclique.

3.6.2.2 Placement des nœuds par niveaux

La seconde étape de l'algorithme consiste à déterminer les positions verticales des nœuds du graphe, en suivant les critères i et ii. Pour cela nous choisissons de partitionner les nœuds du graphe en h sous-ensembles N_1, \dots, N_h , appelés les *niveaux* du graphe, tels que si $a \rightarrow b$ avec $a \in N_i$ et $b \in N_j$, alors $i < j$. h est appelé la *hauteur* du graphe, le cardinal maximal des ensembles N_i étant sa *largeur*.

Le problème consistant à choisir à quel niveau placer un nœud se rapproche des problèmes d'ordonnement de *tâches*, qui sont des problèmes classiques en recherche opérationnelle. Les nœuds du graphe que nous souhaitons afficher sont les tâches, les arêtes les contraintes de précédence. Une première idée consiste naturellement à utiliser les algorithmes classiques d'ordonnement. Nous pouvons par exemple placer les nœuds selon le chemin le plus court ou le plus long. L'inconvénient majeur de cette méthode est que celle-ci peut conduire à des graphes très larges. Pour en limiter les effets, on peut limiter le nombre de nœuds par niveaux. Le problème devient alors similaire à celui de l'ordonnement de tâches de durée unitaire sur machines parallèles ayant un nombre borné de processeurs. Ce problème a été très étudié et est NP-complet quand le nombre de processeurs est supérieur ou égal à trois [8, 32]. La méthode la plus couramment utilisée est celle dite d'*ordonnement de listes* [8, 9, 30]. Le principe consiste à associer des priorités aux différentes tâches, puis à choisir la tâche de priorité maximale compatible avec les contraintes de précédence. Soit p le nombre de processeurs disponibles. Nous notons E l'ensemble des tâches qu'il reste à exécuter, P les contraintes de précédence et $niveau(t)$ la date de début d'exécution de la tâche t . $D(k)$ correspond à l'ensemble des tâches qu'il est possible de faire débiter au temps k . Le principe de l'algorithme consiste à choisir, parmi les tâches qu'il est possible d'exécuter au temps t , ce qui dépend uniquement des contraintes de précédence, celle de priorité maximale. Si le nombre maximal

de tâches que l'on peut exécuter simultanément est atteint, ou si aucune tâche ne peut être exécutée, on cherche alors à placer les tâches restantes au niveau suivant, et ainsi de suite jusqu'à ce que toutes les tâches soient placées.

Algorithme (Ordonnancement de listes)

- *Données* : un graphe orienté acyclique,
- *Résultat* : le niveau de chaque nœud du graphe est connu.

```

k:=1;
i:=1;
D(1):={t'∈E, ∄t''∈E, t' o→o t''};
D(2):=∅;
WHILE E≠∅ DO
  IF ∃ t∈D(k) SUCH THAT ∀t'∈D(k), priorite(t)≥priorite(t');
    E:=E - {t};
    D(k):=D(k) - {t};
    niveau(t):=k;
    i:=i + 1;
    D(k + 1):=D(k + 1) ∪ {t'∈VO→O+(t), ∄t''∈(E ∩ VO→O-(t''))};
    IF i = p + 1 THEN
      D(k + 1):=D(k + 1) ∪ D(k);
      D(k + 2):=∅;
      k:=k + 1;
      i:=1;
    END_IF;
  ELSE
    D(k + 2):=∅;
    k:=k + 1;
    i:=1;
  END_IF;
END_WHILE

```

La difficulté principale des algorithmes d'ordonnancement de liste réside dans la détermination des priorités des tâches qui conduisent à de bons résultats. Une méthode couramment utilisée consiste à choisir la priorité selon le *chemin critique*, c'est-à-dire selon le maximum des longueurs des chemins allant de cette tâche à une tâche terminale. Si h correspond à la hauteur de l'ordonnancement construit à l'aide de cette heuristique, et h_{opt} à la hauteur de l'ordonnancement de hauteur minimale qu'il est possible d'obtenir, on obtient la relation suivante pour des tâches unitaires et $p \geq 3$:

$$h \leq h_{opt} \left(2 - \frac{1}{p-1} \right). \quad (3.2)$$

Remarque. Il est possible d'améliorer ce résultat en précisant la notion de priorité pour les nœuds ayant même chemin critique. La méthode de Coffman-Graham [8] permet de résoudre le problème pour des tâches unitaires et pour $p = 2$ de façon optimale. Dans le cas où $p \geq 3$, on obtient à l'aide de cet algorithme la relation suivante :

$$h \leq h_{opt} \left(2 - \frac{2}{p} \right).$$

Cet algorithme a une complexité en $O(n^2)$ si l'on suppose construite la fermeture transitive du graphe (en $O(n^3)$ avec un algorithme classique). Il existe une variante quasi-linéaire de cet algorithme, proposée par H. N. Gabow [28], qui ne nécessite pas le calcul de la fermeture transitive du graphe, mais qui est difficile à mettre en œuvre. .

Nous disposons donc, avec les algorithmes d'ordonnancement de listes, d'une méthode simple permettant de borner la largeur d'un graphe en multipliant au pire par deux la hauteur du graphe par rapport à la hauteur minimale qu'il est possible d'obtenir. Cependant, nous n'avons pas tenu compte de l'aspect esthétique du graphe dans le choix des niveaux. En particulier, les placements de type ordonnancement de listes placent les nœuds dès que cela est possible. Cela conduit à des graphes pour lesquels les extrémités

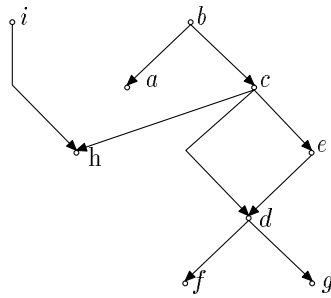


Fig. 3.33: Placement des nœuds par niveaux suivant un algorithme d'ordonnancement de liste suivant le chemin critique.

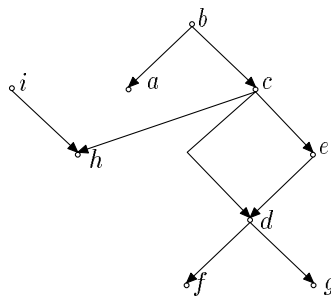


Fig. 3.34: Placement des nœuds par niveaux avec minimisation des longueurs des arêtes par la méthode du simplexe.

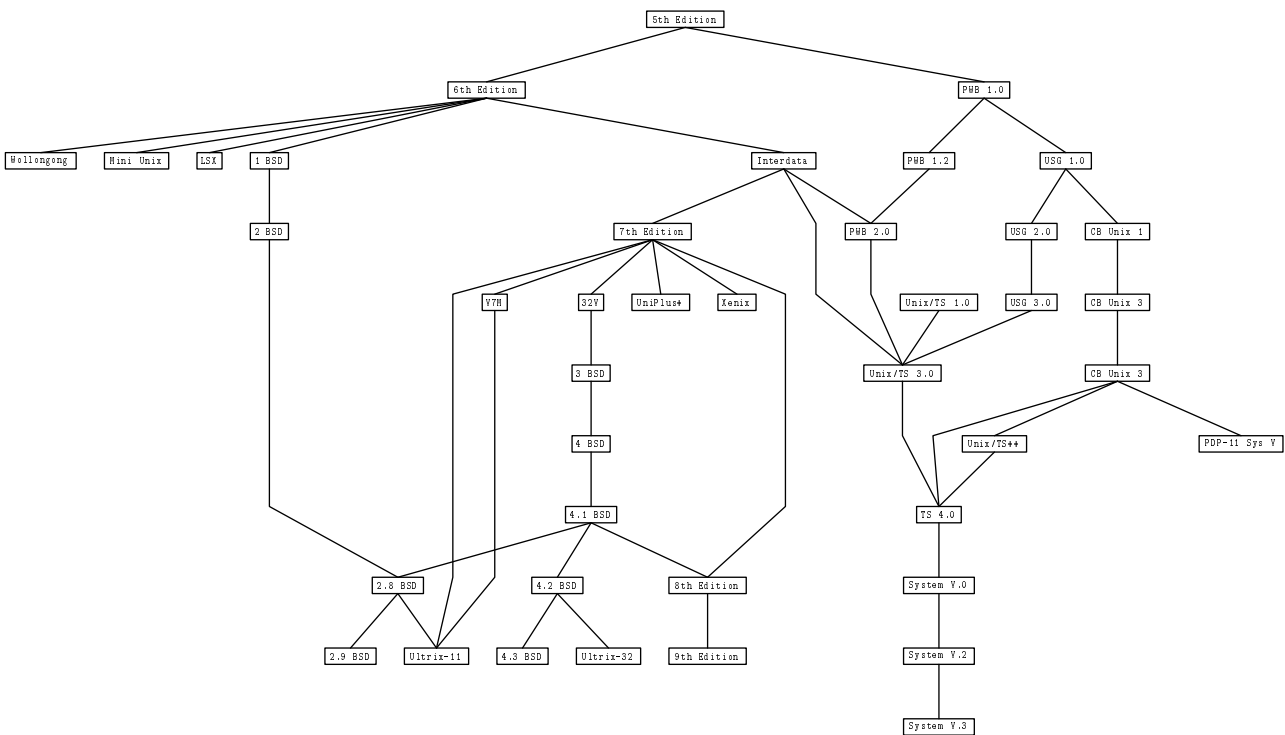


Fig. 3.35: Hiérarchie Unix.

des arêtes peuvent être placées sur des niveaux très distants, comme par exemple le nœud i sur la figure 3.33. Pour résoudre ce problème, on peut choisir comme critère de placement des nœuds le critère de la différence des niveaux des nœuds reliés par des arêtes. Si l'on note $origine(e)$ l'origine d'une arête e , et $destination(e)$ sa destination, on cherche alors à minimiser la somme des différences de niveaux entre les nœuds extrémités des arêtes du graphe :

$$\sum_{e \in G} \text{niveau}(destination(e)) - \text{niveau}(origine(e)),$$

avec les contraintes $\forall e \in G, \text{niveau}(source(e)) \leq \text{niveau}(destination(e))$. Les contraintes et la fonction à minimiser sont linéaires, et le problème peut être résolu par un algorithme de type *simplexe* sur les entiers [7, 40]. Cette méthode est utilisée initialement dans le système DAG [29] et conduit à de très bons résultats. La minimisation de la longueur des arêtes entraîne de surcroît une hauteur du graphe minimale.

Si le graphe est composé de k nœuds n_1, \dots, n_k , et l arêtes, et que l'on note x_1, \dots, x_k les hauteurs cherchées pour les nœuds, le problème revient à minimiser :

$$Cx \text{ avec } Ax \leq b$$

où x , C , A et b sont définis par :

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_k \end{pmatrix}$$

$$C = (\partial_{\circ \rightarrow \circ}^-(n_1) - \partial_{\circ \rightarrow \circ}^+(n_1), \dots, \partial_{\circ \rightarrow \circ}^-(n_k) - \partial_{\circ \rightarrow \circ}^+(n_k))$$

$$A = \begin{pmatrix} a_{11} & \dots & a_{k1} \\ \vdots & a_{ij} & \vdots \\ a_{1l} & \dots & a_{kl} \end{pmatrix}, a_{ij} = \begin{cases} 1 & \text{si } n_i \circ \rightarrow n_j, \\ -1 & \text{si } n_j \circ \rightarrow n_i, \\ 0 & \text{sinon} \end{cases}$$

$$b = \begin{pmatrix} -1 \\ \vdots \\ -1 \end{pmatrix}$$

La figure 3.35 correspond à l'exemple de la figure 2 présenté dans l'article [29], et a été réalisée à l'aide de l'algorithme de tracé de graphes orientés acycliques.

3.6.2.3 Ajout de nœuds temporaires

Pour simplifier les étapes ultérieures de minimisation du nombre de croisements entre arêtes, nous ajoutons des *nœuds temporaires* sur les longues arêtes (nœuds grisés sur la figure 3.36). En d'autres termes, nous modifions le graphe de sorte que deux nœuds reliés par une arête se trouvent sur des niveaux successifs.

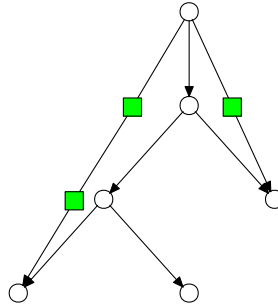


Fig. 3.36: Ajout de nœuds temporaires.

Algorithme (Ajout de nœuds temporaires)

```

FOR ALL arete OF KIND  $a \rightarrow b$  DO
   $nb\_niveaux := \text{niveau}(b) - \text{niveau}(a)$ ;
  supprimer l'arete  $a \rightarrow b$ ;
   $ai := a$ ;
  FOR  $i$  FROM 1 TO  $nb\_niveaux - 1$  DO
    ajouter un nœud temporaire  $bi$ ;
    ajouter l'arete  $ai \rightarrow bi$ ;
     $ai := bi$ ;
  END_FOR;
  ajouter l'arete  $ai \rightarrow b$ ;
END_FOR;

```

Après cette étape on a la propriété suivante :

$$\forall a \rightarrow b, \text{niveau}(b) = 1 + \text{niveau}(a)$$

Remarque. La méthode du simplexe, utilisée lors du choix des niveaux des nœuds, assure que le nombre de nœuds temporaires ajoutés durant cette phase est minimal.

3.6.2.4 Ordonner les nœuds de même niveau

Le but de cette étape est d'ordonner les nœuds d'un même niveau de sorte que le nombre de croisements entre les arêtes soit minimal. Ce problème est NP-complet, y compris si l'on ne considère que deux niveaux [31]. Nous présentons ici une heuristique basée sur l'intuition selon laquelle les nœuds doivent être placés le plus près possible des nœuds auxquels ils sont reliés par une arête. Nous choisissons de placer les nœuds au point médian des emplacements des nœuds qui leur sont voisins. Nous supposons connues une fonction **mediane**(n : nœud) qui calcule la position médiane des points de $V_{\rightarrow}^+(n) \cup V_{\rightarrow}^-(n)$. Une étape **ordonne** de l'algorithme s'écrit alors :

Algorithme (Ordonner les nœuds par niveaux)

```

PROCEDURE ordonne()
  FOR  $i$  FROM 2 TO  $h$ ,
    THEN FOR  $i$  FROM  $h - 1$  TO 1 DO
      FOR ALL  $n \in L_i$  DO
         $x(n) := \text{mediane}(n)$ ;
      END_FOR;
      ordonner( $L_i$ ) selon  $x$ ;
       $an := \text{nil}$ ;
      FOR ALL  $n \in L_i$  DO
        IF  $an \neq \text{nil}$  THEN
           $x(n) := x(an) + \delta_h$ ;
        END_IF;
         $an := n$ ;
      END_FOR;
    END_FOR;
  END_PROCEDURE

```

Il est possible d'augmenter l'efficacité de l'algorithme précédent en diminuant le nombre de croisements entre arêtes, à l'aide d'une méthode de minimisation locale du nombre de croisements. Pour chaque paire de nœuds a et b successifs du niveau L_i , on échange les positions de a et b si cela diminue le nombre de croisements. On s'arrête dès qu'il n'y a plus d'échange possible.

Remarque. Il est possible d'utiliser d'autres heuristiques pour le placement des nœuds sur les niveaux. Il est possible de remplacer l'heuristique médiane par une heuristique barycentre. Il est également possible d'utiliser une méthode globale qui n'utilise pas la méthode d'échange local. Cette méthode, proposée initialement par W. Tutte [55] revient, une fois déterminées les positions sur le premier et le dernier niveau, à résoudre un système linéaire d'équations. D'après [21], si $x(n)$ représente l'abscisse du nœud n , on résout un système d'équations de la forme :

$$x(n) = \frac{\sum_{s \in V_{\rightarrow}^+(n)} x(s)}{2\partial_{\rightarrow}^+(n)} + \frac{\sum_{s \in V_{\rightarrow}^-(n)} x(s)}{2\partial_{\rightarrow}^-(n)}$$

3.6.2.5 Placement vertical et finitions

Le placement vertical des nœuds se fait en fonction des niveaux. Pour chaque niveau, on détermine le nœud de hauteur maximal, et l'on place chaque nœud selon son niveau. Une fois les positions verticales et horizontales connues, il reste ensuite à remplacer les nœuds temporaires par des arêtes brisées et à restituer les arêtes qui ont été retournées.

3.6.2.6 Exemples

3.6.2.6.1 Dérivations d'un système de réécriture. Pour illustrer l'algorithme de tracé de graphes orientés, nous construisons le graphe acyclique correspondant à l'ensemble des dérivations possibles, issues du terme $ss0 \times ss0$, du système de réécriture suivant, défini sur l'alphabet $\{0, s, +, \times\}$:









$$\left\{ \begin{array}{l} x + 0 \rightarrow x \\ x + sy \rightarrow s(x + y) \\ x \times 0 \rightarrow 0 \\ x \times sy \rightarrow (x \times y) + x \end{array} \right. \quad (3.3)$$

Ce système permet d'effectuer l'addition et la multiplication de deux entiers donnés sous forme unaire. Le graphe de toutes les dérivations issues du terme $ss0 \times ss0$ est représenté sur la figure 3.37. Les entiers $s0$, $ss0$, $sss0$ et $ssss0$ sont notés 1, 2, 3 et 4 sur le dessin.

3.6.2.6.2 Nœuds de tailles différentes. La figure 3.38 est un exemple de graphe dont les nœuds ont des tailles différentes. Sur cet exemple, nous remarquons que le placement des nœuds est effectué par niveau, contrairement au choix que nous avons effectué dans le cas des arbres.

3.7 Conclusion

Dans ce chapitre, nous avons présenté plusieurs algorithmes de tracé de graphes dédiés à des classes particulières. Les algorithmes présentés n'ont pas tous été étudiés dans le même but. Certains cherchent à apporter un éclairage nouveau au problème du tracé de certaines classes de graphes : c'est le cas de l'algorithme de tracé de graphes orientés présenté au point 3.6.1 et celui de tracé des graphes non orientés avec conservation des croisements présenté au point 3.5.3. Les autres algorithmes ont été étudiés dans un but précis, permettre leur utilisation conjointement avec l'algorithme que nous allons présenter dans le chapitre suivant. Pour cette raison, nous avons apporté un intérêt particulier à l'extension d'algorithmes, connus pour être performants, au cas de graphes possédant des nœuds de tailles arbitrairement grandes. C'est le cas de l'algorithme de tracé de graphes orientés du point 3.6.2 et celui de tracé des graphes non orientés du point 3.5. Ils nous ont permis de nous rendre compte de la variété des techniques mises en œuvre pour résoudre les problèmes de tracé de graphes. Enfin, l'algorithme de tracé d'arbres enracinés du point 3.3 montre que la méthode des liens, moyennant une modification simple, peut être utilisée dans le cas du tracé de graphes généraux dont les nœuds ne sont pas placés obligatoirement par niveau. Tous ces algorithmes ont été implantés en pratique dans les logiciels que nous présentons au chapitre 5.

Algorithme	Type de graphe			
FR		+		
FR modifié		•	+	
FRCC		•	♠	+
FR orienté		•	+	
STT		♣		
STT modifié		•	♣	♥
RT				
RT modifié		•	◇	

La signification des symboles utilisés est la suivante :

- ● : permet le tracé de graphes ayant des nœuds de tailles différentes,
- + : algorithme incrémental. Le tracé obtenu après une modification mineure du graphe dépend du tracé dont on disposait avant modification, et les différences entre les deux dessins sont peu importantes,
- ♠ : la partition des nœuds du graphe est conservée au cours des étapes de l'algorithme,
- ♣ : possède une heuristique dédiée à la minimisation des croisements d'arêtes,
- ♥ : minimise le nombre de nœuds temporaires ajoutés au graphe,
- ♦ : placement des nœuds non obligatoirement par niveaux.

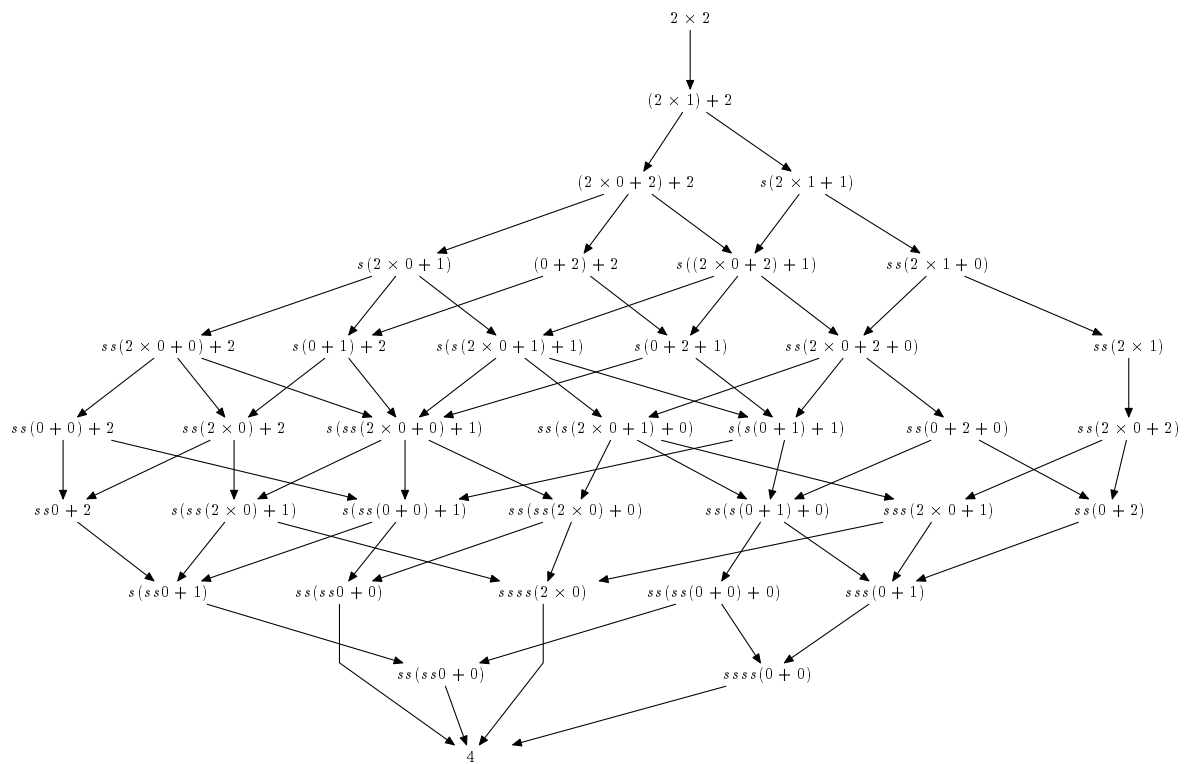


Fig. 3.37: Graphe de toutes les dérivations issues du terme $ss0 \times ss0$, pour le système de réécriture (3.3).

Chapitre 4

Représentation graphique de graphes composés

Nous abordons maintenant le problème de la représentation graphique des structures décomposables. Nous nous intéressons pour cela à la représentation de graphes particuliers, les graphes composés, qui possèdent à la fois des relations d'adjacence et d'inclusion entre les nœuds. Les relations d'inclusion permettent en effet de représenter de façon naturelle les structures construites de façon récursives.

La représentation graphique des graphes possédant des relations d'inclusion n'a été abordée que relativement récemment. On peut citer la réalisation du système `D-ABDUCTOR` par K. Sugiyama et K. Misue [50, 51] pour la représentation de graphes orientés, ou encore les travaux de P. Eades, Q. Feng et X. Lin concernant le tracé de graphes orientés planaires ou en trois dimensions [17, 18]. Notre approche se démarque fortement des algorithmes proposés antérieurement pour la représentation de graphes ayant des relations d'inclusion, et est guidée par la nature récursive et la variété des structures décomposables à représenter. Tout d'abord, les graphes que nous considérons possèdent potentiellement à la fois des arêtes orientées et non orientées. De plus, pour permettre une représentation naturelle des structures décomposables, nous avons choisi de permettre de faire cohabiter plusieurs algorithmes classiques de tracé de graphe au sein d'une même représentation. Ces deux choix nous conduisent à concevoir entièrement de nouveaux algorithmes pour la représentation des graphes composés, dont la particularité essentielle est de permettre l'utilisation d'algorithmes de tracé en nombre potentiellement infini.

Ce chapitre contient les résultats les plus originaux de la thèse. Le premier concerne la réalisation de deux algorithmes dont la combinaison permet la représentation satisfaisante, dans la plupart des cas, de toutes les structures décomposables. Le premier algorithme, présenté en section 4.2.1.2, permet de tracer des graphes composés particuliers, les graphes imbriqués. Le deuxième algorithme, décrit en section 4.4.1, permet de convertir de façon systématique des structures décomposables en graphes imbriqués. Ceci permet au final de disposer d'une première méthode pour la représentation des structures décomposables, qui s'avère bien adaptée pour faire ressortir les propriétés des constructeurs décrivant la structure.

Les résultats suivants du chapitre visent à permettre une plus grande souplesse dans la représentation des structures décomposables. Pour cela, nous cherchons à représenter des sous-classes des graphes composés de plus en plus larges, en laissant totalement libre la conversion des structures en graphes décomposables. Nous proposons ainsi en section 4.5 un algorithme permettant de représenter des graphes décomposables enracinés, qui consiste à ramener le problème à celui du tracé des graphes imbriqués. C'est dans cette partie que se situe la difficulté la plus importante du chapitre. Ceci nous permet ensuite de proposer en section 4.6 un algorithme de tracé de graphes composés quelconques. Enfin, nous terminons en regardant quelques utilisations des graphes composés dans le cadre général de la représentation de graphes.

4.1 Introduction

Nous avons vu dans le chapitre 1 que la théorie des structures décomposables permettait de décrire des objets très variés, parmi lesquels des cycles, des arbres enracinés, des graphes. Nous avons vu également qu'il était possible de substituer les objets de base d'une structure décomposable par n'importe quelle autre structure décomposable. Une méthode naturelle pour représenter ces substitutions consiste à les dessiner de façon imbriquée.

Regardons par exemple le cas de la spécification de structures décomposables suivante :

$$\left\{ \begin{array}{l} A \rightarrow \text{Union}(\mathbf{a}, B) \\ B \rightarrow \text{Union}(C, D) \\ D \rightarrow \text{Prod}(A, B, B) \\ C \rightarrow \text{Cycle}(\mathbf{b}) \\ \mathbf{a} \rightsquigarrow \text{Atom}, \\ \mathbf{b} \rightsquigarrow \text{Atom}. \end{array} \right. \quad (4.1)$$

Une interprétation de cette spécification pourrait être : « des arbres A , dont les feuilles sont des cycles, et dont les nœuds internes sont des arbres A . » Un élément de A est la structure :

```
prod(prod(a,
  cycle(b),
  prod(prod(a, cycle(b), cycle(b)),
    cycle(b, b, b),
    cycle(b))),
  cycle(b, b, b),
  cycle(b, b, b, b, b))
```

dont une représentation naturelle pourrait être celle de la figure 4.1. Nous avons représenté les terminaux \mathbf{a} par des carrés gris foncé, les terminaux \mathbf{b} par des cercles gris foncé, les nœuds internes des structures arborescentes par des carrés blancs, leurs feuilles par des carrés gris clair (contenant des cycles). Pour représenter des cycles, l'algorithme le mieux adapté consiste à placer les nœuds sur un cercle. Pour représenter des arbres enracinés, l'algorithme à utiliser est celui présenté au chapitre 3. Pour représenter des structures décomposables de façon satisfaisante, nous sommes donc amenés à faire cohabiter plusieurs algorithmes sur le même dessin. De plus ces algorithmes, du fait de l'imbrication des structures, doivent permettre de prendre en compte des structures dont les nœuds sont de taille arbitrairement grande, d'où notre préoccupation à ce sujet dans le chapitre 3.

Cet exemple illustre la méthode que nous présentons dans la première partie de ce chapitre : nous représentons les structures décomposables par des graphes imbriqués [2]. Ceci s'avère bien adapté à la représentation de la plupart des structures décomposables. Nous nous intéressons dans la deuxième partie du chapitre à un problème plus vaste, qui consiste à afficher des graphes quelconques possédant des relations d'inclusion et d'adjacence. Le tracé de tels graphes a été étudié ces dernières années.

4.2 Définitions

4.2.1 Graphes composés

Définition 4.2.1 (Graphe composé)

Un *graphe composé* est un couple (N, E) , où N est un ensemble fini de *nœuds* et E un ensemble fini avec répétitions éventuelles de *liens* entre deux nœuds de N . Un lien entre deux nœuds a et b peut être d'un des trois types suivants :

- lien symétrique *arête non orientée* noté $a \circ \circ b$. a et b sont les *extrémités* du lien,
- lien *arête orientée* noté $a \circ \rightarrow b$. a et b sont les *extrémités* du lien, a est appelé l'*origine*, b la *destination* du lien,

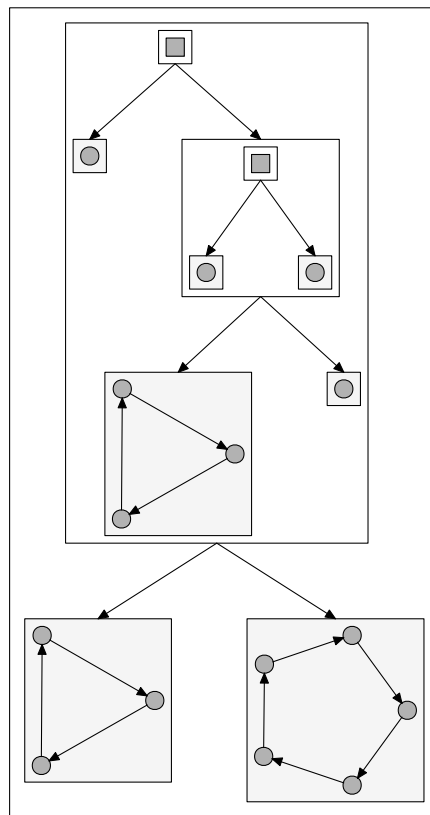


Fig. 4.1: Représentation d'une structure de la spécification (4.1).

- lien d'inclusion noté $a \mathcal{G} \circ b$. a et b sont les *extrémités* du lien, a est appelé l'*origine*, b la *destination* du lien. a est un *père d'inclusion* de b , b un *fil d'inclusion* de a . Nous dirons que a contient b et que b est inclus dans a .

Un lien de la forme $a \circ \circ a$, $a \circ \circ a$, ou $a \mathcal{G} \circ a$, est un *lien réflexif*. .

Ces graphes possèdent à la fois des liens d'inclusion et des liens par des arêtes. Ils contiennent les classes des graphes orientés (ne possédant que des liens $\circ \circ$) et non orientés (ne possédant que des liens $\circ \circ$). On peut dire qu'un graphe composé est équivalent à la donnée de deux graphes orientés ($\circ \circ$, $\mathcal{G} \circ$) et d'un graphe non orienté (*unorient*).

Les notions définies sur les graphes orientés et non orientés s'étendent naturellement au cas des graphes composés, en précisant si nécessaire le type de liens considéré. Des notions similaires sont associées aux liens d'inclusion :

Définition 4.2.2 (Voisinage et degré d'inclusion d'un nœud)

On définit, pour un graphe composé $G = (N, E)$, les notions de voisinage et de degré d'inclusion d'un nœud suivantes :

- le *voisinage entrant d'inclusion* d'un nœud $a \in N$ est l'ensemble $V_{\mathcal{G} \circ}^-(a) = \{f \in N, f \mathcal{G} \circ a\}$. Le *degré entrant d'inclusion* du nœud $a \in N$, noté $\partial_{\mathcal{G} \circ}^-(a)$, est le nombre de liens d'inclusion ayant pour destination le nœud a .
- le *voisinage sortant d'inclusion* d'un nœud $a \in N$ est l'ensemble $V_{\mathcal{G} \circ}^+(a) = \{s \in N, a \mathcal{G} \circ s\}$. Le *degré sortant d'inclusion* d'un nœud $a \in N$, noté $\partial_{\mathcal{G} \circ}^+(a)$ est le nombre de liens d'inclusion ayant le nœud a comme origine. .

Définition 4.2.3 (Chemin et cycle d'inclusion)

Un *chemin d'inclusion* de longueur k est une suite de liens d'inclusion l_1, \dots, l_k , telle que pour tout $i \in [1, k-1]$, la destination de l_i est égale à l'origine de l_{i+1} . L'origine du lien l_1 est l'*origine du chemin*, la destination du lien l_k est la *destination du chemin*. Si l'origine et la destination du chemin sont égales, le chemin est un *cycle d'inclusion*. .

Définition 4.2.4 (Graphe composé connexe selon les liens d'inclusion)

Un graphe composé $G = (N, E)$ est *connexe* selon les liens d'inclusion, si pour tout couple de nœuds $(a, b) \in N \times N$, il existe un chemin d'inclusion d'origine a et de destination b dans le graphe composé $G' = (N, E \cup E')$, avec si $a \mathcal{G} \circ b \in E$, alors $b \mathcal{G} \circ a \in E'$. .

4.2.1.1 Graphes composés enracinés

Les graphes enracinés sont les graphes composés pour lesquels le problème de tracé sous la forme de structures imbriquées à un sens : intuitivement, nous ne voulons pas considérer les graphes dont les nœuds peuvent être inclus dans eux-mêmes par transitivité. Les graphes enracinés sont les graphes composés tels que la restriction du graphe aux seuls liens d'inclusion est un arbre enraciné.

Définition 4.2.5 (Graphe enraciné)

Un *graphe enraciné* $G = (N, E)$ est un graphe composé connexe selon les liens d'inclusion, tel que pour chaque nœud $a \in N$, excepté pour un nœud unique de degré entrant d'inclusion nul, que l'on nomme *racine* du graphe, il existe un et un seul nœud b tel que a est inclus dans b . Nous noterons alors $b = \text{pere}_{\mathcal{G} \circ}(a)$. .

Définition 4.2.6 (Niveau d'inclusion)

Le *niveau d'inclusion* d'un nœud n d'un graphe enraciné $G = (N, E)$, est la longueur du chemin d'inclusion ayant pour origine la racine d'inclusion du graphe et pour destination le nœud n . En d'autres termes, si r est le nœud racine du graphe, on a :

$$\begin{cases} \text{niveau}(r) = 0, \\ \forall n \in N, n \neq r, \text{niveau}(n) = \text{niveau}(\text{pere}_{\mathcal{G} \circ}(n)) + 1. \end{cases}$$
.

4.2.1.2 Graphes composés imbriqués

Les graphes imbriqués sont les graphes pour lesquels il est possible de définir un algorithme simple de tracé, que nous présenterons dans la section 4.3.

Définition 4.2.7 (Graphe imbriqué)

Un *graphe imbriqué* $G = (N, E)$ est un graphe enraciné tel que les propriétés suivantes sont vérifiées :

- i. les nœuds reliés par une arête ont même père d'inclusion :

$$\forall a, b \in N \text{ tels que } \left\{ \begin{array}{l} a \circ \rightarrow b \\ a \circ \circ b \end{array} \right. , \text{pere}_{\circ \circ}(a) = \text{pere}_{\circ \circ}(b),$$

- ii. les pères d'inclusion des nœuds reliés par des arêtes orientées et ceux reliés par des arêtes non orientées sont différents :

$$\forall a, b, a', b' \in N \text{ tels que } \left\{ \begin{array}{l} a \circ \rightarrow b \\ a' \circ \circ b' \end{array} \right. , \text{pere}_{\circ \circ}(a) \neq \text{pere}_{\circ \circ}(a').$$

Si $a \circ \rightarrow b$ (respectivement $a \circ \circ b$), alors le *type de mode d'inclusion* du nœud $\text{pere}_{\circ \circ}(a)$ est orienté (respectivement non orienté).

4.2.2 Notations

Notation. Nous supposons qu'il est possible d'associer les informations suivantes à un nœud n du graphe :

- $(x(n), y(n))$: coordonnées du centre du nœud n , absolues ou relatives à celles du père d'inclusion de n , selon le contexte,
- $\text{boite}(n)$: la *boîte* du nœud n , c'est-à-dire le plus petit rectangle contenant le tracé du nœud n , déterminé par les coordonnées absolues du centre du rectangle, notées $\text{centre_boite}(n)$, ainsi que par sa largeur et sa hauteur.

On définit l'union de deux boîtes de nœuds comme la construction de la plus petite boîte contenant les deux autres.

4.3 Algorithme de tracé de graphes composés imbriqués

Comme nous l'avons déjà indiqué dans l'introduction, pour obtenir des tracés intuitifs des structures décomposables, il faut pouvoir appliquer différents algorithmes à différentes parties du graphe. Nous ajoutons donc, à chaque nœud a du graphe imbriqué, une information supplémentaire, le *mode d'inclusion* du nœud, noté $\text{mode}_{\circ \circ}(a)$. Le mode d'inclusion d'un nœud a permet d'indiquer le nom de l'algorithme à appliquer aux nœuds inclus dans a . Si a est à mode orienté (respectivement non orienté), le mode d'inclusion de a est le nom d'un algorithme s'appliquant à des graphes orientés (respectivement non orientés).

Nous pouvons alors définir les critères esthétiques que doit vérifier le tracé d'un graphe imbriqué G :

- i. si un nœud b est inclus dans un nœud a , le tracé de a doit contenir le tracé de b , c'est-à-dire que l'union des boîtes des deux nœuds est la boîte de a ,
- ii. pour tout nœud a de G , les nœuds de $V_{\circ \circ}^+(a)$ vérifient les critères esthétiques de l'algorithme de nom $\text{mode}_{\circ \circ}(a)$.

Les modes d'inclusion qu'il est possible d'associer aux nœuds peuvent être très variés. Ils doivent seulement correspondre à des algorithmes s'appliquant à des graphes, orientés ou non orientés selon le type de mode, dont les nœuds peuvent avoir des tailles arbitrairement grandes. En particulier, les modes peuvent être les algorithmes présentés au chapitre précédent. Pour simplifier la description de l'algorithme, nous donnons un nom générique à ces algorithmes : un appel à la fonction `placement_mode_{\circ \circ}(n)` (n : nœud) correspond simplement à l'exécution de l'algorithme de nom $\text{mode}_{\circ \circ}(n)$ sur la restriction du graphe G aux nœuds du voisinage sortant d'inclusion du nœud n .

4.3.1 Placement relatif des nœuds

Le principe de l'algorithme est le suivant. Nous supposons que les boîtes des nœuds a tels que $V_{\mathbb{G}\text{-}\circ}^+(a) = \emptyset$ sont connues. Alors pour tout nœud n , en commençant par la racine du graphe, on appelle récursivement l'algorithme sur les nœuds inclus dans n , ce qui permet de connaître leurs boîtes. Ensuite on place ces nœuds suivant le mode d'inclusion de n , puis on détermine la boîte de n comme étant l'union des boîtes des nœuds de son voisinage d'inclusion sortant.

Algorithme (Placement relatif des nœuds)

- *Données* : un graphe imbriqué G et de racine d'inclusion *racine*, dont les boîtes des nœuds n'en contenant pas d'autres sont connues,
- *Résultat* : les positions relatives à leur père d'inclusion des nœuds de G , ainsi que leur boîte.

```

PROCEDURE placement_relatif_inclus(n:nœud)
  IF  $V_{\mathbb{G}\text{-}\circ}^+(n) \neq \emptyset$  THEN
    FOR ALL  $s \in V_{\mathbb{G}\text{-}\circ}^+(n)$  DO
      placement_relatif_inclus(s);
    END_FOR;
     $N' := V_{\circ\text{-}\circ}^+(n)$ ;
     $E' := \{a R b \text{ SUCH THAT } (a,b) \in N' \times N', R \in \{\circ\text{-}\circ, \circ\text{-}\circ\}\}$ ;
    placement_mode  $\mathbb{G}\text{-}\circ(n)((N', E'))$ ;
     $\text{boite}(n) := \bigcup_n \mathbb{G}\text{-}\circ_s \text{boite}(s)$ ;
     $(x(n), y(n)) := \text{centre\_boite}(n)$ ;
    FOR ALL  $s \in V_{\mathbb{G}\text{-}\circ}^+(n)$  DO
       $x(s) := x(s) - x(n)$ ;
       $y(s) := y(s) - y(n)$ ;
    END_FOR;
  END_IF;
END_PROCEDURE;

placement_relatif_inclus(racine);

```

4.3.2 Placement absolu des nœuds

Une fois les positions des nœuds relativement à leur père d'inclusion connues, pour connaître les positions absolues et permettre le tracé du graphe, nous appliquons l'algorithme suivant :

Algorithme (Placement absolu des nœuds)

- *Données* : un graphe imbriqué G de racine *racine*, dont les positions des nœuds relativement à leur père d'inclusion sont connues,
- *Résultat* : les coordonnées absolues des nœuds du graphe G .

```

PROCEDURE coordonnees_absolues(n:nœud, dx:INTEGER, dy:INTEGER)
   $x(n) := x(n) + dx$ ;
   $y(n) := y(n) + dy$ ;
  FOR ALL  $s \in V_{\mathbb{G}\text{-}\circ}^+(n)$  DO
    coordonnees_absolues(s, x(n), y(n));
  END_FOR;
END_PROCEDURE;

coordonnees_absolues(racine, 0, 0);

```

Nous pouvons alors dessiner le graphe. Le tracé de chacun des nœuds est déterminé par leur position et la taille de leur boîte.

4.3.3 Transformations de graphes en fonction des modes d'inclusion

L'algorithme de tracé de graphes imbriqués que nous avons présenté permet d'indiquer pour chaque nœud, le nom de l'algorithme à appliquer aux nœuds de son voisinage sortant d'inclusion. Nous devons donc maintenant nous poser la question suivante : que faire si l'on demande d'appliquer un algorithme spécifique à une classe de graphes donnée, comme par exemple les arbres enracinés, à un graphe qui n'est pas élément de cette classe ? Une première réponse pourrait être tout simplement d'interdire une telle situation. Nous proposons au contraire, lorsqu'un mode particulier est demandé par l'utilisateur, de forcer son application au graphe. Pour cela, nous introduisons des fonctions de conversion réversibles d'un graphe quelconque en un graphe compatible avec le mode demandé. Nous nous limitons dans cette section à la levée des conflits issus de l'utilisation comme mode d'inclusion des algorithmes présentés au chapitre 3, ce qui nous conduit à décrire des procédures de conversion de graphes orientés en graphes orientés acycliques, et de graphes orientés en arbres enracinés.

4.3.3.1 Transformation d'un graphe orienté en graphe orienté acyclique

Nous nous intéressons tout d'abord à la conversion des graphes orientés en graphes orientés acycliques. Nous souhaitons effectuer un minimum de modifications au graphe initial, ce qui nous conduit à essayer de remplacer un nombre minimum d'arêtes par leur arête opposée. Ce problème est NP-complet [31]. Nous présentons deux heuristiques classiques [21]. Leur principe consiste à associer à chaque nœud du graphe son *niveau*, et à retourner les arêtes dont le niveau de l'origine de l'arête est supérieur ou égal à celui de sa destination. La première heuristique est basée sur un simple parcours en profondeur du graphe :

Algorithme (Suppression des cycles par parcours en profondeur)

- *Données* : un graphe orienté G
- *Résultat* : le sens de certaines arêtes du graphe G est inversé, et le graphe modifié est sans cycle.

GLOBAL $h:=0$;

PROCEDURE enleve_cycles(n :noeud)

```

  parcouru( $n$ ):=vrai;
  FOR ALL  $s \in V_{O \rightarrow O}^+(n)$  DO
    IF parcouru( $s$ )=faux THEN
      enleve_cycles( $s$ );
    END_IF;
  END_FOR;
  niveau( $a$ ):= $h$ ;
   $h:=h-1$ ;

```

END_PROCEDURE;

PROCEDURE transforme_acyclique()

```

   $h:=0$ ;
  FOR ALL  $a \in G$  DO
    parcouru( $a$ ):=FALSE;
  END_FOR;

```

```

  FOR ALL  $a \in G$  DO
    IF parcouru( $a$ )=FALSE THEN
      enleve_cycles( $a$ );
    END_IF;
  END_FOR;

```

```

FOR ALL  $e \in G$  OF KIND  $s \xrightarrow{O} t$  DO
  IF niveau( $s$ )  $\geq$  niveau( $t$ ) faire
    remplacer  $e$  par  $t \xrightarrow{s} s$ ;

```

```

END_IF;
END_FOR;
END_PROCEDURE

```

Cette première méthode a l'avantage de ne pas retourner de liens quand le graphe de départ est un arbre. La deuxième méthode présentée est basée sur l'intuition selon laquelle les nœuds ayant beaucoup de fils doivent apparaître en haut du graphe.

Algorithme (Suppression des cycles par méthode glouton)

- *Données* : un graphe orienté G ,
- *Résultat* : le sens de certaines arêtes du graphe G est inversé, et le graphe modifié est sans cycle.

```

PROCEDURE transforme_acyclique()
niveau:=nombre de nœuds dans G;
G':=G;
WHILE G'≠∅ DO
  LET v ∈ G' SUCH THAT ∂O→O+(v) = maxw∈G'(∂O→O+(w));
  niveau(a):=niveau;
  niveau:=niveau-1;
  G':=G'-v;
END_WHILE

FOR ALL lien e OF KIND s O→O t DO
  IF niveau(s)≤niveau(t) DO
    remplacer le lien e par le lien t sO→Ot s;
  END_IF;
END_FOR;
END_PROCEDURE

```

Cette méthode est en moyenne meilleure que la première.

4.3.3.2 Transformation d'un graphe orienté en arbre

Le principe de la procédure de transformation que nous proposons maintenant est inspiré des techniques de *splitting*, que nous appellerons techniques de séparation des nœuds, utilisées pour représenter certains graphes [16]. Nous commençons tout d'abord par rendre le graphe acyclique. Ensuite, si un nœud a du graphe G que nous souhaitons afficher crée un conflit avec la définition des arbres enracinés, c'est-à-dire que le degré entrant orienté d de a est strictement supérieur à un, on lève ce conflit en remplaçant le nœud a par des nœuds a_1, \dots, a_d , puis en remplaçant les d arêtes ayant le nœud a comme extrémité par d arêtes ayant chacune un des a_i comme extrémité, pour $i=1, \dots, d$. Nous dirons alors que a est *scindé* en les nœuds a_1, \dots, a_d . Ensuite, nous ajoutons un nouveau nœud temporaire r au graphe, qui a la particularité d'être supprimé dès que le graphe est tracé. En particulier, ce nœud n'est pas pris en compte pour les calculs des boîtes des nœuds lors du tracé de graphes imbriqués. Ce nœud est relié par des arêtes orientées aux nœuds du graphe G qui sont de degré entrant nul, ce qui garantit l'unicité de la racine de l'arbre. On trace ensuite le graphe modifié, ce qui entraîne que certains nœuds du graphe G seront représentés par plusieurs nœuds sur le dessin final. Les nœuds scindés peuvent alors être éventuellement reliés entre eux par des liens non orientés de couleur particulière, pour indiquer que les différents nœuds sur le dessin représentent en fait le même nœud logique. Le principe de l'algorithme de conversion de l'algorithme est résumé sur les figures 4.2 et 4.3. Les nœuds et arêtes temporaires qui ne figureront pas sur le dessin final sont représentés en grisé.

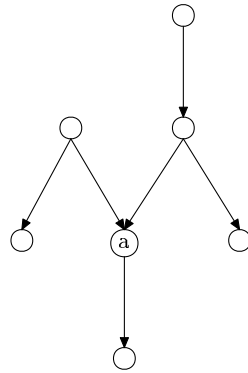


Fig. 4.2: Graphe orienté.

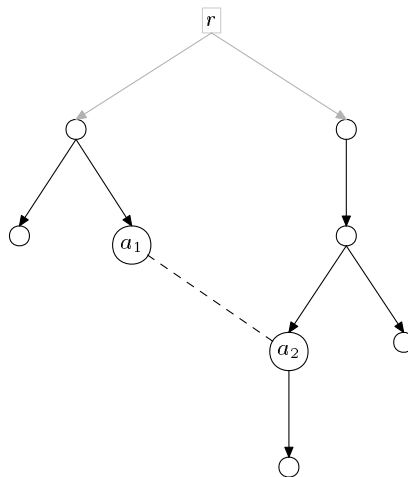


Fig. 4.3: Transformation du graphe de la figure 4.2 en un arbre enraciné.

Nous commençons par ajouter la racine du graphe. Pour cela, nous rendons le graphe acyclique à l'aide de la fonction `transforme_acyclique()`, puis nous ajoutons des liens entre un nœud racine ajouté au graphe et les nœuds du graphe dont le voisinage entrant orienté est nul.

Algorithme

```

transforme_acyclique();
racine:=nouveau_noeud();
FOR ALL  $a \in G$ ,  $a \neq \text{racine}$  SUCH THAT  $V_{\rightarrow}^-(a) = \emptyset$  DO
   $G := G \cup \{\text{racine} \rightarrow a\}$ ;
END_FOR;
```

Ensuite, nous séparons les nœuds qui créent des conflits :

```

FOR ALL  $a \in G$  DO
   $d := \text{degree}^-(a)$ ;
  IF  $d > 1$  DO
    FOR  $i$  FROM 1 TO  $d$  DO
       $a_i := \text{nouveau_noeud}()$ ;
    END_FOR
    FOR ALL  $t \in V_{\rightarrow}^+(a)$  DO
```

```

    G := G ∪ {a1 ○→ t};
    G := G - {a ○→ t};
  END_FOR;
  i := 1;
  FOR ALL s ∈ V○→-(a) DO
    G := G ∪ {s ○→ ai};
    G := G - {s ○→ a};
    i := i + 1;
  END_FOR;
  G := G - {a};
END_IF;
END_FOR;

```

4.4 Représentation des structures décomposables

Nous pouvons d'ores et déjà apporter une première réponse au problème de la représentation des structures combinatoires décomposables, en proposant une représentation standard.

4.4.1 Représentation standard

La procédure de conversion standard que nous proposons consiste à construire un graphe imbriqués en fonction de la forme de la structure à afficher, plus précisément en fonction des constructeurs utilisés pour la décrire.

```

FUNCTION conversion_standard(s: structure_decomposable) : (graphe_imbrique, noeud)
  IF s = t THEN
    ((N, E), r) := conversion_terminal(s);
  ELSE
    constructeur(s1, ..., sk) := s;
    ((N, E), r) := conversion_constructeur(s);
  END_IF;
  RETURN ((N, E), r);
END_FUNCTION;

```

Nous donnons maintenant les fonctions de conversions des terminaux ou des constructeurs de structures décomposables.

4.4.1.1 Terminaux

Nous supposons que nous disposons d'une fonction `nouveau_noeud` qui permet d'ajouter un nœud au graphe dont les attributs graphiques, comme la couleur ou le label du nœud, sont déterminés d'après le nom ou l'étiquette des terminaux de la structure. La fonction de conversion s'écrit alors :

```

FUNCTION conversion_terminal(s: structure_decomposable) : (graphe_imbrique, noeud)
  r := nouveau_noeud(s);
  RETURN (({r}, {}), r);
END_FUNCTION;

```

4.4.1.2 Constructeurs *set*, *sequence* et *cycle*

Dans le cas des constructeurs *set*, *cycle* et *sequence*, nous construisons un graphe imbriqué, formé d'une racine dans laquelle sont incluses les racines des graphes imbriqués obtenus en appelant récursivement l'algorithme sur les composantes de la structure. Les fonctions de conversion des constructeurs *cycle*, *set* et *sequence* sont très proches. Nous présentons uniquement le cas des constructeurs *cycle* :

```

FUNCTION conversion_cycle(s: structure_decomposable) : (graphe_imbrique, noeud)
  cycle(s1, ..., sk) := s;

```

```

r := nouveau_noeud();
N := {r};
E := {};
FOR i FROM 1 TO k DO
  ((Ni, Ei), ri) := conversion_standard(si);
  N := N ∪ Ni;
  E := E ∪ Ei ∪ {r ⊖ ri};
END_FOR;

```

```

FOR i FROM 1 TO k - 1 DO
  E := E ∪ {si ⊖ si+1};
END_FOR
E := E ∪ {sk ⊖ s1};

```

```

mode ⊖(r) := trace_graphe;
RETURN ((N, E), r);
END_FUNCTION;

```

La figure 4.4 illustre le principe de la conversion pour des structures de la forme $cycle(s_1, \dots, s_k)$. L'algorithme de conversion est appelé récursivement sur les structures s_1, \dots, s_k , puis les racines des graphes associés aux composantes sont incluses dans un nœud racine r :

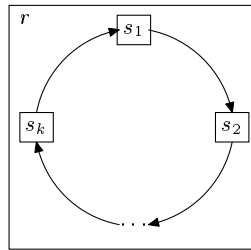


Fig. 4.4: Schéma de conversion pour les structures de la forme $cycle(s_1, \dots, s_k)$.

Le cas du constructeur *sequence* se déduit du cas *cycle* en enlevant la dernière ligne de l'encadré, celui du constructeur *set* en supprimant les lignes de l'encadré.

4.4.1.3 Constructeur *prod*

Le cas du constructeur *prod* est plus difficile que celui des autres constructeurs. Une structure de la forme $prod(s_1, s_2, \dots, s_k)$ est représentée comme un arbre enraciné de racine r_1 , où r_1 est la racine du graphe obtenu en appelant récursivement la fonction de conversion standard sur la structure s_1 . Ensuite, selon la forme des s_1, \dots, s_k , la fonction est différente. Si une structure s_i n'est pas de la forme $prod(\dots)$, nous appelons la procédure de conversion standard sur la structure s_i , et la racine r_i du graphe obtenu est alors le fils du nœud r_1 . Dans le cas où s_i est de la forme $prod(s_{i,1}, s_{i,2}, \dots, s_{i,l_i})$, la racine du graphe obtenu en appelant la fonction de conversion standard sur la structure $s_{i,1}$ est alors le fils de r_1 . Ce principe de conversion, permettant de représenter les arbres, est résumé sur la figure 4.5.

```

FUNCTION conversion_prod_rec(s: structure_decomposable) : (graphe_imbrique, noeud)
  prod(s1, ..., sk) := s;
  ((N1, E1), r1) := conversion_standard(s1);
  FOR i FROM 2 TO k DO
    IF si = prod(...) THEN
      ((Ni, Ei), ri) := conversion_prod_rec(si);
    ELSE
      ((Ni, Ei), ri) := conversion_standard(si);
    END_IF;
  END_FOR;
END_FUNCTION;

```

```

N :=  $\bigcup_{i=1,\dots,k} N_i$ ;
E :=  $\bigcup_{i=1,\dots,k} E_i$ ;
FOR i FROM 2 TO k DO
  E := E  $\cup$  { $r_1 \circ \rightarrow r_i$ };
END_FOR;
RETURN ((N, E), r1);
END_FUNCTION;

FUNCTION conversion_prod(s: structure_decomposable) : graphe_imbrique
(N, E), a := conversion_prod_rec(s);
r := nouveau_noeud();
FOR ALL n  $\in$  N SUCH THAT  $V_{\mathcal{G}_O}^-(n) = \emptyset$  DO
  E := E  $\cup$  { $r \mathcal{G}_O n$ };
END_FOR;
N := N  $\cup$  {r};
mode  $\mathcal{G}_O$ (r) := trace_arbre;
RETURN (N, E);
END_FUNCTION;

```

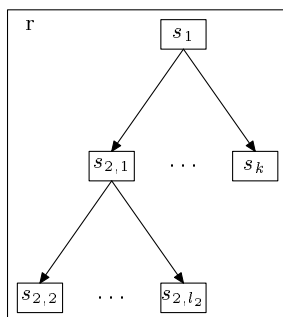


Fig. 4.5: Conversion des structures de la forme $prod(s_1, prod(s_{2,1}, \dots, s_{2,l_2}), \dots, s_k)$.

4.4.1.4 Exemple: les graphes fonctionnels

Les graphes fonctionnels que nous considérons dans cet exemple sont les graphes définis à partir d'une fonction f de la forme $f : \mathcal{E} \rightarrow \mathcal{E}$, où \mathcal{E} est un ensemble fini d'entiers. Les nœuds du graphe sont les éléments de \mathcal{E} , et deux nœuds du graphe sont reliés par une arête orientée $a \circ \rightarrow b$ si et seulement si $f(a) = b$. Ces graphes ont donc la propriété de n'avoir que des nœuds dont le degré sortant orienté est 1. Si l'on souhaite étudier l'ensemble des graphes fonctionnels tels que l'ensemble \mathcal{E} est de cardinalité n , indépendamment du choix de la fonction f associée, nous pouvons alors les considérer comme étant des ensembles de cycles d'arbres. Une spécification sous forme décomposable des graphes fonctionnels est alors de la forme :

$$\left\{ \begin{array}{l} A \rightarrow Set(B) \\ B \rightarrow Cycle(T) \\ T \rightarrow Union(\mathbf{a}, C) \\ C \rightarrow Prod(\mathbf{a}, D) \\ D \rightarrow Sequence(T) \\ \mathbf{a} \rightsquigarrow Atom \end{array} \right. \quad (4.2)$$

Remarque. Nous avons choisi dans cet exemple de considérer des arbres planaires, ce qui permet de montrer la représentation des séquences d'objets. La spécification la plus adaptée aux graphes fonctionnels utiliserait plutôt des arbres non planaires, et l'on aurait alors : $D \rightarrow Set(T)$.

On peut alors générer de tels graphes de façon aléatoire, à l'aide des algorithmes présentés au chapitre 1, puis les afficher après conversion en graphes imbriqués à l'aide des fonctions standard de conversion.

La figure 4.6 correspond au tracé de la structure étiquetée suivante, générée aléatoirement à l'aide de `combstruct`, parmi les structures du non terminal A de la spécification (4.2) ayant des ensembles de cardinalité au moins deux et des cycles de cardinalité au moins trois :

```
Set(Cycle(Prod(A[19],
    Sequence(Prod(A[6], Sequence(A[18])),
    Prod(A[8], Sequence(A[14])),
    Prod(A[10],
        Sequence(Prod(A[17], Sequence(A[4],A[1]))))),),
    A[7],
    Prod(A[13], Sequence(A[3]))),
    Cycle(Prod(A[5],
    Sequence(Prod(A[12],
    Sequence(A[11], A[16]))),
    A[2],
    Prod(A[9], Sequence(A[15]))))
```

Sur la figure 4.6, les structures construites à l'aide du constructeur `set` sont incluses dans le nœud gris foncé, celles construites à l'aide du constructeur `cycle` dans les nœuds gris clair. Les nœuds blancs forment les structures qui correspondent aux arbres du graphe fonctionnel. Les étiquettes des nœuds sont également représentées.

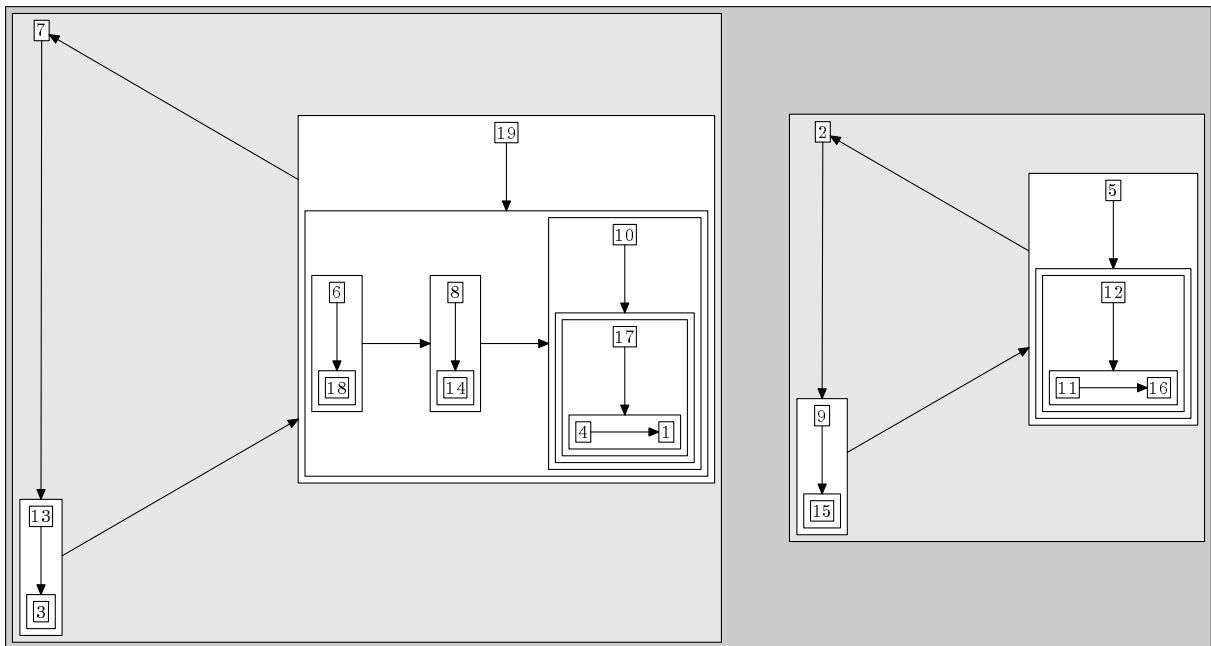


Fig. 4.6: Conversion standard d'une structure décomposable issue de la spécification (4.2).

4.4.2 Représentations particulières

La représentation standard a l'avantage de faire ressortir les propriétés de la spécification adoptée, mais elle ne correspond pas forcément à la vision naturelle que l'on se fait des structures que l'on manipule. Nous présentons maintenant sur des exemples des fonctions particulières de conversion de structures décomposables en graphes imbriqués.

4.4.2.1 Graphe série-parallèle

Les graphes série-parallèle sont définis par analogie avec les circuits électriques construits à l'aide de résistances placées en série ou en parallèle. Une spécification de ces graphes est la suivante :

$$\begin{cases} G \rightarrow \text{Union}(S, P) \\ S \rightarrow \text{Union}(\mathbf{a}, \text{Sequence}(P, \text{card} \geq 2)) \\ P \rightarrow \text{Union}(\mathbf{a}, \text{Set}(S, \text{card} \geq 2)) \\ \mathbf{a} \rightsquigarrow \text{Atom} \end{cases} \quad (4.3)$$

G est l'ensemble des graphes série-parallèle, S ceux construits en plaçant en série des parties parallèles, P ceux construits en plaçant des parties séquentielles. \mathbf{a} correspond aux résistances du circuit.

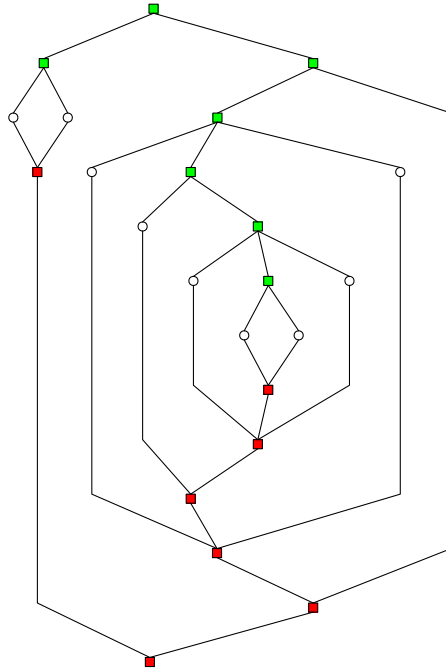


Fig. 4.7: Graphe série-parallèle.

Un exemple de graphe série-parallèle formé par dix résistances est représenté sur la figure 4.7. Les résistances sont les nœuds circulaires blancs. Le début d'une partie parallèle est indiqué par un carré gris clair, sa fin par un carré gris foncé. Les fonctions de conversion des structures issues de la spécification (4.3) en des graphes de cette forme s'écrivent :

```

FUNCTION conversion_serie(s: structure_decomposable): (graphe_imbrique, noeud, noeud)
  Sequence(s1, ..., sk):=s;
  N:={};
  E:={};
  FOR i FROM 1 TO k DO
    ((Ni, Ei), debuti, fini):=conversion_parallele(si);
    N:=N∪Ni;
    E:=E∪Ei;
    IF i≠1 THEN
      E:=E∪{fini-1 ⇨ debuti};
    END_IF;
  END_FOR;
  RETURN ((N, E), debut1, fink);
END_FUNCTION

```

```

FUNCTION conversion_parallele(s: structure_decomposable): (graphe_imbrique, noeud, noeud)
  Set(s1, ..., sk):=s;
  debut:=nouveau_noeud();
  fin:=nouveau_noeud();
  N:={debut} ∪ {fin};
  E:={};
  FOR i FROM 1 TO k DO
    (Ni, Ei, debuti, fini):=conversion_serie(si);
    N:=N ∪ Ni;
    E:=E ∪ Ei;
    E:=E ∪ {debut ◊ ◊ debuti} ∪ {fini ◊ ◊ fin};
  END_FOR;
  RETURN ((N, E), debut, fin);
END_FUNCTION

```

```

FUNCTION conversion_sp(s: structure_decomposable) : graphe_imbrique
  r:=nouveau_noeud(s);
  IF s = Sequence(...) THEN
    (N, E, debut, fin):=conversion_serie(s);
  ELSE
    (N, E, debut, fin):=conversion_parallele(s);
  END_IF
  N:=N ∪ r;
  FOR ALL n ∈ N DO
    E:=E ∪ r ◊ ◊ n
  END_FOR;
  RETURN (N, E);
END_FUNCTION

```

4.4.2.2 Graphe fonctionnel

Reprenons l'exemple des graphes fonctionnels présentés au paragraphe 4.4.1.4. Une représentation plus naturelle du graphe de la figure 4.6 pourrait être celle de la figure 4.8. Nous ne décrivons pas en détail les fonctions de conversion sur cet exemple, étant donné qu'elles sont relativement proches de celles de conversion standard. Voici cependant quelques schémas qui en illustrent le principe. L'idée est d'associer un graphe particulier en fonction des formes des composantes des structures sur les figures 4.9 et 4.10.

4.5 Algorithme de tracé de graphes composés enracinés

Nous avons présenté un algorithme de tracé de graphes imbriqués dans les sections précédentes. Cet algorithme est bien adapté à la représentation, sous la forme de graphes imbriquées, de la plupart des structures décomposables. Cependant, sur certains exemples, il peut être intéressant de s'affranchir de la condition i de la définition 4.2.7 des graphes imbriqués (point 4.2.1.2, page 109), qui demande à ce que les extrémités des arêtes aient même père d'inclusion. Prenons par exemple le cas des arbres généraux planaires, dont nous rappelons une spécification :

$$\left\{ \begin{array}{l} A \rightarrow \text{Union}(\mathbf{a}, B) \\ B \rightarrow \text{Prod}(\mathbf{a}, C) \\ C \rightarrow \text{Sequence}(A) \\ \mathbf{a} \rightsquigarrow \text{Atom} \end{array} \right. \quad (4.4)$$

Une représentation possible d'un tel graphe pourrait être celle de la figure 4.11, qui a l'avantage de mettre en avant la structure de données généralement employée pour décrire de tels graphes. Cette représentation correspond à un graphe enraciné qui n'est pas imbriqué.

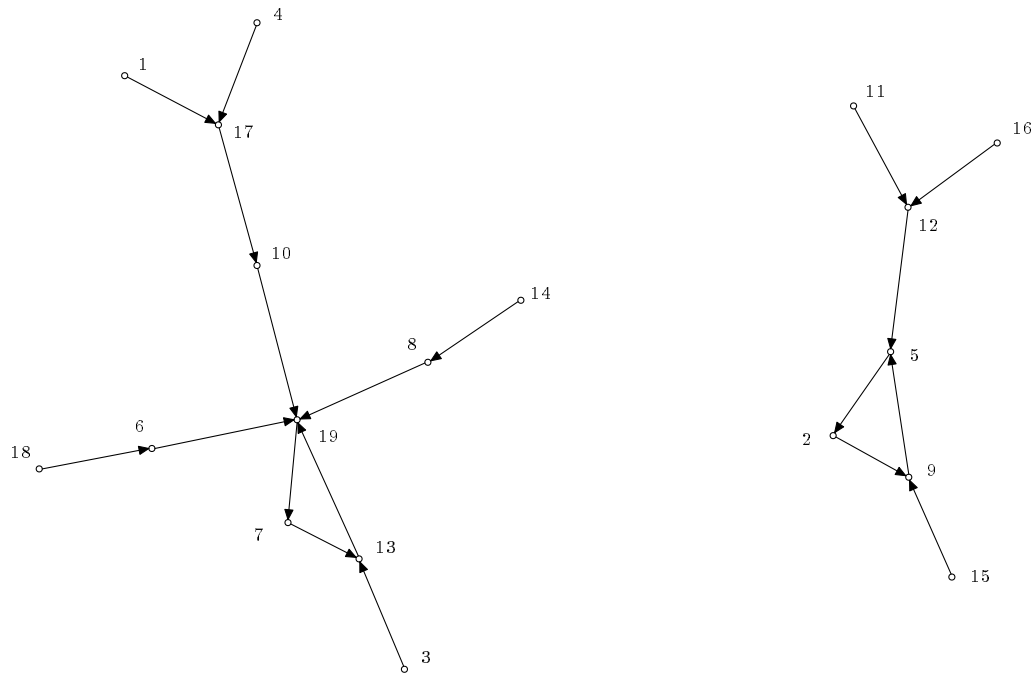


Fig. 4.8: Représentation classique du graphe fonctionnel de la figure 4.6.

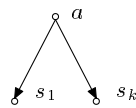


Fig. 4.9: Graphe associé aux structures de la forme $prod(a, sequence(s_1, \dots, s_k))$

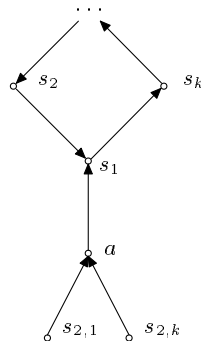


Fig. 4.10: Graphe associé aux structures de la forme $cycle(s_1, prod(a, sequence(s_{2,1}, \dots, s_{2,k})), \dots, s_k)$

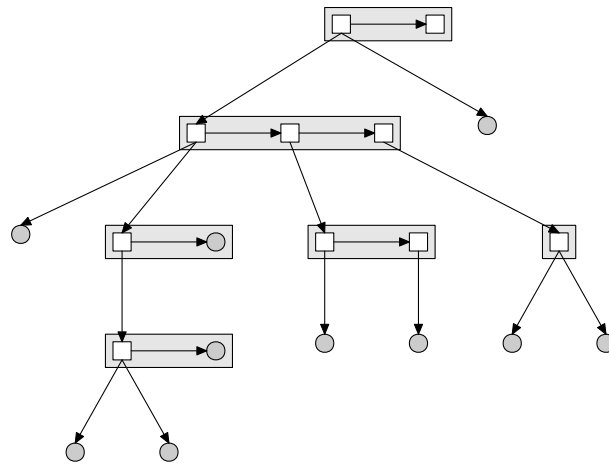


Fig. 4.11: Représentation d'une structure A de taille 20 de la spécification (4.4).

Dans le même ordre d'idée, la représentation du graphe série-parallèle de la figure 4.7 pourrait être remplacée par celle de la figure 4.12, dans laquelle les parties parallèles du graphe sont incluses dans des nœuds.

Il peut également être utile de représenter plusieurs types de liens sur le même graphe, comme sur la figure 4.13.

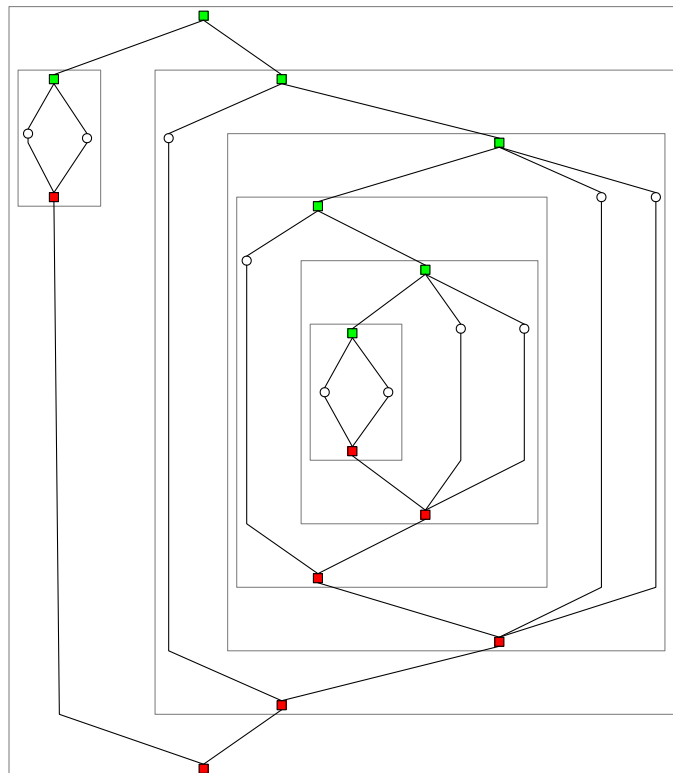


Fig. 4.12: Graphe série-parallèle de taille 10.

Ces exemples nous amènent à nous intéresser à la représentation des graphes enracinés. Nous pouvons

déjà nous poser les deux questions suivantes, qui serviront à expliquer nos choix lors de la description de l'algorithme :

- A. comment les arêtes qui ont comme extrémités des nœuds dont les pères d'inclusion sont différents doivent-elles influencer le tracé de la structure?
- B. comment les graphes dont les nœuds sont reliés à la fois par des arêtes orientées et non orientées doivent-il être dessinés?

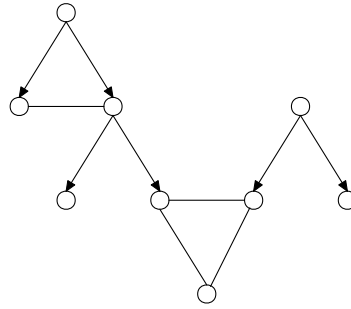


Fig. 4.13: *Grappe imbriquée ne vérifiant pas la propriété ii de la définition 4.2.7.*

4.5.1 Principe de l'algorithme

L'idée de l'algorithme de tracé proposé dans cette section est de ramener le problème général du tracé des graphes enracinés à celui plus simple du tracé des graphes imbriqués. Pour ce faire, nous apportons des modifications au graphe initial pour le transformer en un graphe imbriqué, en cherchant à ce que le placement des nœuds de ce dernier soit cohérent avec les informations contenues dans le graphe initial. Une fois les positions des nœuds connues, nous restituons ensuite les données du graphe initial. Pour cela, nous mémorisons les modifications effectuées par l'intermédiaire de liens à mémoire :

Définition 4.5.1 (Lien à mémoire)

Un *lien à mémoire* est un lien du graphe auquel on adjoint un couple de nœuds. Un lien à mémoire est noté $a \overset{c}{R} \overset{d}{b}$, où $R \in \{ \circ-\circ, \circ \rightarrow \circ, \circ \leftarrow \circ \}$.

Nous introduisons également la notion de nœud commun d'inclusion :

Définition 4.5.2 (Nœud commun d'inclusion)

Le nœud commun d'inclusion de deux nœuds a et b , noté $commun_{\circ-\circ}(a, b)$, est le nœud de niveau d'inclusion le plus élevé tel qu'il existe deux chemins d'inclusion ayant comme origine ce nœud et comme destination les nœuds a et b . On note $subs_{\circ-\circ}(a, b)$ le nœud du chemin d'inclusion d'origine le nœud $commun_{\circ-\circ}(a, b)$ et de destination a tel que le père d'inclusion de $subs_{\circ-\circ}(a, b)$ est $commun_{\circ-\circ}(a, b)$.

La transformation d'un graphe enraciné en un graphe imbriqué se fait en cherchant à vérifier successivement les propriétés i et ii de la définition 4.2.7.

4.5.1.1 Propriété i

La première propriété de la définition des graphes imbriqués demande à ce que les extrémités des arêtes des graphes enracinés aient le même père d'inclusion. L'idée que nous proposons pour vérifier cette condition consiste à remplacer chaque arête aRb , où $R \in \{ \circ \rightarrow \circ, \circ-\circ \}$, par l'arête $subs_{\circ-\circ}(a) \overset{a}{R} \overset{b}{subs_{\circ-\circ}(b)}$. Les extrémités de chaque arête ont alors même père d'inclusion par définition.

Ceci apporte une première réponse à la question A que nous nous étions posée en préambule de cette section. Deux nœuds reliés par une arête qui n'ont pas même père d'inclusion influent le tracé du graphe par l'intermédiaire du mode d'inclusion de leur nœud commun d'inclusion. Intuitivement ce choix

est raisonnable : l'idée sous-jacente est que si deux nœuds a et b doivent être placés suivant un critère esthétique donné, le fait de placer suivant ce critère deux nœuds a' et b' , a' contenant a et b' contenant b , conduira sensiblement au même placement de a et de b .

4.5.1.2 Propriété ii

La propriété ii de la définition 4.2.7 demande que la restriction du graphe aux nœuds compris dans un nœud donné ne contienne pas à la fois des arêtes orientées et des arêtes non orientées. La figure 4.13 est un exemple de graphe qui ne vérifie pas cette propriété. Une solution à ce problème consiste à ajouter de nouveaux nœuds au graphe, les *nœuds temporaires*, qui permettent de séparer le graphe en parties orientées et non orientées distinctes. Les figures 4.14 et 4.15 montrent deux solutions possibles à ce problème dans le cas du graphe de la figure 4.13. Nous choisissons un type de liens, orienté ou non orienté, et nous incluons chaque composante connexe suivant ces liens dans un nœud temporaire, puis nous changeons les extrémités des arêtes, toujours pour vérifier la propriété i de la définition des graphes imbriqués.

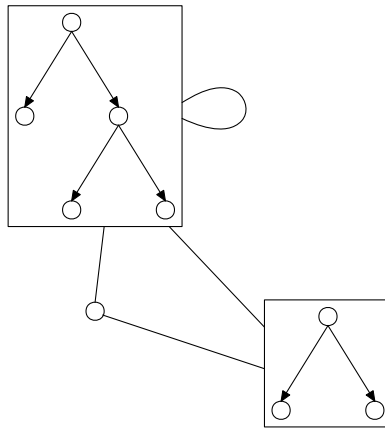


Fig. 4.14: Modification du graphe de la figure 4.13 (solution 1).

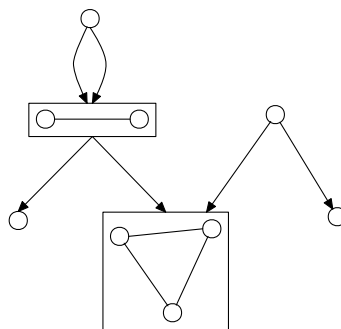


Fig. 4.15: Modification du graphe de la figure 4.13 (solution 2).

La réponse à la question B pourrait être alors : pour dessiner des nœuds ayant à la fois des liens orientés et non orientés, on considère chaque composante connexe selon les liens orientés (ou non orientés) comme un seul et même nœud, et l'on trace le graphe en tenant compte uniquement du type d'arête non orienté (ou orienté). Vouloir appliquer ces principes conduit à se poser les deux questions suivantes :

B.a. quels nœuds faut-il placer dans des nœuds temporaires ?

B.b. quel mode d'inclusion choisir pour les nœuds temporaires?

Pour permettre de répondre à ces deux questions de façon satisfaisante, nous proposons d'associer une nouvelle information à chaque nœud a du graphe, son *mode propre*, noté $mode_{\circ}(a)$. Les nœuds à inclure dans les nœuds temporaires, ainsi que les modes d'inclusion des nœuds temporaires, sont alors déterminés en fonction de cette information. De façon plus précise, nous choisissons de vérifier la propriété suivante :

$$\forall r, a \text{ tel que } r \text{ } \mathcal{G}_{\circ} a, mode_{\mathcal{G}_{\circ}}(r) = mode_{\circ}(a) \quad (4.5)$$

Ceci conduit à un algorithme plus compliqué que celui envisagé sur l'exemple du graphe 4.13 : une même composante connexe ne sera pas située forcément dans le même nœud temporaire, étant donné que les modes propres des nœuds peuvent être différents.

Remarque. Une autre solution pourrait être d'associer à chaque nœud son mode d'inclusion, et une information supplémentaire, son mode d'inclusion temporaire. Pour le graphe inclus dans un nœud, les nœuds à placer dans des nœuds temporaires seraient choisis en fonction des composantes connexes, selon le type de mode d'inclusion du nœud. Le mode d'inclusion des nœuds temporaires serait donné par le mode d'inclusion temporaire du nœud. Cette solution est plus simple à comprendre et à mettre en œuvre que la solution que nous développons dans cette section, mais elle offre moins de souplesse. Les modes d'inclusion des nœuds temporaires ayant même père d'inclusion seraient alors tous égaux, contrairement à ce qu'il est possible d'envisager avec la solution retenue. .

4.5.2 Étapes de l'algorithme

Nous détaillons maintenant les étapes de l'algorithme proposé pour le tracé des arbres enracinés. Nous cherchons tout d'abord à vérifier la propriété i de la définition 4.2.7, puis la condition ii, avant de restituer les données du graphe initial une fois appliqué l'algorithme de tracé des graphes imbriqués.

4.5.2.1 Propriété i

L'algorithme permettant de remplacer les arêtes qui n'ont pas même père d'inclusion se fait en calculant tout d'abord le niveau d'inclusion des nœuds, ce qui permet ensuite de trouver rapidement le nœud commun des extrémités de chaque arête.

4.5.2.1.1 Calcul du niveau d'inclusion. Le calcul du niveau d'inclusion de chaque nœud du graphe se fait par un simple parcours en profondeur de l'arbre d'inclusion :

Algorithme (Calcul du niveau d'inclusion)

- *Données* : un graphe enraciné $G = (N, E)$ de racine d'inclusion *racine*,
- *Résultat* : le niveau d'inclusion de chaque nœud du graphe G .

```
PROCEDURE calcul_niveau_inclusion(a:nœud, h:INTEGER)
  niveau_{\mathcal{G}_{\circ}}(a) := h;
  FOR ALL nœud b \in V_{\mathcal{G}_{\circ}}^+(a) DO
    calcul_niveau_inclusion(b, h + 1);
  END_FOR;
END_PROCEDURE;
```

```
calcul_niveau_inclusion(racine, 0);
```

La complexité de l'algorithme est en $O(n)$, où n est le nombre de nœuds du graphe, si l'on suppose que l'accès au voisinage d'un nœud est direct.

4.5.2.1.2 Remplacement des arêtes qui n'ont pas même père d'inclusion.

Algorithme (Remplacement des arêtes)

- *Données* : un graphe enraciné $G = (N, E)$ dont le niveau d'inclusion des nœuds est connu,
- *Résultat* : le graphe G est modifié : les arêtes relient des nœuds de même niveau d'inclusion, juste sous le nœud commun d'inclusion.

```

PROCEDURE remplacement_aretes()
  FOR ALL aRb, R ∈ {○→○, ○-○} DO
    a' := a;
    b' := b;
    WHILE pere_G○(a') ≠ pere_G○(b') DO
      IF niveau_G○(a') = niveau_G○(b') THEN
        a' := pere_G○(a');
        b' := pere_G○(b');
      ELSE IF niveau_G○(a') > niveau_G○(b') THEN
        a' := pere_G○(a');
      ELSE
        b' := pere_G○(b');
      END_IF;
    END_WHILE;
    remplacer le lien a R b par le lien a' R b';
  END_FOR;
END_PROCEDURE;

```

remplacement_aretes();

Le nombre d'arêtes du graphe n'est pas modifié.

Proposition 4.5.3

La complexité en temps de l'algorithme est au pire en $O((e_{\text{○→○}} + e_{\text{○-○}}) \cdot e_{\text{G○}})$. La complexité en moyenne est en $O((e_{\text{○→○}} + e_{\text{○-○}}) \cdot \log(e_{\text{G○}}))$.

Démonstration. Le nombre de tests de la boucle **WHILE** est déterminé par la hauteur de l'arbre selon les liens d'inclusion. Au pire, le nombre de tests effectués pour remplacer une arête est donc en $O(e_{\text{G○}})$, son nombre moyen en $O(\log(e_{\text{G○}}))$. \square

4.5.2.2 Propriété ii

On note $type_mode_oppose_{\text{G○}}(n)$ le type opposé du type de mode d'inclusion d'un nœud n , qui est orienté si le mode d'inclusion est non orienté et réciproquement. L'idée de l'algorithme que nous proposons pour séparer les types d'arêtes du graphe est le suivant. Soit n un nœud du graphe. Pour chaque nœud a de son voisinage sortant d'inclusion, nous appliquons récursivement l'algorithme. Ensuite si le nœud a ne vérifie pas la propriété (4.5), nous ajoutons un nœud temporaire r au voisinage sortant d'inclusion de n , puis nous incluons dans ce nœud les nœuds de la composante connexe, selon les liens $type_mode_oppose_{\text{G○}}(n)$, contenant a , à l'aide de la fonction **separe_modes_propres**. Lors de cette dernière phase, les nœuds de la composante connexe qui n'ont pas même mode propre que le nœud a sont inclus à nouveau dans un nœud temporaire, lui-même inclus dans le nœud r . L'utilisation de la variable $nb_parcours$ lors du parcours des composantes connexes, ainsi que la propriété (i), garantissent que les nœuds ne sont parcourus qu'une seule fois lors de l'exécution de l'algorithme.

Algorithme (Séparations selon les algorithmes)

- *Données* : un graphe enraciné $G = (N, E)$ dont les extrémités des arêtes ont même père d'inclusion,
- *Résultat* : le graphe G est modifié par l'ajout de nœuds temporaires, et les nœuds vérifient la propriété 4.5.


```

GLOBAL nb_parcours:=0;

PROCEDURE separe_modes_propres(n: nœud,type_arete)
  FOR ALL a∈Vtype_arete-(n),
  THEN FOR ALL a∈Vtype_arete+(n) DO
  IF parcouru(a)≠nb_parcours THEN
    parcouru(a):=nb_parcours;
    pn:=pere ⊗(n);
    IF mode ⊗(a) = mode ⊗(n) THEN
      supprimer le lien pere ⊗(a) ⊗ a;
      ajouter le lien pn ⊗ a;
    ELSE
      ajouter un nœud r au graphe G;
      temporaire(r):=TRUE;
      parcouru(r):=TRUE;
      mode ⊗(r):=mode ⊗(pn);
      mode ⊗(r):=mode ⊗(a);
      supprimer le lien pere ⊗(a) ⊗ a;
      ajouter le lien pn ⊗ r;
      ajouter le lien r ⊗ a;
    END_IF;
    separe_modes_propres(a);
  END_IF;
END_FOR;
END_PROCEDURE;

PROCEDURE separe_modes_inclus(n: nœud)
  FOR ALL s∈V⊗+ DO
    separe_modes_inclus(s);
  END_FOR
  FOR ALL s∈V⊗+ SUCH THAT temporaire(s)=FALSE DO
    nb_parcours:=nb_parcours + 1;
    IF parcouru(s)≠nb_parcours THEN
      parcouru(s):=nb_parcours;
      IF mode ⊗(s)≠mode ⊗(n) THEN
        ajouter un nœud r au graphe G;
        temporaire(r):=TRUE;
        parcouru(r):=TRUE;
        mode ⊗(r):=mode ⊗(n);
        mode ⊗(r):=mode ⊗(a);
        supprimer le lien n ⊗ a;
        ajouter le lien n ⊗ r;
        ajouter le lien r ⊗ a;
        separe_modes_propres(a,type_mode_oppose_inclusion);
      END_IF;
    END_IF;
  END_FOR;
END_PROCEDURE;

pour tout nœud n faire
  parcouru(a):=nb_parcours;
fin_pour;
nb_parcours:=nb_parcours + 1;

separe_modes_inclus(racine);

```

Proposition 4.5.4

Le coût en temps de l'algorithme de séparation des nœuds est en $O(n)$, où n est le nombre de nœuds du graphe initial. Le nombre de nœuds temporaires ajoutés est au pire en $O(n)$.

Remarque. Le graphe retourné n'est pas unique en général, et dépend de l'ordre de parcours des nœuds. Par contre, si les modes propres des nœuds des composantes connexes selon les liens orientés sont les mêmes, ainsi que ceux des nœuds des composantes connexes selon les liens non orientés, le graphe résultat est unique quel que soit le parcours des nœuds.

Nous venons de rajouter de nouveaux nœuds au graphe, les nœuds temporaires, lors de la phase de séparation des nœuds selon leurs modes. Il convient donc de remplacer à nouveau les arêtes dont les extrémités n'ont pas même père d'inclusion :

```
calcul_niveau_inclusion(racine,0);
remplacement_aretes();
```

Malgré nos efforts, il peut arriver que la propriété ii ne soit pas vérifiée après cette étape. Dans ce cas, les arêtes incompatibles avec le mode d'inclusion de leur père d'inclusion sont supprimées ou remplacées par des arêtes réflexives (Fig. 4.14).

4.5.2.3 Placement des nœuds

Le graphe que l'on souhaite afficher est maintenant un graphe imbriqué. Nous pouvons donc appliquer l'algorithme présenté en section 4.5.

```
placement_relatif_inclus(racine);
coordonnees_absolues(racine,0,0);
```

4.5.2.4 Restitution du graphe initial

Une fois appliqué l'algorithme au graphe modifié, il reste à restituer les arêtes du graphe initial, et à supprimer les nœuds temporaires.

Il convient, lors de la restitution des liens à mémoire correspondant à des arêtes, d'éviter les chevauchements entre le tracé des arêtes et celui des nœuds. La solution que nous avons retenue consiste à tracer les arêtes modifiées par des lignes brisées. Cela nécessite de calculer la distance entre la droite reliant les extrémités des nœuds et l'ensemble des nœuds du graphe inclus dans le nœud commun d'inclusion des extrémités qui ne contiennent pas par transitivité les extrémités de l'arête. Le résultat de cet algorithme est illustré sur les figures 4.16 et 4.17, mais nous ne le détaillons pas.

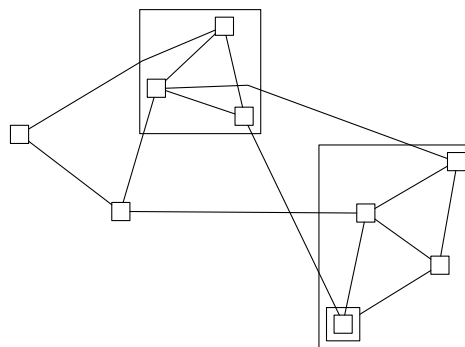


Fig. 4.16: Exemple de restitution d'arêtes modifiées.

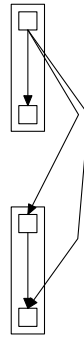


Fig. 4.17: Exemple de restitution d'arêtes orientées.

Remarque. La méthode que nous avons retenue a l'avantage de s'appliquer quelles que soient les modes d'inclusion des nœuds. Dans le cas des arêtes orientées, dont le mode du nœud commun d'inclusion des extrémités est l'algorithme de tracé d'arbres ou de graphes orientés présenté au chapitre 3, il est possible de proposer une méthode plus simple, de coût constant, qui consiste à contourner les nœuds extrémités de l'arête, comme indiqué sur la figure 4.17. L'écartement entre l'arête et le nœud est déterminé en fonction des distances entre les nœuds extrémités de l'arête.

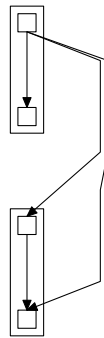


Fig. 4.18: Exemple de restitution d'arêtes orientées.

4.5.2.5 Enlèvement des nœuds temporaires

La dernière étape consiste enfin à retirer les nœuds temporaires du graphe.

Algorithme (Enlèvement des nœuds temporaires)

- *Données* : un graphe enraciné $G = (N, E)$ de racine d'inclusion r ,
- *Résultat* : les nœuds temporaires du graphe G sont supprimés.

```

FOR ALL  $a \in N$ ,  $a \neq r$  SUCH THAT  $temporaire(a) = \text{TRUE}$  DO
  FOR ALL  $s \in V_{\mathbb{G} \circ}^+(a)$  DO
    supprimer le lien  $r \mathbb{G} \circ s$ ;
    ajouter le lien  $perc_{\mathbb{G} \circ}(r) \mathbb{G} \circ s$ ;
  END_FOR;
  supprimer le nœud  $a$ ;
END_FOR;

```

4.6 Algorithme de tracé de graphes composés quelconques

Pour représenter les graphes composés quelconques, nous proposons de convertir ces graphes en graphes enracinés. Pour cela, nous pouvons utiliser le principe de séparation des nœuds utilisé pour la conversion des graphes orientés en arbres enracinés peut être utilisé pour permettre de représenter tous les graphes composés. L'algorithme est exactement le même que celui utilisé pour la conversion des graphes orientés en arbres enracinés, si ce n'est que l'on considère les liens d'inclusion au lieu des liens orientés. Ceci nous permet au final de disposer d'un algorithme de tracé de graphes composés quelconques.

4.7 Visualisation de graphes

Nous discutons maintenant quelques applications des graphes composés pour la représentation de structures décomposable et de graphes en général. Nous regardons tout d'abord quelques opérations concernant l'affichage des graphes composés, en abordant le problème du masquage et de la réduction de parties du graphe. Ensuite nous regardons comment diminuer le nombre d'arêtes d'un graphe en ajoutant un contenu sémantique aux liens d'inclusion.

4.7.1 Sélection visuelle de parties d'un graphe

Pour représenter des graphes de grandes tailles et permettre de prendre connaissance des informations situées aux nœuds, plusieurs techniques sont disponibles. La plus simple consiste à ne considérer qu'une partie du graphe par un effet de loupe (zoom). L'inconvénient de cette méthode est qu'elle fait perdre le contexte de la partie du graphe considérée. Pour répondre à ce problème, de nouvelles techniques ont été envisagées, comme les représentations *fish-eye view* [48], ou encore les regroupements de plusieurs nœuds du graphe en un seul [50].

Nous proposons d'utiliser la structure de graphe composé pour permettre de mettre en avant certaines parties d'un graphe. Pour cela nous choisissons de permettre de masquer entièrement un sous graphe inclus dans un nœud, ou encore d'en réduire la taille à l'affichage. Prenons par exemple le graphe de la figure 4.19. Les parties représentées en grisé sont celles dont nous souhaitons réduire l'importance dans le dessin final. Après application de l'algorithme de réduction, nous obtenons alors le graphe de la figure 4.20. L'avantage de cette méthode est que l'on conserve la vision globale du graphe, sans utiliser de déformations peu naturelles.

4.7.2 Réduction du nombre d'arêtes d'un graphe

En ajoutant un contenu sémantique à la relation d'inclusion, il est possible de diminuer de façon importante le nombre d'arêtes du graphe. L'idée consiste à considérer qu'un graphe G dont un nœud a est relié par une arête non orientée à un nœud b , avec les nœuds $s_1, \dots, s_k \in V_{\mathcal{G}_O}^+(b)$, est équivalent au graphe G' ne contenant pas le nœud b et tel que le nœud a est relié aux nœuds s_1, \dots, s_k par une arête non orientée. Par exemple, les deux graphes des figures 4.21 et 4.22 sont équivalents.

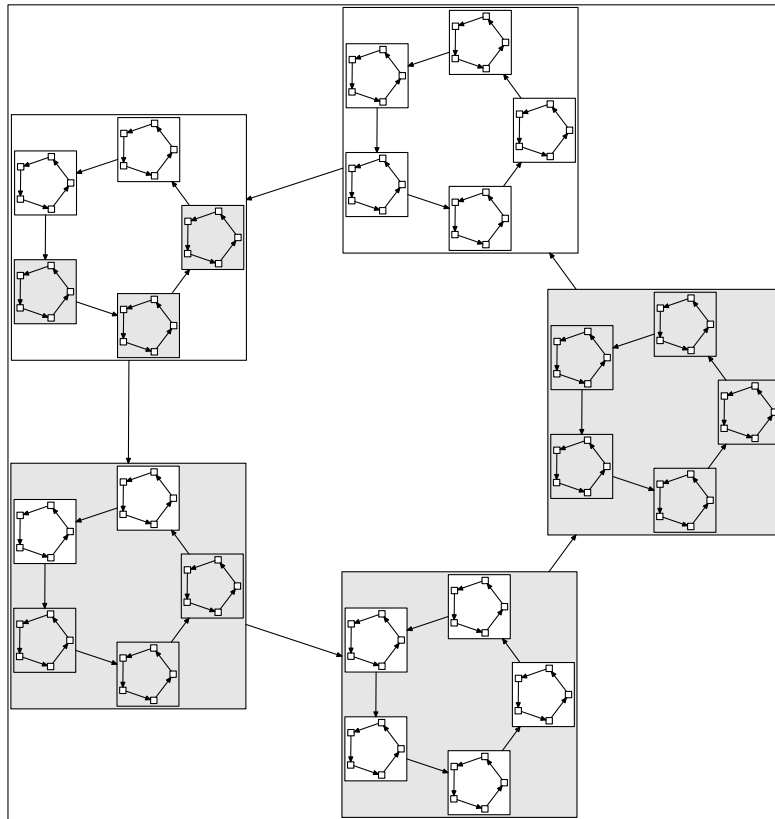


Fig. 4.19: Graphe avant réduction.

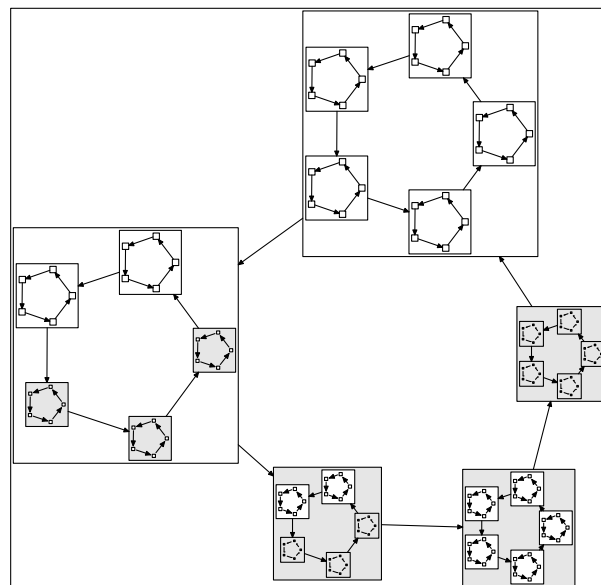
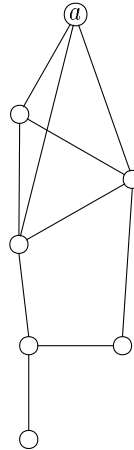
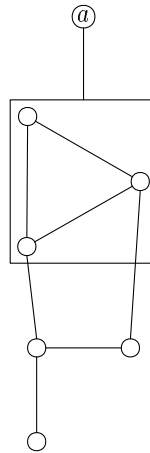


Fig. 4.20: Graphe de la figure 4.19, après réduction des parties grisées.

Fig. 4.21: *Grappe non orienté.*Fig. 4.22: *Grappe équivalent au grappe de la figure 4.21*

Il est possible d'appliquer le même genre de règles dans le cas d'arêtes orientées. Le tableau de la figure 4.23 est un exemple de manipulation d'un graphe suivant cette sémantique. Le graphe de départ est formé de neuf noeuds, représentant les entiers de 1 à n , reliés par des arêtes orientées si un entier en divise un autre.

4.8 Conclusion

Nous avons présenté dans ce chapitre un algorithme de tracé de graphes composés, basé sur la conversion du graphe à afficher en un graphe imbriqué. Cet algorithme offre la possibilité de faire cohabiter plusieurs algorithmes de tracé de classes de graphes particulières sur le même dessin, avec comme seule contrainte le fait que ces algorithmes doivent permettre le tracé de graphes dont les noeuds sont de tailles arbitrairement grandes. Nous avons également proposé une méthode de conversion standard des structures décomposables en graphes composés, dont l'intérêt est qu'elle fait ressortir graphiquement les propriétés combinatoires de la structure. Ceci nous permet de disposer d'une méthode pour la représentation des structures décomposables. L'intérêt de notre approche est qu'elle offre la possibilité à l'utilisateur de créer ses propres fonctions de conversion de structures décomposables en graphes composés, comme nous

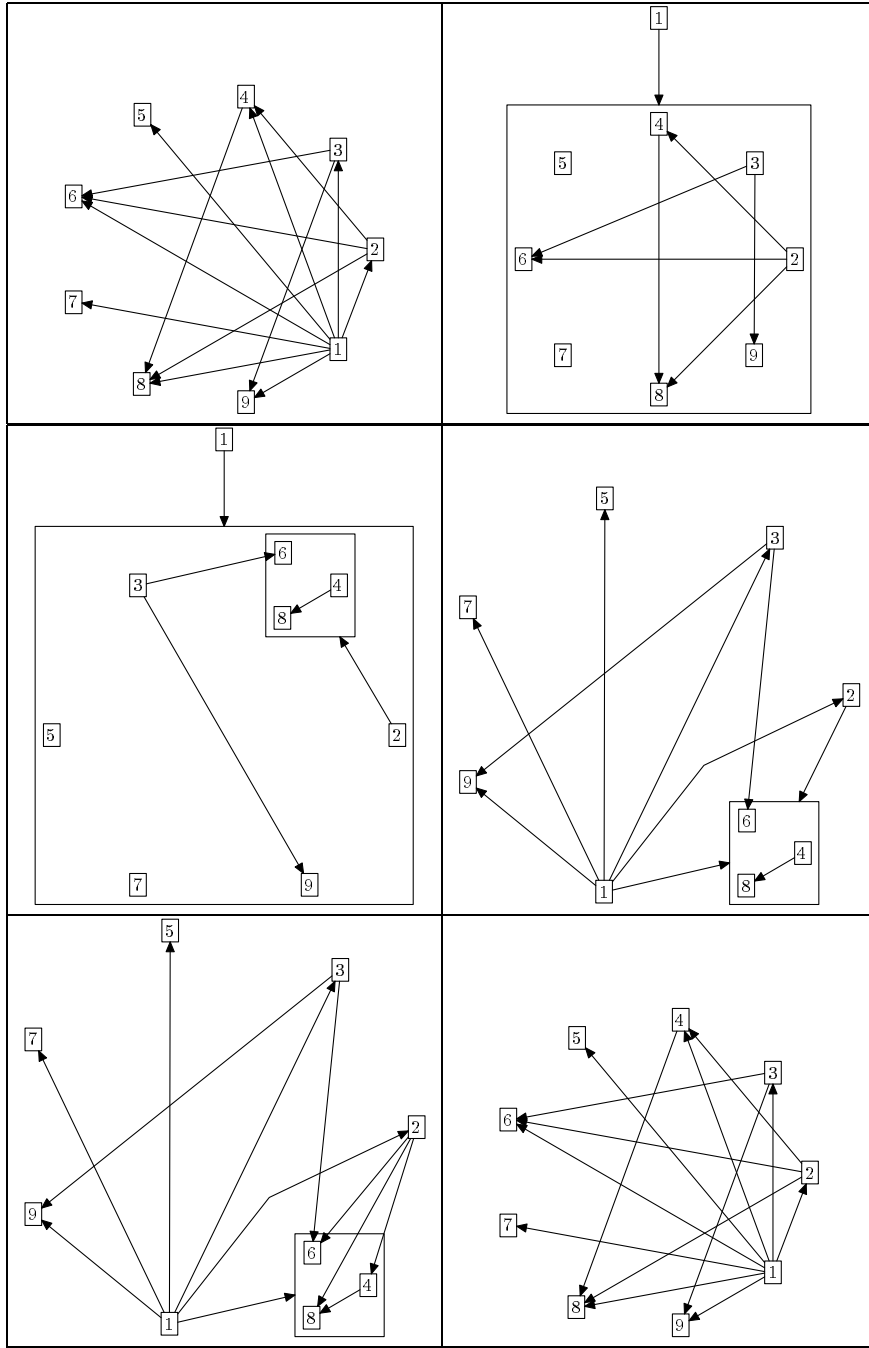


Fig. 4.23: Suite de transformations équivalentes.

l'avons montré sur des exemples en section 4.4.2. Ceci nous permet de réaliser un compromis entre la création d'algorithmes nouveaux pour chaque classe de structures manipulée et l'utilisation systématique d'une représentation standard pour toutes les classes. Enfin, nous avons évoqué en fin de chapitre que l'utilisation des graphes composés pour la représentation des structures décomposables était bien adaptée à la représentation de structures de grandes tailles, en présentant un mécanisme permettant de porter son attention sur des parties précises du graphe, tout en conservant la vision globale de la structure. En conclusion, nous disposons maintenant de la possibilité de représenter automatiquement, avec une certaine souplesse, des structures générées aléatoirement à l'aide des techniques présentées au chapitre 1 ou de façon systématique à l'aide de celles du chapitre 2. Pour cela, nous utilisons les algorithmes que nous avons présenté au chapitre 3 comme modes de tracé de l'algorithme général. La concrétisation de ce travail est la réalisation du logiciel **CGraph**, que nous présentons dans le chapitre suivant, qui permet l'utilisation conjointe de ces techniques pour former un outil intégré de génération et de tracé de structures décomposables.

Chapitre 5

Réalisations logicielles

LES ASPECTS PRINCIPAUX de deux réalisations logicielles réalisées dans le domaine de la représentation de graphes sont abordés dans ce chapitre. Le premier logiciel, appelé **Padnon**, concerne l'affichage de graphes orientés ou non orientés en trois dimensions. Il est conçu pour permettre de tracer de façon incrémentale un graphe, et est pour cette raison bien adapté à l'animation d'algorithmes. Le second logiciel, appelé **CGraph**, permet la manipulation interactive et le tracé automatique de graphes composés. Il implante les algorithmes de tracé de graphes composés présentés au chapitre 4. Ces deux programmes peuvent être reliés à des programmes extérieurs, comme par exemple le système **Maple**, pour former des outils intégrés de génération et de visualisation de structures décomposables.

5.1 Padnon

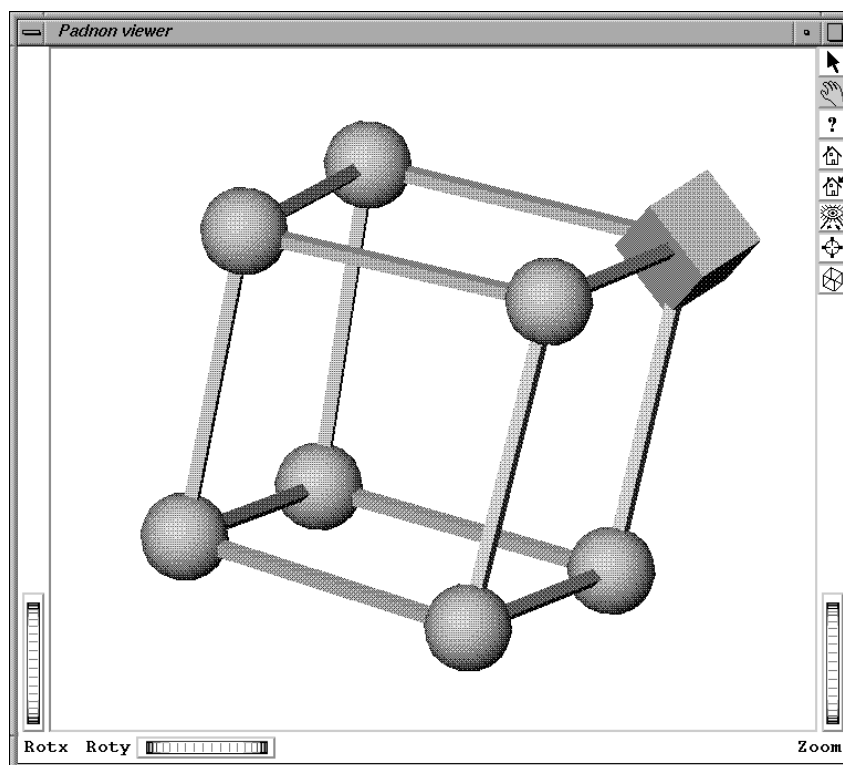


Fig. 5.1: Padnon : fenêtre de visualisation du graphe

Le programme **Padnon** [3] permet de représenter des graphes orientés ou non orientés en trois dimensions. Il implante les algorithmes de tracé par « ressorts » de graphes orientés et non orientés présentés au chapitre 3. L'interface du programme permet de manipuler de façon intuitive le graphe représenté, par effet de rotation ou de déplacement dans la scène en trois dimensions. **Padnon** peut être relié facilement à d'autres programmes pour permettre d'afficher un graphe construit automatiquement. Les algorithmes de tracé permettent, après une petite modification du graphe, d'obtenir une nouvelle représentation qui tienne compte du tracé dont on disposait avant modification. Ceci permet de conserver la représentation mentale que l'on s'est faite du graphe au cours des étapes précédentes. Cette propriété rend le programme **Padnon** particulièrement bien adapté à la représentation dynamique des graphes. Il peut servir comme interface pour la représentation de phénomènes dynamiques sur les graphes, ou encore servir à l'animation d'algorithmes. **Padnon** est par exemple utilisé pour afficher l'automate dynamique sous-jacent d'un programme de recherche de motifs dans un texte [4, 38].

Le programme **Padnon** est écrit en C++. Il s'appuie sur la bibliothèque **Inventor** [60] pour la représentation en trois dimensions des graphes, et la bibliothèque **Motif** [35] pour l'interface utilisateur du programme.

5.2 CGraph

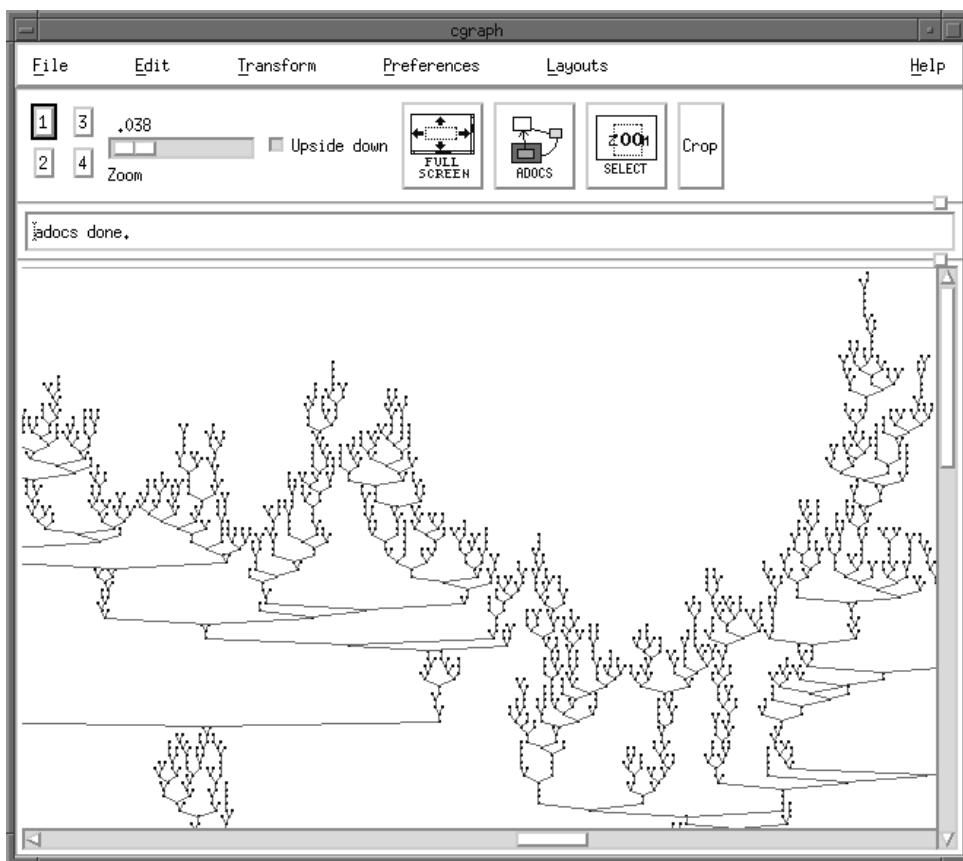


Fig. 5.2: CGraph: fenêtre principale

Le programme **CGraph** est un logiciel permettant l'édition et le tracé automatique de graphes composés. Il met en œuvre l'algorithme de tracé de graphes composés quelconques présenté au chapitre 4. Les modes d'inclusion des nœuds utilisés par cet algorithme sont l'algorithme de tracé d'arbres généraux,

l'algorithme **FR** de tracé de graphes non orientés et l'algorithme classique de tracé de graphes orientés présentés au chapitre 3, ainsi qu'un algorithme permettant de placer les nœuds d'un graphe sur un cercle. Ce programme implante également l'algorithme **FRCC** que nous avons proposé au chapitre 3, ainsi qu'un algorithme inspiré de l'algorithme **DH** utilisant les techniques d'optimisation de recuit simulé [11]. Il est également possible d'appliquer l'algorithme de tracé de graphes planaires proposés dans la bibliothèque **LEDA** [39].

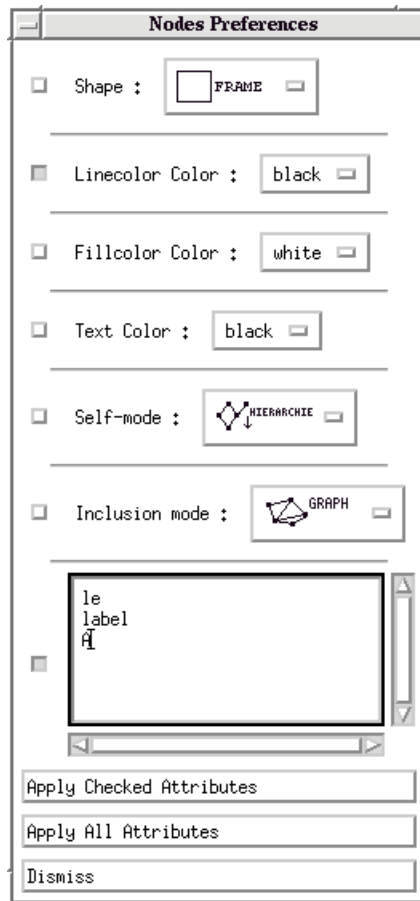


Fig. 5.3: Choix des préférences graphiques et des modes de nœuds du graphe.

L'interface utilisateur du programme (Fig. 5.2) est conçu permet une utilisation concrète des principes exposés dans cette thèse. L'éditeur possède l'ensemble des fonctionnalités que l'on est en droit d'attendre d'un tel outil. Il est possible d'ajouter, de supprimer, de déplacer ou encore de dupliquer des ensembles de nœuds ou d'arêtes. Il est possible de modifier tout ou partie des attributs graphiques d'un ensemble de nœuds ou d'arêtes (Fig. 5.3), sauvegarder les dessins obtenus dans un format propre à **CGraph** ainsi qu'aux formats Postscript ou **metapost**. Il est enfin possible de relier **CGraph** à d'autres programmes, ce qui permet d'afficher des graphes construits automatiquement par des programmes extérieurs. Il est en particulier possible de relier **CGraph** directement à **Maple**, générer une structure à l'aide de **combstruct**, puis d'utiliser le module **convert2cgraph** de conversion de structures décomposables en graphe composé obtenues à l'aide de **combstruct**, dont le principe a été exposé au chapitre 4.

Le programme **CGraph** est écrit en **C++**. Il s'appuie sur la bibliothèque **LEDA** [39] pour la gestion des structures de données élémentaires et la représentation de graphes planaires, la bibliothèque **lp_solve** de résolution de contraintes, et la bibliothèque **Motif** [35] pour l'interface utilisateur du programme.

5.3 Conclusion

Nous avons présenté deux programmes de tracé de graphe. Le premier, **Padnon**, permet le tracé en trois dimensions de graphes orientés et non orientés, et est spécialement conçu pour la représentation de structures dynamiques. Le second, **CGraph**, permet le tracé de graphes composés selon l'algorithme décrit au chapitre 4. S'il est conçu initialement pour la représentation de structures décomposables, **CGraph** n'est cependant pas limité à ce seul domaine, et trouve des applications dans la représentation de graphes en général. Ces deux programmes permettent la communication avec d'autres programmes, en particulier ceux mettant en œuvre les techniques de génération de structures décomposables présentées aux chapitres 1 et 2. Ils permettent donc au final de former des outils complets de génération de structures et de visualisation, tout en restant ouverts à d'autres utilisations ultérieures.

Bibliographie

- [1] F. Bergeron, G. Labelle et P. Leroux. – *Théorie des espèces et combinatoire des structures arborescentes*. – LaCIM, 1994. Publication n° 19.
- [2] F. Bertault. – Adocs: a drawing system for generic combinatorial structures. *In: Symposium on Graph Drawing*, éd. par F. J. Brandenburg, *Lecture Notes in Computer Science*, volume 1027, pp. 24–27. – Passau (Allemagne), 1995. Springer Verlag.
- [3] F. Bertault. – *Padnon: un logiciel de tracé automatique de graphes en trois dimensions*. – Rapport de recherche n° RR-3130, INRIA-Lorraine, 1997.
- [4] F. Bertault et G. Kucherov. – Visualisation of dynamic automata using padnon. *In: International Workshop on Implementing Automata, Lecture Notes in Computer Science*. – London (Canada), septembre 1997. Springer Verlag. Accepté.
- [5] A. Bloesch. – Aesthetic layout of generalized trees. *Software-Practice and Experience*, vol. 23, n° 8, 1993, pp. 817–827.
- [6] E. Catalan. – Propositions et questions diverses. *Bull. Soc. Math. de France*, vol. 16, 1887–88, pp. 128–129.
- [7] V. Chvatal. – *Linear Programming*. – W. H. Freeman and Company, New York, 1983.
- [8] E. G. Coffmann et R. L. Graham. – Optimal scheduling for two processor systems. *Acta Informatica*, no1, 1972, pp. 200–213.
- [9] M. Cosnard et D. Trystram. – *Algorithmes et architectures parallèles*. – InterEditions, 1993.
- [10] W. Creyaufmüller. – *Primzahlfamilien*. – Aachen, W. Creyaufmüller, 1995, première édition.
- [11] R. Davidson et D. Harel. – *Drawing Graphs Nicely Using Simulated Annealing*. – Rapport technique n° CS 89–13, Departement of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, 1989.
- [12] M. Delest, J. M. Fédou, G. Mélançon et N. Rouillon. – Faire des mathématiques avec CalICo. *Bulletin AMQ Association mathématique du Québec*, mai 1994, pp. 30–37.
- [13] G. Di Battista, P. D. Eades, R. Tamassia et I. G. Tollis. – Algorithms for drawing graphs: an annotated bibliography,. *Computational Geometry: theory and applications*, vol. 4, n° 5, 1994, pp. 235–282.
- [14] I. Dutour. – *Grammaires d'objets: énumération, bijections et génération aléatoire*. – Thèse de doctorat, Université Bordeaux I, 1996.
- [15] P. D. Eades. – A heuristic for graph drawing. *Congressus Numerantium*, vol. 42, 1984, pp. 149–160.
- [16] P. D. Eades et C. F. X. de Mendonça N. – Vertex splitting and tension-free layout. *In: Symposium on Graph Drawing*, éd. par F. J. Brandenburg, *Lecture Notes in Computer Science*, volume 1027, pp. 202–211. – Passau (Allemagne), 1995. Springer Verlag.

-
- [17] P. D. Eades et Q.-W. Feng. – Multilevel visualisation of clustered graphs. *In: Symposium on graph drawing*, éd. par S. North, volume 1190, pp. 101–111. – Berkeley (États-Unis), 1996. Springer Verlag.
- [18] P. D. Eades, Q.-W. Feng et X. Lin. – Straight-line drawing algorithms for hierarchical graphs and clustered graphs. *In: Symposium on graph drawing*, éd. par S. North, volume 1190, pp. 113–127. – Berkeley (États-Unis), 1996. Springer Verlag.
- [19] P. D. Eades, T. Lin et X. Lin. – Two tree drawing conventions. *International Journal of Computational Geometry and Applications*, vol. 3, n° 2, 1993, pp. 133 – 153.
- [20] P. D. Eades, J. Marks et S. North. – Graph-drawing contest report. *In: Symposium on Graph Drawing*, éd. par S. North, *Lecture Notes in Computer Science*, volume 1190. – Berkeley (États-Unis), 1997. Springer Verlag.
- [21] P. D. Eades et K. Sugiyama. – How to draw a directed graph. *Journal of Information Processing*, vol. 13, n° 4, 1990, pp. 424–437.
- [22] I. Fary. – On straight line representing of planar graphs. *Acta. Sci. Math.*, vol. 11, 1948, pp. 229–233.
- [23] P. Flajolet. – Elements of a general theory of combinatorial structures. *In: Fundamentals of Computation Theory*, éd. par Lothar Budach, *Lecture Notes in Computer Science*, volume 199, pp. 112–127. – Cottbus, GDR, (Invited Lecture), septembre 1985. Springer Verlag.
- [24] P. Flajolet, B. Salvy et P. Zimmermann. – Lambda-Upsilon-Omega: An assistant algorithms analyzer. *In: Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, éd. par T. Mora, *Lecture Notes in Computer Science*, volume 357, pp. 201–212. – Rome, Juillet 1988, 1989.
- [25] P. Flajolet et M. Soria. – The cycle construction. *SIAM Journal on Discrete Mathematics*, vol. 4, n° 1, février 1991, pp. 58–60.
- [26] P. Flajolet, P. Zimmermann et B. Van Cutsem. – A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, vol. 132, n° 1-2, 1994, pp. 1–35.
- [27] T. Fruchterman et E. Reingold. – Graph drawing by force-directed placement. *Software-Practice and Experience*, vol. 21, n° 11, 1991, pp. 1129–1164.
- [28] H. N. Gabow. – An almost-linear algorithm for two-processor scheduling. *Journal of the ACM*, vol. 29, n° 3, 1982, pp. 766–780.
- [29] E. R. Gansner, S. C. North et K. P. Vo. – Dag - a program that draws directed graphs. *Software Practice and Experience*, vol. 18, n° 11, 1988, pp. 1047–1062.
- [30] M. R. Garey et D. S. Johnson. – Two-processor scheduling with start-times and deadlines. *SIAM Journal on computing*, no6, 1977, pp. 416–426.
- [31] M. R. Garey et D. S. Johnson. – Crossing number is NP-complete. *SIAM Journal of Algebraic and Discrete Methods*, vol. 4, n° 3, 1983, pp. 312–316.
- [32] M. Y. Gonzales. – Deterministic processor scheduling. *ACM Computing Surveys*, vol. 9, n° 3, 1977, pp. 173–204.
- [33] T. Granlund. – The GNU Multiple Precision Arithmetic Library. édition 2.0, 1996. Disponible à l'adresse <http://www.nada.kth.se/~tege/gmp/>.
- [34] F. Harary. – *Graph Theory*. – Addison-Wesley, 1969.
- [35] D. Heller et M. Paula. – *Motif programming manual: for OSF/Motif release 1.2*. – O'Reilly and Associates, 1994.
- [36] A. Joyal. – Une théorie combinatoire des séries formelles. *Advances in Mathematics*, vol. 42, n° 1, 1981, pp. 1–82.

-
- [37] T. Kamada et S. Kawai. – An algorithm for drawing general undirected graphs. *Information Processing Letters*, vol. 31, 1989, pp. 7–15.
- [38] G. Kucherov et M. Rusinowitch. – Matching a set of strings with variable length don't cares. *Theoretical Computer Science*, vol. 178, 1997, pp. 129–154.
- [39] K. Mehlhorn et S. Naher. – LEDA, a Platform for Combinatorial and Geometric Computing. *Communications of the ACM*, vol. 38, n° 1, 1995, pp. 96–102.
- [40] G. L. Nemhauser et L. A. Wolsey. – *Integer and Combinatorial Optimization*. – W. H. Freeman and Company, New York, 1988.
- [41] A. Nijenhuis et H. S. Wilf. – *Combinatorial Algorithms*. – Academic Press, 1978, seconde édition.
- [42] J. M. Pallo. – Enumerating, ranking and unranking binary trees. *Computational Journal*, vol. 29, 1986, pp. 171–175.
- [43] J. M. Pallo et R. Racca. – A note on generating binary trees in A-order and B-order. *International Journal Computer Mathematics*, vol. 18, n° 1, 1985, pp. 27–39.
- [44] E. M. Reingold et J. S. Tilford. – Tidier drawings of trees. *IEEE Transactions on Software Engineering*, vol. SE-7, n° 2, mars 1981, pp. 223–228.
- [45] N. Rouillon. – *Calcul et Image en Combinatoire*. – Thèse de doctorat, Université Bordeaux I, 1991.
- [46] B. Salvy. – *Asymptotique automatique et fonctions génératrices*. – Thèse de doctorat, École Polytechnique, 1991.
- [47] B. Salvy et P. Zimmermann. – *Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable*. – Rapport technique n° 143, INRIA, 1992.
- [48] M. Sarkar et M. H. Brown. – Graphical fisheye views. *Communications of the ACM*, vol. 37, n° 12, 1994, pp. 73–84.
- [49] N. J. A. Sloane et S. Plouffe. – *The encyclopedia of integer sequences*. – Academic Press, 1995.
- [50] K. Sugiyama et K. Misue. – Visualisation of structural information: automatic drawing of compound digraphs. *IEEE Trans. SMC*, vol. 4, n° 21, 1991, pp. 876–893.
- [51] K. Sugiyama et K. Misue. – A generic compound graph visualizer/manipulator: D-ABDUCTOR. In: *Symposium on Graph Drawing*, éd. par F. J. Brandenburg, *Lecture Notes in Computer Science*, volume 1027, pp. 500–503. – Passau (Allemagne), 1995. Springer Verlag.
- [52] K. Sugiyama, S. Tagawa et M. Toda. – Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, vol. 11, n° 2, 1981, pp. 109–125.
- [53] K. J. Supowit et E. M. Reingold. – The complexity of drawing trees nicely. *Acta Informatica*, vol. 18, 1983, pp. 377–392.
- [54] W. T. Tutte. – Convex representation of graphs. *Proceedings of the London Mathematical Society*, vol. 10, 1960, pp. 304–320.
- [55] W. T. Tutte. – How to draw a graph. *Proceedings of the London Mathematical Society*, vol. 3, n° 13, 1963, pp. 743–768.
- [56] V. Vajnovszki. – *Algorithmes de génération de rang et de rang inverse pour certaines classes d'arbres*. – Thèse de doctorat, Université de Dijon, 1994.
- [57] J. Vaucher. – Pretty-printing of trees. *Software Practice and Experiences*, vol. 10, 1980, pp. 553–561.
- [58] J. S. Vitter et P. Flajolet. – Analysis of algorithms and data structures. In: *Handbook of Theoretical Computer Science*, éd. par J. van Leeuwen, chap. 9, pp. 431–524. – North Holland, 1990.

-
- [59] K. Wagner. – Bemerkungen zum Vierfarbenproblem. *Jber. Deutsch. Math.-Verein*, vol. 46, 1936, pp. 26–32.
- [60] J. Wenecke. – *The Inventor Mentor*. – Addison Wesley, 1994.
- [61] C. Whetherel et A. Shannon. – Tidy drawings of trees. *IEEE Transactions on software engineering*, vol. SE, n° 5, 1979, pp. 514–520.
- [62] P. Zimmermann. – *Séries génératrices et analyse automatique d'algorithmes*. – Thèse de doctorat, École Polytechnique, 1991.
- [63] P. Zimmermann. – Gaïa: a package for the random generation of combinatorial structures. *Maple-Tech*, vol. 1, n° 1, 1994, pp. 38–46.

Index

- arbre
 - contour, 70
 - de rang inverse, 56
 - feuille, 56
 - nœud interne, 56
 - enraciné, 68
 - enraciné planaire, 68
 - feuille, 14, 68
 - fil, 14
 - général, 14
 - hauteur, 68
 - Motzkin, 15, 22
 - nœud interne, 14
 - planaire, 14
 - racine, 14, 68
 - sous-arbre, 68
 - symétrique, 68
- arête
 - destination, 68
 - extrémité, 67, 68
 - non orientée, 67, 106
 - orientée, 68
 - origine, 68
- boîte, 109
- Boustrophédon, 26, 28, 32
- Ccombstruct**, 23
- chaîne, 67
 - extrémité, 67
 - longueur, 67
- chemin, 35
 - aire, 35
 - critique, 97
 - d'inclusion, 108
 - de Dyck, 35
 - destination, 68, 108
 - extrémités, 35
 - hauteur, 35
 - longueur, 35, 68
 - orienté, 68
 - origine, 68, 108
- classe de spécification
 - $\hat{\Omega}$, 13
 - Ω , 13
- coefficient multinomial, 49
- combstruct**, 22
- composante
 - directe, 11
- constructeur, 1
 - admissible, 13
 - arité, 10
 - composante directe, 10
 - de structures, 10
 - opérande, 10
- construction, 9
- contour
 - d'un arbre enraciné, 70
 - extérieur, 70
 - intérieur, 70
- cycle
 - d'inclusion, 108
 - orienté, 68
- degré
 - entrant d'inclusion, 108
 - entrant orienté, 68
 - non orienté, 67
 - sortant d'inclusion, 108
 - sortant orienté, 68
- destination, 106, 108
- distance
 - horizontale, 73
- étiquette, 12
- extrémités, 106, 108
- famille aliquote, 80
- fil d'inclusion, 108
- fish-eye view*, 129
- fonction de rang, 41
- fonction énergie J , 83
- fonctions de calcul, 31
- forme, 43
 - énumération, 43
 - énumération valide, 43
 - information, 44
 - rang, 44
 - sous-rang, 52
- générateur, 88

- grammaire, 11
- graphe
- composé, 106
 - connexe, 67, 68, 108
 - de Cayley, 88
 - enraciné, 108
 - hauteur, 96
 - imbriqué, 109
 - largeur, 96
 - niveau, 96, 111
 - non orienté, 67
 - orienté, 68
 - sous-graphe, 67, 68
- inclusion, 108
- langage, 11
- lien, 70, 106
- à mémoire, 122
 - réflexif, 108
- longueur, 108
- mode
- d'inclusion, 109
 - propre, 124
- mot dérivé, 11
- multi-constructeur, 10
- niveau d'inclusion, 108
- nœud, 67, 68, 106
- fls, 68
 - fls droit, 68
 - fls gauche, 68
 - hauteur, 68
 - père, 68
 - scindé, 112
 - temporaire, 99, 123
- non terminaux, 11
- opération, 11
- ordonnancement de listes, 96
- origine, 106, 108
- pas, 35
- père d'inclusion, 108
- placement au plus près, 71
- points, 35
- production, 11
- racine, 108
- rang inverse, 41
- arbre, 56
 - fonction, 41
- simplexe, 99
- spécification, 1, 11
- bien fondée, 12
- spring embedder*, 83
- structure, 9
- atomique, 10
 - combinatoire, 10
 - composante, 11
 - composante terminale, 11
 - décomposable, 1
 - élémentaire, 10
 - forme, 43
- suite aliquote, 80
- suite de dénombrement, 13
- symbole d'entrée, 11
- tâche, 96
- terminaux, 11
- type de mode d'inclusion, 109
- univers
- étiqueté, 12
 - non étiqueté, 12
- voisinage
- entrant d'inclusion, 108
 - entrant orienté, 68
 - non orienté, 67
 - sortant d'inclusion, 108
 - sortant orienté, 68

Résumé

L'objet de cette thèse est la réalisation d'algorithmes et d'outils d'aide à l'étude des propriétés de structures combinatoires particulières, les structures décomposables. Nous nous intéressons pour cela à la génération aléatoire et systématique de structures décomposables, puis à leur représentation graphique automatique. Ce travail se situe à la frontière entre calcul mathématique et visualisation.

Les structures décomposables sont les structures combinatoires qu'il est possible de former récursivement en utilisant des constructeurs aux propriétés particulières. Le point de vue est similaire à celui adopté dans la théorie des espèces de structures, où l'on privilégie la description d'ensembles de structures à partir de transformations d'ensembles existants. Il est alors possible, grâce à des spécifications, de décrire une infinité d'ensembles de structures combinatoires parmi lesquels les permutations, les graphes fonctionnels, les arbres enracinés ou encore les hiérarchies. L'intérêt de cette démarche tient au fait que l'on sait résoudre des problèmes de dénombrement et de comportement asymptotique sur ces ensembles et générer aléatoirement de façon uniforme des structures de ces ensembles. Les applications concernent le calcul de complexité en moyenne d'algorithmes, et la génération de jeux de tests pour la validation expérimentale ou l'étalonnage d'algorithmes.

Nous présentons dans cette thèse deux types de résultats. Les premiers concernent la génération de structures décomposables, les seconds leur représentation graphique. Nous présentons une implantation d'un algorithme classique de génération aléatoire de structures décomposables, et nous proposons des techniques permettant de générer tous les éléments d'un ensemble à partir de sa spécification. Nous proposons également un algorithme de tracé de graphes particuliers, pour lesquels il existe à la fois des relations d'adjacence et d'inclusion entre les nœuds. Ces graphes, que nous appelons les graphes composés, sont en effet bien adaptés à la représentation de la nature générique des structures décomposables. Ce travail est concrétisé par la réalisation de deux logiciels de tracé de structures combinatoires. Leur utilisation n'est cependant pas limitée à ce seul domaine et les aspects liés à leur application à la visualisation de graphes en général sont abordés.

Mots-clés: combinatoire, structures décomposables, génération aléatoire, rang inverse, tracé de graphes, graphes composés.

Abstract

The main goal of the thesis is to conceive algorithms and tools to assist people who study a special kind of combinatorial structures, namely decomposable structures. The two major questions we try to solve are:

- How to generate, randomly or by some systematic procedure, a decomposable structure.
- How to draw decomposable structures.

Decomposable structures are combinatorial structures that are recursively described using a small set of constructors. The idea consists in considering a structure as a process of construction from simpler structures. The main interest of the decomposable structure theory is that we can describe an infinite number of different sets of structures, including permutations, various kind of trees or functional graphs, for which we can solve counting and random generation problems. Possible applications are the average case analysis of algorithms and the production of test inputs for the experimental validation of programs.

We propose an algorithm for drawing decomposable structures based on the translation into specialgraphs, that we call composed graphs, in which both inclusion and adjacency relationships can exist. The principle is based on the translation of decomposable structures into composed graphs, i.e. graphs with both inclusion and adjacency relationships. The drawing of composed graph is achieved by using different classical graph drawing algorithm together. The number of algorithms that can be used on a same drawing is infinite. The only restriction is that the algorithms must be able to draw graphs with arbitrary node sizes.

We present two graph drawing software realisations that we wrote in order to validate the algorithms presented in the thesis. They can be linked to combinatorial structure generation programs in order to form integrated systems. We also investigate their use for the visualisation of large data structures.

Keywords: combinatorics, decomposable structures, random generation, unranking, graph drawing, clustered graphs

